Hugo Miguel Oliveira Romualdo Simões

Amortised Resource Analysis for Lazy Functional Programs

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

Fevereiro de 2014

Hugo Miguel Oliveira Romualdo Simões

Amortised Resource Analysis
for
Lazy Functional Programs

Tese submetida à Faculdade de Ciências da Universidade do Porto
para obtenção do grau de Doutor em Ciência de Computadores
Supervisors: Prof. Mário Florido and Prof. Kevin Hammond

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Fevereiro de 2014

To my wife and sons.

Acknowledgements

I would like to express my deepest thanks to the people who directly contributed to the conclusion of this thesis. First, I would like to thank my supervisors Mário Florido and Kevin Hammond for their encouragement, support and optimism. I especially thank Kevin and his wife for a warm welcome and making me feel at home during my stay in bonnie St Andrews together with my wife.

My thanks extend to the functional programming group in St Andrews for valuable discussions and, in particular, I would also like to thank Steffen Jost and Armelle Bonenfant, and their respective families, for our hiking trips across Scotland and for putting our shared interests in board gaming into practice.

A very special thanks goes to my friends and colleagues Steffen Jost and Pedro Vasconcelos for their continuous help in pursuing a practical approach to the problem of resource analysis for lazy functional programs. Our long collaboration formed the basis for this thesis.

I would like to thank Mário, Kevin, Steffen and Pedro for reviewing drafts of this thesis, with special thanks to Sandra Alves and Olivier Danvy for also actually volunteering for that task.

Many thanks to the external examiners present at my viva, Vasco Thudichum Vasconcelos and Ricardo Peña, for their kind comments and interesting observations.

After my research grant was over, I was able to regularly work on my thesis, while developing mobile applications, thanks to Luís Damas and Michel Ferreira at Geolink Lda. Similarly, I would like to thank Eduardo Carqueja at AppGeneration for gracefully handling my indecision over setting the end date of my leave of absence while I was finishing writing this thesis.

Financial support is acknowledged from the “Fundação para a Ciência e Tecnologia”, for the Ph.D. grant SFRH/BD/17096/2004 and for a research grant at project RESCUE (REliable and Safe Code execUtion for Embedded systems) PTDC/EIA/65862/2006, and also from the LIACC (Laboratory of Artificial Intelligence and Computer Science) of the University of Porto, Portugal.

Finally, I thank my wife, not only for her unconditional support during this long Ph.D. period, but also for sharing the happiest days of my life together with our three sons. To happiness!

Resumo

Esta tese descreve a primeira tentativa bem-sucedida, de que temos conhecimento, de definir uma análise estática, automatizada e baseada em sistemas de tipos, capaz de encontrar majorantes relativos à quantidade de recursos utilizados em programas funcionais *lazy*. A avaliação lazy permite melhorar a composição de programas, mas dificulta quase sempre as previsões de recursos. A nossa análise utiliza a abordagem de amortização automatizada desenvolvida por Hofmann e Jost, que estava anteriormente restringida à avaliação *eager*. Nesta tese, estendemos este trabalho a sistemas lazy através da captura em anotações de tipos dos custos de expressões por avaliar e da amortização do pagamento destes custos utilizando uma noção de *potencial lazy*. Apresentamos a nossa análise como um sistema de demonstração que prevê (em tempo de compilação) a quantidade total de alocações de memória *heap* de uma linguagem funcional mínima (incluindo funções de ordem superior e tipos de dados recursivos) e definimos um modelo de custos formal baseado na semântica de Launchbury para avaliação lazy. Provamos a correção da nossa análise face ao modelo de custos. A nossa abordagem é ilustrada através de derivações de tipos de exemplos representativos e não triviais, que foram analisados utilizando um protótipo da implementação da nossa análise.

Palavras-chave: avaliação lazy, análise amortizada, análise de recursos, sistema de tipos, call-by-need, análise estática

Abstract

This thesis describes the first successful attempt, of which we are aware, to define an automatic, type-based static analysis of resource bounds for lazy functional programs. Lazy evaluation allows improved modularity of programs, but often makes resource usage difficult to predict. Our analysis uses the automatic amortisation approach developed by Hofmann and Jost, which was previously restricted to eager evaluation. In this thesis, we extend this work to a lazy setting by capturing the costs of unevaluated expressions in type annotations and by amortising the payment of these costs using a notion of *lazy potential*. We present our analysis as a proof system for predicting (at compile-time) total heap allocations of a minimal functional language (including higher-order functions and recursive data types) and define a formal cost model based on Launchbury's natural semantics for lazy evaluation. We prove the soundness of our analysis with respect to the cost model. Our approach is illustrated by type derivations of a number of representative and non-trivial examples that have been analysed using a prototype implementation of our analysis.

Keywords: lazy evaluation, amortized analysis, resource analysis, type system, call-by-need, static analysis

Contents

List of Figures

List of Theorems and Definitions

1. Introduction

Non-strict functional programming languages, such as Haskell [PAB$^+$99], offer important benefits over more conventional eagerly-evaluated languages in terms of modularity and abstraction [Hug89] through exploiting lazy evaluation. A key practical obstacle to their wider use, however, is that extra-functional properties, such as time- and space-behaviour, are often difficult to determine prior to actually running the program. This is largely because the effects of lazy evaluation are hard to predict without actually running a program, since evaluation order is determined dynamically: reduction is carried out if and when it is found to be needed and consequently memory is allocated only if and when needed. Given this difficulty, providing guarantees about memory usage or time performance would both increase confidence in software reliability and performance of lazily-evaluated programs, and open new resource-critical applications such as real-time, memory-limited systems.

Recent advances in static cost analyses, such as *sized types* [VH05, SHFV07, Vas08] and *type-based amortisation* [HJ03, HAH11] have enabled the *automatic* prediction of resource bounds for eager functional programs, including uses of higher-order functions [JLHH10]. This thesis develops a new mechanism, *lazy potential*, that allows execution costs to be transferred from one point of a program to another, as part of an *amortised analysis*. By exploiting this mechanism, we are then able to extend type-based amortisation to lazy evaluation, describing a static analysis for determining *a-priori* worst-case bounds on execution costs (specifically, dynamic memory allocations).

Our amortised analysis derives costs with respect to a cost semantics for lazy evaluation that derives from Launchbury's natural operational semantics of graph reduction [Lau93]. It deals with both first-order and higher-order functions, but does not consider polymorphism. Moreover, the analysis is *compositional*, i.e. it can be applied to program fragments as well as to complete programs. For simplicity, we restrict our attention to total heap allocation, but previous results have shown that the amortised analysis approach also extends to other countable resources, such as worst-case execution time [JLH$^+$09]. In order to ensure a good separation of concerns, our analysis assumes the availability of Hindley-Milner type information [Mil78]. We extend Hofmann and Jost's type annotations for capturing *potential* costs [HJ03] with information about the latent costs of unevaluated expressions. The analysis produces a set of constraints over cost variables that we solve in our prototype implementation using an external LP-solver. We have thus demonstrated all the steps that are necessary to produce a fully-automatic analysis for determining bounds on resource usage for lazily-evaluated programs.

Although we do not directly address the issue of algorithmic type reconstruction in this thesis, a prototype implementation[*] and previous work in the strict setting [HJ03, JLHH10, HAH11] suggests that our analysis should be *fully automatable*, e.g. by performing a standard Damas-Milner type inference [DM82] with types decorated with fresh annotation variables and producing a set of linear inequalities that can then be automatically solved by a standard LP solver. No guidance from the programmer is necessary.

1.1 Contributions

This thesis makes the following novel contributions:

- we present the first successful attempt, of which we are aware, to produce an automatic, efficient, type-based, static analysis with formally guaranteed data-dependent resource bounds for lazy evaluation;
- we introduce a cost model for heap allocations for a lazy functional language based on Launchbury's natural semantics for lazy evaluation [Lau93], and use this as the basis for developing a resource analysis;
- we prove the soundness of our analysis with respect to the cost-instrumented semantics;
- we develop an analysis for eager functional programs with the purpose of better contrasting the analysis for laziness; and

[*] Pedro Vasconcelos implemented in Haskell a publicly accessible web-prototype for our analysis (available at `http://www.dcc.fc.up.pt/~pbv/cgi/aalazy.cgi`) — a much welcome relief from the burden of manually testing program examples.

- we demonstrate the effectiveness of the analysis by deriving costs for some non-trivial examples.

The research on which this thesis is based was done in collaboration with others. In particular, the automatic amortised analysis for lazily-evaluated functional programs has previously been reported in a published paper [SVF$^+$12] which was jointly authored by Pedro Vasconcelos, Steffen Jost, my two supervisors Mário Florido and Kevin Hammond, and myself: Hugo Simões, Pedro Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'12)*, pages 165–176, Copenhagen, Denmark, September 2012. The technical differences to the published paper are that this thesis:

- fixes a minor problem in the soundness proof (caused by rule LET of our type system);
- changes the language to be compatible with Launchbury's semantics (replaces match with case expressions, removes parentheses of constructor applications and merges letcons with let expressions);
- simplifies annotations by replacing the double cost annotations with a single cost annotation (this is possible since we are analysing a monotonic resource: total heap allocations);
- restricts the inversion lemmas of Section 5.5.1 to have zero on the turnstile of the type judgements (otherwise those lemmas would not hold);
- adds a side-condition to rule WEAK of our type system;
- contrasts the lazy system with an eager system that is specifically tailored to emphasise the key elements of the novel analysis; and
- illustrates the effectiveness of the analysis with detailed derivations of some non-trivial examples;

Note that meanwhile the soundness proof was double checked in detail, since the first five items above forced almost all of the previous technical work (including proofs) to be rewritten in this thesis.

Also in the course of his PhD plan, during the introductory studies on the field of static resource analysis, the author contributed to another paper [SHFV07]: Hugo R. Simões, Kevin Hammond, Mário Florido, and Pedro Vasconcelos. Using Intersection Types for Cost-Analysis of Higher-Order Polymorphic Functional Programs. In Thorsten Altenkirch and Conor McBride, editors, *Revised Selected Papers of the International Workshop on Types for Proofs and Programs (TYPES'06), Nottingham, UK, April, 2006*, volume 4502 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2007. This paper improves the quality of a previous analysis for eagerly evaluated programs by showing how discrete polymorphism helps reduce the problem of *size aliasing*. However, since it is not a direct contribution to the field of analysis for lazy evaluation (the core topic of this thesis), the result is simply referenced here.

1.2 Overview

In the remainder of this thesis we start by reviewing some related work in Chapter 2. Next, in Chapter 3, we review some background on *amortisation*, covering the description of the general technique and its application to type-based analyses.

Then, in Chapter 4, we define a simple functional language and present a cost model for measuring the total heap allocations under a call-by-need semantics of programs written in this language.

In Chapter 5 we develop a type-based amortised analysis for lazy evaluation and provide a soundness proof as the main contribution of this thesis, guaranteeing that the cost bound of the analysis is observed with respect to the cost model.

An experimental assessment of the analysis is given in Chapter 6 through a range of illustrative examples.

Finally, Chapter 7 concludes.

2. Related Work

2.1 Semantics for Lazy Evaluation

We build heavily on Launchbury's natural semantics for lazy evaluation [Lau93], as subsequently adapted by Sestoft [Ses97], and exploit ideas that were developed by Encina and Peña [EP02, EP03a]. There is a significant body of other work on the semantics of call-by-need evaluation. Pre-dating Launchbury's work, Josephs [Jos89] gave a *denotational* semantics of lazy evaluation, using a continuation-based semantics to model sharing, and including an explicit store. However, this approach does not fit well with standard proof techniques. Maraist et al. [MOW98] subsequently defined both natural and reduction semantics for the call-by-need lambda calculus, so enabling equational reasoning, and a similar approach was independently described by Ariola and Felleisen [AF97].

Like Encina and Peña [EP03a, EP09], Mountjoy [Mou98] derived an operational semantics for the Spineless Tagless G-Machine from the natural semantics of Launchbury and Sestoft, including poly-applicative λ-expressions. The main differences between these approaches are that Encina and Peña correct some mistakes in Mountjoy's presentation; that they provide correctness proofs; that their semantics correctly deals with partial applications in the Spineless Tagless G-Machine; that they deal with partial applications as normal forms; and that they consider two distinct implementation variants, based on push/enter versus apply/eval. More recently, Pirog and Biernacki [PB10] have established the equivalence between the Spineless Tagless G-Machine and an extended version of the natural semantics of Launchbury and Sestoft as evidenced by Danvy et al.'s [ADM04] functional correspondence between abstract machines and evaluators.

Bakewell and Runciman [BR01] have previously defined an operational semantics for Core Haskell that gives time and space execution costs in terms of Sestoft's semantics for his

Mark 1 abstract machine. The work has subsequently been extended to give a model that can be used to determine space leaks by comparing the space usage for two evaluators using a bisimulation approach [BR00]. Gustavsson and Sands [GS99] have similarly defined a space-improvement relation that guarantees that some optimisation can never lead to asymptotically worse space behaviour for call-by-need programs and Moran and Sands [MS99] have defined an improvement relation for call-by-need programs that can be used to determine whether one terminating program improves another in all possible contexts.

Finally, given that compilers for lazy evaluation eventually generate optimised code based on information from *strictness analysis* [Myc81, BHA86, MN92, WH87] or *cheapness analysis* [Myc80, Fax00] and thus implement in fact a non-strict semantics rather than call-by-need, it is worth noting an alternative non-strict reduction strategy by Ennals et al. [EP03b, Enn03], called *optimistic evaluation*, that, in an attempt to improve the average time performance against call-by-need, is based on speculatively evaluating expressions that are considered to be usually used and usually cheap to evaluate and aborting if an embedded profiler determines that it is not the case. Although the approach promised to achieve considerable performance improvements, its development is currently suspended from industry-strenght compilers given the difficulties in maintaining the supporting framework (i.e. speculation, profiling and abortion) while implementing other features.

Our own work differs from this body of earlier work in that we provide a cost semantics from which we derive a static analysis to automatically determine upper bounds on the memory requirements of lazily evaluated programs.

2.2 Resource Analyses for Lazy Evaluation

Resource analysis based on profiling and manual code inspection has long formed the state-of-the-art and still is current practice in many cases. Indeed, for non-strict functional languages, such as Haskell, ad-hoc techniques, manual analysis or symbolic profiling are the only currently viable approaches: the dynamic *demand-driven* nature of lazy functional programming creates particular problems for resource analysis, whether manual or automatic. There has therefore been very little work on static resource analysis for lazy functional programs, and, to our knowledge, no *previous automatic static analysis has ever been produced*. The most significant previous work in the area is that by Sands [San90a, San90b], whose PhD thesis proposed a cost calculus for reasoning about sufficient and necessary execution time for lazily evaluated higher-order programs, using an approach based on *evaluation contexts* [Wad88, San98] to capture information about evaluation degree and appropriate *projections* [WH87] to project this information to the required approach. Wadler [Wad88] had earlier proposed a similar approach to that taken by Sands, but limited to first-order functions and using only *strictness analysis* combined with appropriate projections, rather than the *neededness analysis* that Sands also uses. Around the same time, Bjerner and Holmström [BH89] developed an approach using *demand analysis* which requires, *a-priori*, a domain structure describing an approximation of the output of the analysed program. A primary disadvantage of such approaches lies in the complexity of the domain structure and associated projections that must be used when analysing even simple data structures such as lists. In contrast, our approach easily extends to algebraic data structures. A secondary disadvantage is that a demand analysis approach requires knowing in advance much information about the output value and, unlike the self-contained analysis we have described, projection-based approaches rely on the existence of a complex and powerful external *neededness analysis* to determine evaluation contexts for expressions. These are serious practical disadvantages: in fact, to date, we are not aware of any fully automatic static analysis that has been produced using these techniques.

Transforming lazy programs into eager ones would be a possible approach to producing an analysis for lazily evaluated programs. The resulting programs would then be analysed using (simpler) techniques for eagerly evaluated programs. Unlike our work, these approaches would suffer from the problems that they would produce very poor quality bounds (many programs requiring a small finite amount of resources under lazy evaluation, would require an infinite amount if evaluated eagerly), that they would be, in general, not cost-preserving, that they would lead to potentially exponential code explosion, and that, because they would alter the program, they would not be suitable for use with standard compilers for lazy functional languages. Perhaps because of such drawbacks, no one appears to have actually done this.

Several authors have proposed approaches where programs are annotated with additional cost parameters. For example, Albert et al. [ASV03] describes how to automatically construct recurrence relations by adding extra cost parameters to each function under a call-by-name semantics and suggests extending the approach to call-by-need through an additional linearisation phase together with guarded constraints (to handle sharing and so avoid cost duplication); and Hope [Hop08] describes how to derive an instrumented function for determining time and space usage, including a simple deallocation model, for a strict functional language and outlines how this could be extended to lazy evaluation. By constrast, our work is capable of inferring cost bounds. Also, unlike Albert et al. [ASV03]'s work, our system deals directly with higher-order programs, as opposed to using program transformation techniques such as *defunctionalisation* [Rey72] which are, in general, not cost-preserving and require a whole-program analysis.

Another approach followed by Wadler [Wad92] uses *monads* to capture execution costs through a tick-counting function; Danielsson [Dan08] takes this work a stage further, describing a library that can be used to annotate (lazy) functions with the time that is needed to compute their result. An annotated monad is then used to combine these time complexity annotations. This can be used to verify the time complexity of (lazy) functional data structures and algorithms against Launchbury's semantics, using a dependent type approach. However, some of the annotations must be manually introduced by the programmer and that may require ingenuity. Moreover, unlike our work, the system is not capable of inference.

2.3 Amortised Analyses

The amortised analysis approach has been previously studied by a number of authors, but has never previously been used to automatically determine the costs of lazy evaluation. Tarjan [Tar85] first described amortised analysis, but as a manual technique. Okasaki [Oka98] subsequently described how Tarjan's approach could be applied to (lazy) data structures, but again as a manual technique. While there has subsequently been significant interest in the use of amortised analysis for automatic resource usage analysis, using an advanced per-reference potential, none of this newer work, however, considers lazy evaluation. Hofmann and Jost [HJ03] were the first to develop an *automatic* amortised analysis for heap consumption, exploiting a difference metric similar to that used by Crary and Weirich [CW00] (the latter, however, only *check* bounds, and therefore does not perform an automatic static analysis of the kind we require); Hofmann et al. have extended their method to cover a comprehensive subset of Java, including imperative updates, inheritance and type casts [HJ06, HR09]; Shkaravska et al. [Svv07] subsequently developed a polynomial heap consumption analysis for first-order polymorphic lists, with restricted (*shapely*) functions, but did not consider the efficiency of the suggested inference; in the meantime, Herrmann, Bonenfant et al. [HBH$^+$07] showed an automatic amortised analysis for *worst-case execution time*; and Campbell [Cam08, Cam09] has developed the ideas of depth-based and temporary credit uses to give better results for stack usage. Jost et al. [JLH$^+$09, JLHH10] significantly extended previous analyses by dealing with higher-order, polymorphic functions, varying resource metrics, arbitrary recursive data types, the creation of circular data, and the possibility of directly adding constraints on resource annotations in types through resource parametric functions; later, Hoffmann et al. [HH10, HAH11] achieved another breakthrough by extending the technique to infer (multivariate) polynomial cost functions, still only requiring efficient LP solving.

The analysis presented in this thesis is yet another member of the Hofmann and Jost based family of amortised analyses and can be further put into context by Figure 2.1 which extends a related figure from Jost's PhD thesis. The work presented here corrects a minor technical problem found in the soundness proof of Simões et al. [SVF$^+$12] and, except for *object-orientation* and *imperative update* (which are unrelated to the purely functional language of this thesis), all remaining features not handled by our analysis are discussed in Section 7.2 as further work.

2.4 Other Heap Analyses for Eager Evaluation

Finally, several authors have recently studied analyses for heap usage in eager languages, without considering lazy evaluation. For example, Albert et al. [AGG09] present a fully automatic, live heap-space analysis for an object-oriented bytecode language with a scoped-memory manager, and have subsequently extended this to consider garbage collection [AGG10], but, unlike our system, data-dependencies cannot be expressed. Braberman et al. [BFGY08] infer polynomial bounds on the live heap usage for a Java-like language with automatic memory management, but do not cover general recursive methods. Finally, Chin et al. [CNPQ08] present a linearly-bounded heap and stack analysis for a low-level (assembler) language with explicit (de)-allocation, but do not cover lazy evaluation or high-level functional programming constructs.

Feature	[HJ03]	[HJ06]	[Svv07]	[HBH$^+$07]	[Cam08, Cam09]	[JLH$^+$09]	[JLHH10]	[Jos10]	[HH10, HAH11]	[SVF$^+$12] and This Thesis
full recursion	+	+	+	+	+	+	+	+	+	+
aliasing	+	+		+	+	+	+	+	+	+
inference	+		?	+	+	+	+	+	+	+
object-orientation		+								
imperative update		+								
non-termination					+			+		
total heap allocation usage	+	+	+		+	+	+	+	+	+
heap usage (w/ deallocation)	+	+			+	+	+	+	+	
worst-case execution time				+		+	+		+	
stack usage					+	+	+		+	
varying resource metrics						+	+		+	
arbitrary recursive data types						+	+			+
polymorphism							+			
resource parametricity							+	+		
higher-order							+	+		+
creating circular data							+			+
delayed execution								+		+
requires shapely functions (*)			–							
super-linear bounds			+						+	
laziness										+

(*) Requiring *shapely functions* is actually not a feature, but a limitation.

Figure 2.1: Family feature comparison

3. Amortisation

3.1 Classical Amortisation Technique

First described by Tarjan [Tar85], *amortisation* is a technique in the field of complexity analysis of algorithms. It is a manual method that tries to take advantage of the correlated effects of a sequence of operations on a data structure in order to obtain tighter bounds than for example the sum of the worst-case costs of each operation in the sequence.

There are two equivalent views of amortisation. We will focus hereafter on the so-called physicist's view as it better serves the intuition behind the analysis described in this thesis.

To use the general technique, we define a potential function Φ mapping any configuration of a data structure to a number, henceforth referred to as the *potential* of that configuration. The amortised cost a_i of an operation to a data structure is then defined as the actual cost t_i of the operation plus the difference between the potential of the configuration of the data structure after the operation Φ_i and the potential of the configuration before the operation Φ_{i-1}.

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

In a sequence of n such operations the following equality holds:

$$\sum_{i=1}^{n} t_i = \sum_{i=1}^{n} (a_i - \Phi_i + \Phi_{i-1}) = \Phi_0 - \Phi_n + \sum_{i=1}^{n} a_i$$

If we ensure that potential is always non-negative then the potential of the initial configuration plus the sum of the amortised costs provide an upper bound on the actual cost of the sequence.

$$\sum_{i=1}^{n} t_i \;\leq\; \Phi_0 + \sum_{i=1}^{n} a_i, \text{ if } \Phi_i \geq 0 \text{ for all } i$$

By cleverly defining the potential function, the goal is to further simplify the bounding expression by making the amortised costs zero or at least (bounded by a) constant, thus being able to easily bound the fluctuations of the successive actual operation costs.

3.1.1 Example: Analysing a Stack

To better understand the intuition and the application of the amortisation technique the following example[*] will be used:

Consider the manipulation of a stack using the two standard primitives: *push*, which adds a new element to the top of the stack, and *pop*, which returns and removes the top element from the stack.

Now consider an additional compound operation consisting of applying any number of pops followed by exactly one push. Starting with an empty stack, we would like to analyse the cost — in terms of the number of pushes and pops — of a sequence of n such compound operations.

The worst-case cost of a single operation is n, corresponding to the case where no pops occur in the first $n-1$ operations and the last operation applies $n-1$ pops followed by the mandatory push. So, although the cost of each of the first $n-1$ operations is 1 (a push), the cost of the last operation is n ($n-1$ pops plus 1 push).

Compare a worst-case analysis in which we sum the worst-case cost of a single operation for each operation in the sequence, obtaining $n * n$, to the following worst-case analysis using amortisation.

Define the potential of a stack to be the number of elements it contains. It follows that, if a stack has m elements, the amortised cost of an operation that pops k elements followed by a push is $(k+1)+(m-k+1)-m=2$. Since the potential is always non-negative (by definition) and the initial potential is zero (we start with an empty stack), we know that the actual cost of the sequence of n operations is bounded by $\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} 2 = 2n$.

[*] Due to Tarjan [Tar85].

We know that in a sequence of n operations, starting with an empty stack, we have exactly n pushes (one per operation). Given that each pop must correspond to an earlier push, we can have at most n pops as well. In fact, since push is the last primitive applied in an operation, we cannot pop the last element and so we have at most $n-1$ pops. Thus, the maximum number of pushes and pops in a sequence of n operations is $n+(n-1)=2n-1$.

The amortised analysis of this example allowed us to obtain a tight worst-case bound $2n$ of the actual worst-case cost $2n-1$.

3.2 Automatic Amortised Analysis

As a manual technique, amortisation has two shortcomings. Firstly, it requires ingenuity when defining a useful mapping from each configuration to a number representing its potential (since it is unfeasible to try all possible mappings), thus restricting the widespread use of the technique. Secondly, as Okasaki [Oka98] notes, "traditional methods of amortization break in presence of persistence". This represents a problem, given that persistent data structures are commonly found in functional settings.

A type-based approach solves both of these issues. It has been successfully applied [HJ03, Cam09, JLH$^+$09, JLHH10, HH10, HAH11] as a way not only to provide a means to automatically determine a suitable potential function, but also to deal with persistent data structures (by assigning potential on a per-reference basis, instead of resorting to a lazy evaluation strategy as in Okasaki's approach [Oka98]).

The first type-based automatic amortised analysis was developed by Hofmann and Jost [HJ03] for analysing the heap-space consumption of first-order eager functional programs. Although at that time unaware of the connection to Tarjan's work [Tar85], their goal was to produce an automatic analysis that could find bounds to resource usage at the press of a button. For that purpose, their fundamental idea was to collect linear inequalities arising from the side conditions of a type derivation and then solve them with an LP-solver (such as the *glpk*[†]). The main limitation was the expressiveness of the bounds — the potential function was linearly tied to the number of nodes of data structures and since the analysis depends on the potential of the initial configuration, it could only hope to find linear bounds as well — but the end result in itself, and the program examples that could be successfully

[†] `http://www.gnu.org/software/glpk/`

analysed, made the approach interesting.

Since then, keeping the fundamental idea, their technique has been successfully applied in the analyses of stack usage [Cam09], generic resource metrics [JLH$^+$09], higher-order and polymorphic functions [JLHH10] and in efficiently finding multivariate polynomial bounds [HAH11] through using non-linear potential functions.

3.2.1 Informal Description

In the classical amortisation technique, the first step in developing an amortised analysis is to define the potential function — the mapping from configurations to numbers. In Hofmann and Jost's approach, this corresponds to defining the annotated types the type system will handle. The annotated data types, in particular, carry the contributions of a node in a particular data structure to the overall potential of the memory configuration. For example, a red-black binary tree [Bay72] is a binary tree data structure that is easier to maintain balanced than its regular counterpart. It consists of three possible constructors: a `Red` and a `Black` binary constructors having a left and a right red-black binary tree as arguments, and a zero-arity `Leaf` constructor. Consider the following annotated data type for red-black binary trees of `Ints`:

$$\texttt{RBTree}(q_r, q_b, q_l, \texttt{Int})$$

In a tree with this type, where q_r, q_b and q_l are non-negative rational numbers, each `Red` and `Black` node contributes with q_r and q_b, respectively, and each `Leaf` node contributes with q_l to the potential of the tree. Given a tree with n_r red nodes, n_b black nodes and n_l leaf nodes, the potential of such tree is $n_r \times q_r + n_b \times q_b + n_l \times q_l$. Note that the potential of the tree is linear with respect to its number of nodes. Restricting to linear potential with respect to the number of constructors in a data structure is common in type systems following the approach of Hofmann and Jost, with a notable exception [HH10, HAH11]. Since our main concern here is to extend the approach to a lazy setting, we keep the linear restriction, leaving as further work the adoption of super-linear bounds in our analysis.

Also, recall from Section 3.1 that the goal of any amortised analysis is to find a constant that bounds the fluctuations of the successive actual operation costs (in order to simplify the overall bounding expression). That is the purpose of the annotated type systems following Hofmann and Jost's approach: to ensure the amortised costs are zero, so that the potential of the initial configuration is an upper bound of the overall actual cost.

Once the type system is defined, these type-based amortised analyses obtain their result automatically by performing the following 4 steps:

1) perform a Damas-Milner type inference [DM82] to obtain a type derivation (without annotation variables);

2) decorate the Hindley-Milner types [Mil78] with fresh annotation variables;

3) traverse the type derivation, gathering linear constraints among annotation variables according to the rules of the type system;

4) feed the linear constraints to a standard linear programming solver with the objective of minimising the overall expression cost.

Note that only the first or the last step may fail, i.e. either the program being analysed is not well-typed or the gathered linear constraints cannot be solved.

Each solution to the generated linear program corresponds to a particular bound on the execution cost. However, these bounds are then only useful provided a correctness guarantee exists. As such, a soundness proof is the key result of these systems, since it establishes the link between cost model and type system. This ensures the run-time actual costs never exceed the compile-time predicted bounds.

It is important to note that the analysis produces data-dependent bounds. For example, using an automatic amortised analysis, Loidl and Jost [LJ09] learned that insertion, in their cost model, is generally more expensive for a red-black tree having many black nodes, since coefficient q_b was about 3 times higher than q_r.

In this thesis we present a type-based amortised analysis for lazy functional programs following Hofmann and Jost's approach and show its complete development in Chapter 5 — from the chosen annotated types, to the invariants required for the soundness proof.

4. Cost Model

In this chapter we present a cost model that allows us to measure total heap allocations. It is given as an operational semantics that formalises the cost of evaluating an expression. We define a cost model for two reasons: to prove the soundness of our analysis (Chapter 5), i.e. to prove that evaluating an expression never costs more than the analysis predicted, and to measure the quality of our analysis against a range of examples (Chapter 6), i.e. to compare the costs of evaluating an expression with the costs predicted by the analysis for the same expression.

The cost model we present is built on Encina and Peña's corrected version [EP02] of Sestoft's revision [Ses97] of Launchbury's natural semantics for lazy evaluation [Lau93]. Launchbury's semantics forms one of the earliest and most widely-used operational accounts of lazy evaluation for the λ-calculus. Encina and Peña [EP02] [EP03a] subsequently proved that the *Spineless Tagless G-Machine* [Jon92] is sound and complete with respect to one of Sestoft's abstract machines. More recently, Pirog and Biernacki [PB10] have established the equivalence between the Spineless Tagless G-Machine and their extended version of the natural semantics of Launchbury and Sestoft. This equivalence is evidenced by Danvy et al.'s [ADM04] functional correspondence between abstract machines and evaluators. We therefore have a high degree of confidence that the cost model for lazy evaluation developed in this thesis is not just theoretically sound, but also that it could, in principle, be extended to model real implementations of lazy evaluation, such as the GHC implementation of Haskell.

Before looking at the cost model in Section 4.3, we will see in detail the operational semantics on which it is based. However, we first need to define the language to be used on both the cost model and the analysis.

4.1 Language Syntax

The Fun language (Figure 4.1) is similar to the one found in Sestoft's revision [Ses97] of Launchbury's natural semantics for lazy evaluation [Lau93]. The reader unfamiliar with the mentioned references should note that arguments to both applications and constructor applications are restricted to variables and that this can be achieved through a process called *normalisation* [Lau93], which consists of naming the arguments using *let* expressions. We have thus a normalised λ-calculus extended with (possibly recursive) local bindings, (saturated) constructor applications and *case* expressions.

In contrast to Launchbury and Sestoft's language, we consider only (for simplicity) single-variable let-bindings (multiple let-bindings can be encoded, if needed, using pairs and projections). Also, constructor applications appear only in let-bindings as in Encina and Peña's semantics for lazy evaluation [EP09]. However, Encina and Peña's motivation for such restriction was different from ours: they wanted to be as close as possible to the STG language, while we simply need to distinguish between *allocating* a constructor and merely *referencing* an existing one, since these are handled differently by our analysis.

As in Sestoft's language, we do not require bound variables (either lambda-, let- or case-bound) to be distinct, except that, for each case expression, each element in multiset $\{\overrightarrow{x_i}\}$ must be distinct, for $i = 1, \ldots, n$. For example,

$$\text{case } e \text{ of } c_1\ x\ y \text{ -> } x,\ c_2\ y \text{ -> } y$$

would be a valid program, whereas the following would not

$$\text{case } e \text{ of } c_1\ x\ x \text{ -> } x,\ c_2\ y \text{ -> } y$$

4.2 Operational Semantics

Our big-step operational semantics is based on Launchbury's natural semantics for lazy evaluation [Lau93], as subsequently adapted by Sestoft [Ses97], as corrected for case expressions by Encina and Peña [EP02]. Figure 4.2 shows the set of rules that define our operational semantics.

$$
\begin{array}{rcll}
 & & \text{– Variables} & \\
v & ::= & x \;\mid\; y & \text{– bound variable} \\
 & \mid & l & \text{– free variable (location)} \\
 & & & \\
 & & \text{– Expressions} & \\
e & ::= & v & \text{– variable} \\
 & \mid & \lambda x.\, e & \text{– lambda abstraction} \\
 & \mid & e\; v & \text{– application} \\
 & \mid & \text{let } x = \widehat{e} \text{ in } e & \text{– (possibly recursive) let-binding} \\
 & \mid & \text{case } e \text{ of } \{c_i\ \overrightarrow{x_i} \text{ -> } e_i\}_{i=1}^{n} & \text{– case expression} \\
 & & & \\
 & & \text{– Augmented expressions} & \\
\widehat{e} & ::= & c\ \vec{v} & \text{– (saturated) constructor application} \\
 & \mid & e & \text{– expression} \\
 & & & \\
 & & \text{– Weak head normal forms} & \\
w & ::= & \lambda x.\, e & \text{– lambda abstraction} \\
 & \mid & c\ \vec{l} & \text{– constructor application}
\end{array}
$$

Figure 4.1: Language Fun

Judgements of the form $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \widehat{e} \Downarrow w, \mathcal{H}'$ should be read as "in the heap $\mathcal{H}$, (augmented) expression $\widehat{e}$ evaluates to *whnf* (weak head normal form) w, producing the new heap $\mathcal{H}'$", where a *heap* is a partial function mapping distinct variable names to *thunks* and a *thunk* is an augmented expression (bound in the heap) that may be further evaluated to *whnf*. Note that, as usual (and seen in Figure 4.1), weak head normal forms are expressions whose outermost structure is a lambda or a constructor. The auxiliary set $\mathcal{L}$ of *locations* under evaluation was one of the changes introduced by Sestoft[*] to improve the renaming mechanism of Launchbury's semantics. The auxiliary set $\mathcal{S}$ was introduced by Encina and Peña[†] in order to fix a freshness property of Sestoft's rules, and, although in their paper it contains the alternatives of case expressions $\{c_i\ \overrightarrow{x_i} \text{ -> } e_i\}_{i=1}^{n}$, we simply keep the bound variables of such alternatives, since these are sufficient to fix the problem.

We next define the set of bound variables contained in a Fun expression in order to later formalise the notion of freshness of variables.

Definition 4.1 (Bound Variables of Fun Expressions)**.** The *bound* variables of a Fun expression $\widehat{e}$, denoted by $\mathrm{BV}(\widehat{e})$, are defined in the usual way as shown in Figure 4.3.

[*] In [Ses97] this set is called A.

[†] In [EP02] this set is called C.

$$\frac{w \text{ is in } \textit{whnf}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash w \Downarrow w, \mathcal{H}} \qquad (\textsc{Whnf}_{\Downarrow})$$

$$\frac{\ell \notin \mathcal{L} \qquad \mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash \mathcal{H}(\ell) \Downarrow w, \mathcal{H}'}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \ell \Downarrow w, \mathcal{H}'[\ell \mapsto w]} \qquad (\textsc{Var}_{\Downarrow})$$

$$\frac{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash e \Downarrow \lambda x.\, e', \mathcal{H}' \qquad \mathcal{H}', \mathcal{S}, \mathcal{L} \vdash e'[\ell/x] \Downarrow w, \mathcal{H}''}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash e\, \ell \Downarrow w, \mathcal{H}''} \qquad (\textsc{App}_{\Downarrow})$$

$$\frac{\ell \text{ is fresh} \qquad \mathcal{H}[\ell \mapsto \widehat{e}[\ell/x]], \mathcal{S}, \mathcal{L} \vdash e[\ell/x] \Downarrow w, \mathcal{H}'}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \text{let } x = \widehat{e} \text{ in } e \Downarrow w, \mathcal{H}'} \qquad (\textsc{Let}_{\Downarrow})$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{S} \cup \bigcup_{i=1}^{n} \left(\{\overrightarrow{x_i}\} \cup \mathrm{BV}(e_i)\right), \mathcal{L} \vdash e \Downarrow c_k\, \vec{\ell}, \mathcal{H}' \\ \mathcal{H}', \mathcal{S}, \mathcal{L} \vdash e_k[\vec{\ell}/\overrightarrow{x_k}] \Downarrow w, \mathcal{H}''\end{array}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \text{case } e \text{ of } \{c_i\ \overrightarrow{x_i} \text{ -> } e_i\}_{i=1}^{n} \Downarrow w, \mathcal{H}''} \qquad (\textsc{Case}_{\Downarrow})$$

Figure 4.2: Lazy operational semantics

$$\begin{array}{l}
\mathrm{BV}(v) = \emptyset \\
\mathrm{BV}(\lambda x.\, e) = \{x\} \cup \mathrm{BV}(e) \\
\mathrm{BV}(e\ v) = \mathrm{BV}(e) \\
\mathrm{BV}(\text{let } x = \widehat{e} \text{ in } e) = \{x\} \cup \mathrm{BV}(\widehat{e}) \cup \mathrm{BV}(e) \\
\mathrm{BV}(\text{case } e \text{ of } \{c_i\ \overrightarrow{x_i} \text{ -> } e_i\}_{i=1}^{n}) = \mathrm{BV}(e) \bigcup_{i=1}^{n} (\{\overrightarrow{x_i}\} \cup \mathrm{BV}(e_i)) \\
\mathrm{BV}(c\ \vec{v}) = \emptyset
\end{array}$$

Figure 4.3: Bound variables of Fun expressions

The following auxiliary definition of freshness of variables is due to Encina and Peña [EP02]:

Definition 4.2 (Freshness)**.** In a judgement $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \widehat{e} \Downarrow w, \mathcal{H}'$ a variable is *fresh* if it is not in $\mathrm{dom}(\mathcal{H})$ nor $\mathcal{S}$ nor $\mathcal{L}$ and it is not bound in either $\mathrm{ran}(\mathcal{H})$ or $\widehat{e}$.

Expressions in *whnf* (lambda abstractions and constructor applications) are already values and should therefore evaluate to themselves, keeping the heap unchanged. This is reflected in rule $\textsc{Whnf}_{\Downarrow}$.

Rule $\textsc{Var}_{\Downarrow}$ states that in order to evaluate a location ℓ, present in a heap $\mathcal{H}$, we evaluate $\mathcal{H}(\ell)$ with ℓ included in the set of locations under evaluation. If, as a result, we obtain a *whnf* w and a heap $\mathcal{H}'$, then evaluating ℓ in $\mathcal{H}$ evaluates to the same w and the new heap produced is $\mathcal{H}'$ with a mapping updating ℓ to w. Note that once ℓ is updated its subsequent accesses obtain the corresponding *whnf* immediately, effectively implementing sharing of named expressions. Also note that if ℓ depends directly on itself before evaluating to *whnf*, when attempting to evaluate ℓ for the second time, no rule will apply, since ℓ will be marked as being under evaluation in rule $\textsc{Var}_{\Downarrow}$. This situation is known as a "black-hole": a detectably self-dependent infinite loop. In Launchbury's semantics, a black-hole is detected by removing ℓ from the heap before evaluating its contents. Since Sestoft's revision of the semantics, black-holes can equivalently be detected using the set of locations marked as being under evaluation. In this thesis we need to keep ℓ in the heap since the mappings defined for the invariants of our soundness proof in Chapter 5 must apply to all heap locations (regardless of being under evaluation). Thus, we use set $\mathcal{L}$ to detect black-holes (in addition to the benefits that motivated its introduction).

The $\textsc{App}_{\Downarrow}$ rule deals with function applications and, assuming the term is well-typed, evaluation is done in two steps: first, its expression e is evaluated in the original heap, producing a lambda abstraction and an intermediate heap. Then, substituting the lambda variable by the argument of the application, the body of the function is evaluated in the intermediate heap to a final *whnf*, producing a final heap as well.

The $\textsc{Let}_{\Downarrow}$ rule starts by creating a fresh location. Then, the let-bound variable is renamed to this fresh location in all sub-expressions. The location is then allocated to the heap, mapping to the respective augmented expression, and the body of the let is evaluated in this larger heap, with the results being carried over.

Finally, rule $\textsc{Case}_{\Downarrow}$ first evaluates the case discriminant, adding to $\mathcal{S}$ the bound variables of the case alternatives in order to avoid such variables from being used as locations. Assuming this evaluates to a constructor application in an intermediate heap, then, depending on the constructor that results from the evaluation, the selected alternative is evaluated in the intermediate heap, substituting the formal constructor arguments by the concrete ones. The results of evaluating the alternative are then carried over as the results of evaluating the whole case expression. Note that the set $\mathcal{S}$ was introduced by Encina and Peña [EP02] to keep freshness locally checkable, a property that motivated Sestoft's revision [Ses97] to Launchbury's semantics [Lau93].

To illustrate the purpose of set $\mathcal{S}$, consider the following artificial example (in lack of a meaningful short one):

$$\text{case } (\text{let } s = \mathit{Succ}\ s \text{ in } s) \text{ of } \mathit{Succ}\ x \text{ -> } \lambda y.\, x$$

Note that s is defined as a cyclic successor of itself and that the expected result of evaluating the whole expression is a function that discards its single argument and returns the cyclic successor. However, when evaluating let $s = \mathit{Succ}\ s$ in s, had the lambda-bound variable y not been added to set $\mathcal{S}$, we could have chosen y as a fresh location and, although not violating the freshness condition, we would have ended up with the identity function instead as the result, since (with naive substitution) the term $\lambda y.\, x[y/x]$ is equivalent to $\lambda y.\, y$. The set $\mathcal{S}$ avoids such variable captures.

We now present a lemma that states that the contents of heap locations that are under evaluation are preserved during intermediate evaluations.

Lemma 4.3 (Invariant Locations Under Evaluation). *If $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \widehat{e} \Downarrow w, \mathcal{H}'$ then for all $\ell \in \mathcal{L}$ we have $\ell \in \mathcal{H}$* iff *$\ell \in \mathcal{H}'$ and if $\ell \in \mathcal{H}$ then $\mathcal{H}'(\ell) = \mathcal{H}(\ell)$.*

Proof. By inspection of the operational semantics (Figure 4.2) we observe that $\text{VAR}_{\Downarrow}$ is the only rule that modifies an existing location ℓ and that this rule does not apply when $\ell \in \mathcal{L}$. □

4.3 Cost-instrumented Operational Semantics

In order to measure the total number of heap allocations of a given program, we have defined a cost model by instrumenting the rules of Figure 4.2 with a non-negative counter as shown in Figure 4.4.

In the new rules, judgements of the form $\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{m}{\vdash} \widehat{e} \Downarrow w, \mathcal{H}'$ should be read as "in the heap $\mathcal{H}$, expression $\widehat{e}$ evaluates to *whnf* w, producing the new heap $\mathcal{H}'$, and m new heap cells have been allocated".

For simplicity, but without loss of generality, we choose a uniform cost-model where evaluation costs one (heap) unit for each fresh heap location (regardless of its content) that is needed during evaluation — essentially counting the number of new locations in the

$$\frac{w \text{ is in } \textit{whnf}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{0}{\vdash} w \Downarrow w, \mathcal{H}} \qquad (\text{WHNF}_{\Downarrow\text{C}})$$

$$\frac{\ell \notin \mathcal{L} \qquad \mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{\ell\} \overset{m}{\vdash} \mathcal{H}(\ell) \Downarrow w, \mathcal{H}'}{\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{m}{\vdash} \ell \Downarrow w, \mathcal{H}'[\ell \mapsto w]} \qquad (\text{VAR}_{\Downarrow\text{C}})$$

$$\frac{\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{m}{\vdash} e \Downarrow \lambda x.\, e', \mathcal{H}' \qquad \mathcal{H}', \mathcal{S}, \mathcal{L} \overset{m'}{\vdash} e'[\ell/x] \Downarrow w, \mathcal{H}''}{\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{m+m'}{\vdash} e\,\ell \Downarrow w, \mathcal{H}''} \qquad (\text{APP}_{\Downarrow\text{C}})$$

$$\frac{\ell \text{ is fresh} \qquad \mathcal{H}[\ell \mapsto \widehat{e}[\ell/x]], \mathcal{S}, \mathcal{L} \overset{m}{\vdash} e[\ell/x] \Downarrow w, \mathcal{H}'}{\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{1+m}{\vdash} \text{let } x = \widehat{e} \text{ in } e \Downarrow w, \mathcal{H}'} \qquad (\text{LET}_{\Downarrow\text{C}})$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{S} \cup \bigcup_{i=1}^{n} \left(\{\overrightarrow{x_i}\} \cup \mathrm{BV}(e_i)\right), \mathcal{L} \overset{m}{\vdash} e \Downarrow c_k\ \vec{\ell}, \mathcal{H}' \\ \mathcal{H}', \mathcal{S}, \mathcal{L} \overset{m'}{\vdash} e_k[\vec{\ell}/\overrightarrow{x_k}] \Downarrow w, \mathcal{H}''\end{array}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{m+m'}{\vdash} \text{case } e \text{ of } \{c_i\ \overrightarrow{x_i} \texttt{ -> } e_i\}_{i=1}^{n} \Downarrow w, \mathcal{H}''} \qquad (\text{CASE}_{\Downarrow\text{C}})$$

Figure 4.4: Cost-instrumented lazy operational semantics

heap (i.e. the number of newly allocated locations). We could have chosen other metrics [JLH^{+}09], modelling the usage of other countable resources such as execution time or stack space, but we believe this simplicity has allowed us to focus on the principles needed to develop a resource analysis for call-by-need. Cost-metric refinements are left to further work.

The only change introduced in Figure 4.4 with respect to Figure 4.2 is the introduction of the non-negative value above the turnstile. This value corresponds to the cost of evaluation in terms of quantity of heap cells required. We will now describe how the rules in Figure 4.4 affect this total heap allocation counter.

As we have seen, rule WHNF$_{\Downarrow}$ leaves the heap unchanged. Thus, no heap cells are allocated in rule WHNF$_{\Downarrow\text{C}}$, corresponding to a cost of zero.

In rules APP$_{\Downarrow\text{C}}$ and CASE$_{\Downarrow\text{C}}$ the cost of evaluation is the sum of the costs of each of the two evaluation steps.

Rule VAR$_{\Downarrow\text{C}}$ states that the cost of evaluating a location ℓ is the cost of evaluating the corresponding heap expression $\mathcal{H}(\ell)$. Note that although the resulting heap is updated, ℓ was already in the domain of $\mathcal{H}'$ (by Lemma 4.3) and thus no new heap cell was added at that point which justifies the preservation of cost m.

Rule LET$_{\Downarrow\text{C}}$ is the only rule that effectively allocates heap cells, costing one heap cell for the binding created.

4.4 Example: Modelling Call-By-Need

This section illustrates through a couple of simple examples that we are modelling call-by-need rather than call-by-value or call-by-name.

To stress the sequential nature of evaluation we lay out the rules of our cost model vertically: if $\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{m}{\vdash} \widehat{e} \Downarrow w, \mathcal{H}'$ we write

$$\begin{array}{l}
\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \widehat{e} \\
\left\{\begin{array}{l} \\ \text{a sub-rule premise} \\ \\ \end{array}\right. \\
\left\{\begin{array}{l} \\ \text{another sub-rule premise} \\ \\ \end{array}\right. \\
\Downarrow^{m} w, \mathcal{H}'
\end{array}$$

Consider the expression below, which includes a divergent term:

$$\text{let } \mathrm{z} = \mathrm{z} \text{ in } (\lambda \mathrm{x}.\, \lambda \mathrm{y}.\, \mathrm{y})\ \mathrm{z} \tag{4.1}$$

Under a *call-by-value* semantics this would fail to terminate, because z does not admit a normal form. In our *call-by-need* semantics, however, evaluation succeeds:

$$\begin{array}{l}
\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \text{let } \mathrm{z} = \mathrm{z} \text{ in } (\lambda \mathrm{x}.\, \lambda \mathrm{y}.\, \mathrm{y})\ \mathrm{z} \\
\left\{\begin{array}{l}
\mathcal{H}[\ell_3 \mapsto \ell_3], \mathcal{S}, \mathcal{L} \vdash (\lambda \mathrm{x}.\, \lambda \mathrm{y}.\, \mathrm{y})\ \ell_3 \\
\left\{\begin{array}{l}
\mathcal{H}[\ell_3 \mapsto \ell_3], \mathcal{S}, \mathcal{L} \vdash \lambda \mathrm{x}.\, \lambda \mathrm{y}.\, \mathrm{y} \\
\Downarrow^{0} \lambda \mathrm{x}.\, \lambda \mathrm{y}.\, \mathrm{y}, \mathcal{H}[\ell_3 \mapsto \ell_3]
\end{array}\right. \\
\left\{\begin{array}{l}
\mathcal{H}[\ell_3 \mapsto \ell_3], \mathcal{S}, \mathcal{L} \vdash \lambda \mathrm{y}.\, \mathrm{y} \\
\Downarrow^{0} \lambda \mathrm{y}.\, \mathrm{y}, \mathcal{H}[\ell_3 \mapsto \ell_3]
\end{array}\right. \\
\Downarrow^{0} \lambda \mathrm{y}.\, \mathrm{y}, \mathcal{H}[\ell_3 \mapsto \ell_3]
\end{array}\right. \\
\Downarrow^{1} \lambda \mathrm{y}.\, \mathrm{y}, \mathcal{H}[\ell_3 \mapsto \ell_3]
\end{array}$$

The final heap is augmented with a fresh location ℓ_3 whose content is a cyclic self-reference; because the argument $\mathtt{z}$ is discarded by the application, its evaluation is never attempted.

We can see that the semantics is call-by-need rather than call-by-name by observing the sharing of normal forms. Consider,

$$\begin{gathered}\text{let } \mathtt{f} = \text{let } \mathtt{z} = \mathtt{z} \text{ in } (\lambda \mathtt{x}.\, \lambda \mathtt{y}.\, \mathtt{y})\, \mathtt{z} \\ \text{in let } \mathtt{i} = \lambda \mathtt{x}.\, \mathtt{x} \text{ in let } \mathtt{v} = \mathtt{f}\ \mathtt{i} \text{ in } \mathtt{f}\ \mathtt{v}\end{gathered} \tag{4.2}$$

where $\mathtt{f}$ is bound to the thunk (4.1) and applied twice to the identity function. Evaluation of $\mathtt{f}\ \mathtt{v}$ forces the thunk. After the thunk is evaluated, the location ℓ_0 that is associated with $\mathtt{f}$ is updated with the corresponding *whnf* $\lambda \mathtt{y}.\, \mathtt{y}$. The second evaluation of $\mathtt{f}$ does not re-evaluate the thunk (4.1).

Following the rules of our cost model and starting from the empty configuration, Figure 4.5 shows how we derive the following judgement:

$$\emptyset, \emptyset, \emptyset \overset{4}{\vdash} \text{(4.2)} \Downarrow \lambda \mathtt{x}.\, \mathtt{x}, [\ell_0 \mapsto \lambda \mathtt{y}.\, \mathtt{y}, \ell_1 \mapsto \lambda \mathtt{x}.\, \mathtt{x}, \ell_2 \mapsto \lambda \mathtt{x}.\, \mathtt{x}, \ell_3 \mapsto \ell_3]$$

Evaluating expression (4.2) thus costs four heap cells, that is, one cell for each let expression. Under a call-by-name semantics, the cost would instead be 5, since the let expression that is bound to $\mathtt{f}$ would then be evaluated twice, rather than once as here.

The next chapter shows an analysis for lazy evaluation and its validation against the cost model developed in this chapter.

$$
\begin{array}{l}
\emptyset, \emptyset, \emptyset \vdash \text{let } \mathtt{f} = \text{let } \mathtt{z} = \mathtt{z} \text{ in } (\lambda x.\, \lambda y.\, y)\, \mathtt{z} \text{ in let } \mathtt{i} = \lambda x.\, x \text{ in let } \mathtt{v} = \mathtt{f}\,\mathtt{i} \text{ in } \mathtt{f}\,\mathtt{v} \\
\left\{\begin{array}{l}
[\ell_0 \mapsto \text{let } z = z \text{ in } (\lambda x.\, \lambda y.\, y)\, z], \emptyset, \emptyset \vdash \text{let } \mathtt{i} = \lambda x.\, x \text{ in let } \mathtt{v} = \ell_0\, \mathtt{i} \text{ in } \ell_0\, \mathtt{v} \\
\left\{\begin{array}{l}
[\ell_0 \mapsto \text{let } z = z \text{ in } (\lambda x.\, \lambda y.\, y)\, z, \ell_1 \mapsto \lambda x.\, x], \emptyset, \emptyset \vdash \text{let } \mathtt{v} = \ell_0\, \ell_1 \text{ in } \ell_0\, \mathtt{v} \\
\left\{\begin{array}{l}
[\ell_0 \mapsto \text{let } z = z \text{ in } (\lambda x.\, \lambda y.\, y)\, z, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1], \emptyset, \emptyset \vdash \ell_0\, \ell_2 \\
\left\{\begin{array}{l}
[\ell_0 \mapsto \text{let } z = z \text{ in } (\lambda x.\, \lambda y.\, y)\, z, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1], \emptyset, \emptyset \vdash \ell_0 \\
\left\{\begin{array}{l}
[\ell_0 \mapsto \text{let } z = z \text{ in } (\lambda x.\, \lambda y.\, y)\, z, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1], \emptyset, \{\ell_0\} \\
\quad \vdash \text{let } z = z \text{ in } (\lambda x.\, \lambda y.\, y)\, z \\
\left\{\begin{array}{l}
[\ell_0 \mapsto \text{let } z = z \text{ in } (\lambda x.\, \lambda y.\, y)\, z, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3], \emptyset, \{\ell_0\} \\
\quad \vdash (\lambda x.\, \lambda y.\, y)\, \ell_3 \\
\left\{\begin{array}{l}
[\ell_0 \mapsto \text{let } z = z \text{ in } (\lambda x.\, \lambda y.\, y)\, z, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3], \emptyset, \{\ell_0\} \\
\quad \vdash \lambda x.\, \lambda y.\, y \\
\Downarrow^0 \lambda x.\, \lambda y.\, y, \\
\quad [\ell_0 \mapsto \text{let } z = z \text{ in } (\lambda x.\, \lambda y.\, y)\, z, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3]
\end{array}\right. \\
\left\{\begin{array}{l}
[\ell_0 \mapsto \text{let } z = z \text{ in } (\lambda x.\, \lambda y.\, y)\, z, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3], \emptyset, \{\ell_0\} \\
\quad \vdash \lambda y.\, y \\
\Downarrow^0 \lambda y.\, y, \\
\quad [\ell_0 \mapsto \text{let } z = z \text{ in } (\lambda x.\, \lambda y.\, y)\, z, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3]
\end{array}\right. \\
\Downarrow^0 \lambda y.\, y, [\ell_0 \mapsto \text{let } z = z \text{ in } (\lambda x.\, \lambda y.\, y)\, z, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3]
\end{array}\right. \\
\Downarrow^1 \lambda y.\, y, [\ell_0 \mapsto \text{let } z = z \text{ in } (\lambda x.\, \lambda y.\, y)\, z, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3]
\end{array}\right. \\
\Downarrow^1 \lambda y.\, y, [\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3]
\end{array}\right. \\
\left\{\begin{array}{l}
[\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3], \emptyset, \emptyset \vdash \ell_2 \\
\left\{\begin{array}{l}
[\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3], \emptyset, \{\ell_2\} \vdash \ell_0\, \ell_1 \\
\left\{\begin{array}{l}
[\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3], \emptyset, \{\ell_2\} \vdash \ell_0 \\
\left\{\begin{array}{l}
[\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3], \emptyset, \{\ell_2, \ell_0\} \vdash \lambda y.\, y \\
\Downarrow^0 \lambda y.\, y, [\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3]
\end{array}\right. \\
\Downarrow^0 \lambda y.\, y, [\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3]
\end{array}\right. \\
\left\{\begin{array}{l}
[\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3], \emptyset, \{\ell_2\} \vdash \ell_1 \\
\left\{\begin{array}{l}
[\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3], \emptyset, \{\ell_2, \ell_1\} \vdash \lambda x.\, x \\
\Downarrow^0 \lambda x.\, x, [\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3]
\end{array}\right. \\
\Downarrow^0 \lambda x.\, x, [\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3]
\end{array}\right. \\
\Downarrow^0 \lambda x.\, x, [\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \ell_0\, \ell_1, \ell_3 \mapsto \ell_3]
\end{array}\right. \\
\Downarrow^0 \lambda x.\, x, [\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \lambda x.\, x, \ell_3 \mapsto \ell_3]
\end{array}\right. \\
\Downarrow^1 \lambda x.\, x, [\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \lambda x.\, x, \ell_3 \mapsto \ell_3]
\end{array}\right. \\
\Downarrow^2 \lambda x.\, x, [\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \lambda x.\, x, \ell_3 \mapsto \ell_3]
\end{array}\right. \\
\Downarrow^3 \lambda x.\, x, [\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \lambda x.\, x, \ell_3 \mapsto \ell_3]
\end{array}\right. \\
\Downarrow^4 \lambda x.\, x, [\ell_0 \mapsto \lambda y.\, y, \ell_1 \mapsto \lambda x.\, x, \ell_2 \mapsto \lambda x.\, x, \ell_3 \mapsto \ell_3]
\end{array}
$$

Figure 4.5: Evaluation under a call-by-need semantics

5. Amortised Analysis

In this chapter we present a type-based amortised analysis of total heap allocation for higher-order lazy functional programs.

Our approach is based on the principle of amortisation as described in Chapter 3. We start by defining the types and presenting the rules of our type system. We then define some auxiliary mappings that help us construct the invariants to the main contribution of this thesis: a soundness proof connecting our analysis to the cost model of Chapter 4, proving that the upper bounds given by our analysis are not exceeded under the cost model. Finally, at the end of the chapter we present a system for eager evaluation, derived with minimal changes from the lazy, in order to, by contrast, emphasise the key elements needed in the development of our analysis for lazy evaluation.

5.1 Types and Typing Contexts

As described in Chapter 3, to develop a type-based amortised analysis, we start by defining the annotated types the type system (to be presented later in this chapter) will handle.

The syntax of allowed types is shown in Figure 5.1. We use meta-variables A, B, C for types, X, Y for type variables and p, q for annotations, i.e. non-negative rational numbers representing potential or cost (whenever possible we use p for potential and q for cost annotations). The allowed types include type variables, function types and possibly recursive data types over labelled sums of products (representing the types of each constructor) and thunk types.

Except for type variables, all types have annotations. Annotation q in a function type expresses the cost incurred by each evaluation of the corresponding function; similarly, the annotation q in a thunk type captures the cost of evaluating the corresponding thunk

$$\begin{array}{llll} A, B, C & ::= & X & \text{– type variable} \\ & \mid & A \xrightarrow{q} B & \text{– function type} \\ & \mid & \mu X.\{c_1 : (p_1, \vec{B}_1) \mid \cdots \mid c_n : (p_n, \vec{B}_n)\} & \text{– data type} \\ & \mid & \mathsf{T}^q(A) & \text{– thunk type} \end{array}$$

with $q, p_1, \ldots, p_n \in \mathbb{Q}_0^+$

Figure 5.1: Annotated types

(this cost can be zero if the thunk is known to be in *whnf*). As in previous type-based amortised analyses, our potential function (from data structures to numbers) is defined in a type-directed way: in particular, we choose to annotate data types with non-negative coefficients p_i that specify the contribution of each constructor c_i to the potential of the data structure. Although this representation of potential in data types limits the analysis to express bounds that are linear functions to the number of constructors in data structures, we keep our focus on the contributions found in this thesis, showing a range of interesting examples successfully handled by the analysis (Chapter 6), and leave as further work the study of an extension to super-linear bounds, along the lines of recent work [HAH11].

Recall from Chapter 4 that arguments to constructor applications are always variables (locations) and never values, and also that constructor applications are always introduced by *lets*. As such, constructor applications and their arguments are always stored in the heap and are referenced by a location, i.e. constructors in our cost model are *boxed*, and this is reflected in data types, where each type in $\vec{B}_i$, with $i = 1, \ldots, n$, is implicitly a thunk type. Similarly, since arguments to applications are always variables, the argument of function types is implicitly a thunk type as well.

We consider only recursive data types that are *non-interleaving* [Mat98], i.e. we exclude μ-types whose bound variables overlap in scope, e.g. $\mu X.\{c_1 : (\ldots, \mu Y.\{c_2 : (\ldots, \mathsf{T}(X))\})\}$. This helps us prove a crucial lemma on cyclic structures (Lemma 5.17) in the key soundness proof (Theorem 5.13). Note that this restriction does not prohibit nested data types; e.g. the type of lists of lists of naturals is $\mu Y.\{\,\texttt{Nil} : (p'_n, ()) \mid \texttt{Cons} : (p'_c, (\mathsf{T}(\texttt{LN}), \mathsf{T}(Y)))\,\}$, where $\texttt{LN} = \mu Y.\{\,\texttt{Nil} : (p_n, ()) \mid \texttt{Cons} : (p_c, (\mathsf{T}(\texttt{N}), \mathsf{T}(Y)))\,\}$ is the type of list of naturals and $\texttt{N} = \mu X.\{\,\texttt{Zero} : (p_z, ()) \mid \texttt{Succ} : (p_s, \mathsf{T}(X))\,\}$ is the type of naturals, and we have omitted thunk type annotations for simplicity. Also, note that distinct lists can be assigned different constructor annotations in their types, thus improving the precision of the cost analysis.

For simplicity, we also exclude *resource parametricity* [JLHH10], since this is only important for functions that are re-used in different circumstances and not for thunks that are evaluated at most once, being thus orthogonal to this thesis. Nevertheless, adopting resource parametricity in our analysis is left as further work.

Typing contexts are multisets of pairs $x{:}A$ of variables and annotated types; we use multisets to allow separate potential to be accounted for in multiple references. We use Γ, Δ, Θ for contexts and $\Gamma\restriction_x$ for the multiset of types associated with x in Γ, i.e. $\Gamma\restriction_x = \{A \mid x{:}A \in \Gamma\}$. Note that, since variables in typing contexts represent locations in a heap, their corresponding types are implicitly thunk types, similar to the above observation on arguments to function types and data types.

5.2 Sharing Relation

Figure 5.2 shows the syntactical rules for an auxiliary judgement $\curlyvee(A \,|\, B_1, \ldots, B_n\,)$ that is used to *share* a type A among a finite multiset of types $\{B_1, \ldots, B_n\}$. It is used to limit contraction in our type-system. Rule SHAREEMPTY accepts sharing a type among an empty multiset of types. Data type annotations for potential associated with A are linearly distributed among B_1, ..., B_n (SHAREDAT), whereas cost annotations for functions and thunks are preserved (SHAREFUN and SHARETHUNK). Rule SHAREVEC is applied to vectors of types with the same length and is used as a premise of SHAREDAT.

Figure 5.3 extends the sharing relation for typing contexts. With rule SHARECTX a typing context $\{x_1{:}A_1, \ldots, x_n{:}A_n\}$ shares to another typing context Δ *iff* there is a partition $\Delta_1, \ldots, \Delta_n$ of Δ such that $\curlyvee(x_i{:}A_i \,|\, \Delta_i\,)$ holds and $\mathrm{dom}(\Delta_i) \subseteq \{x_i\}$, for $(1 \leq i \leq n)$.

For example,

$$\curlyvee\big(x{:}\mathsf{T}^2(\mu X.\{\texttt{Unit}{:}(5,())\}) \,\big|\, x{:}\mathsf{T}^3(\mu X.\{\texttt{Unit}{:}(3,())\}),\ x{:}\mathsf{T}^5(\mu X.\{\texttt{Unit}{:}(2,())\})\,\big)$$
$$\curlyvee\Big(f{:}\mathsf{T}^2(\mathsf{T}^5(A) \xrightarrow{2} B) \,\Big|\, f{:}\mathsf{T}^3(\mathsf{T}^3(A) \xrightarrow{3} B),\ f{:}\mathsf{T}^5(\mathsf{T}^2(A) \xrightarrow{5} B)\Big)$$

hold, whereas

$$\curlyvee\big(x{:}\mathsf{T}^2(\mu X.\{\texttt{Unit}{:}(5,())\}) \,\big|\, x{:}\mathsf{T}^3(\mu X.\{\texttt{Unit}{:}(3,())\}),\ y{:}\mathsf{T}^5(\mu X.\{\texttt{Unit}{:}(2,())\})\,\big)$$
$$\curlyvee\big(x{:}\mathsf{T}^2(\mu X.\{\texttt{Unit}{:}(5,())\}) \,\big|\, x{:}\mathsf{T}^3(\mu X.\{\texttt{Unit}{:}(3,())\}),\ x{:}\mathsf{T}^5(\mu X.\{\texttt{Unit}{:}(3,())\})\,\big)$$
$$\curlyvee\Big(f{:}\mathsf{T}^2(\mathsf{T}^2(A) \xrightarrow{2} B) \,\Big|\, f{:}\mathsf{T}^3(\mathsf{T}^3(A) \xrightarrow{3} B)\Big)$$

$$\frac{}{\curlyvee(A \mid \emptyset)} \tag{\textsc{ShareEmpty}}$$

$$\frac{}{\curlyvee(X \mid X, \ldots, X)} \tag{\textsc{ShareVar}}$$

$$\frac{\begin{array}{c} B_i = \mu X.\{c_1 : (p'_{i1},\, \vec{B}_{i1}) \mid \cdots \mid c_m : (p'_{im},\, \vec{B}_{im})\} \\ \curlyvee\left(\vec{A}_j \,\middle|\, \vec{B}_{1j},\, \ldots,\, \vec{B}_{nj}\right) \qquad p_j \geq \sum_{i=1}^{n} p'_{ij} \qquad (1 \leq i \leq n,\ 1 \leq j \leq m) \end{array}}{\curlyvee\left(\mu X.\{c_1 : (p_1,\, \vec{A}_1) \mid \cdots \mid c_m : (p_m,\, \vec{A}_m)\} \;\middle|\; B_1,\, \ldots,\, B_n\right)} \tag{\textsc{ShareDat}}$$

$$\frac{\curlyvee(A_i \mid A) \qquad \curlyvee(B \mid B_i) \qquad q_i \geq q \qquad (1 \leq i \leq n)}{\curlyvee(A \xrightarrow{q} B \mid A_1 \xrightarrow{q_1} B_1,\, \ldots,\, A_n \xrightarrow{q_n} B_n)} \tag{\textsc{ShareFun}}$$

$$\frac{\curlyvee(A_j \mid B_{1j},\, \ldots,\, B_{nj}) \qquad m = \left|\vec{A}\right| = \left|\vec{B}_i\right| \qquad (1 \leq i \leq n,\ 1 \leq j \leq m)}{\curlyvee\left(\vec{A} \,\middle|\, \vec{B}_1,\, \ldots,\, \vec{B}_n\right)} \tag{\textsc{ShareVec}}$$

$$\frac{\curlyvee(A \mid A_1, \ldots, A_n) \qquad q_i \geq q \qquad (1 \leq i \leq n)}{\curlyvee(\mathsf{T}^{q}(A) \mid \mathsf{T}^{q_1}(A_1), \ldots, \mathsf{T}^{q_n}(A_n))} \tag{\textsc{ShareThunk}}$$

Figure 5.2: Sharing relation

$$\frac{}{\curlyvee(\Gamma \mid \emptyset)} \tag{\textsc{ShareEmptyCtx}}$$

$$\frac{\curlyvee(A \mid B_1, \ldots, B_n) \qquad \curlyvee(\Gamma \mid \Delta)}{\curlyvee(x : A,\, \Gamma \mid x : B_1, \ldots, x : B_n, \Delta)} \tag{\textsc{ShareCtx}}$$

Figure 5.3: Sharing relation extended to contexts

do not. The last three sharing examples fail since in the first of these a typing for y appears only at the right-hand side of the sharing relation; in the second, the potential on the left-hand side is not linearly distributed with respect to the right-hand side ($5 \not\leq 3 + 3$); and the last example fails since sharing is contravariant in the left argument of functions and thus, while the cost of the outermost thunk type on the right-hand side can exceed the corresponding cost on the left-hand side, the cost of the inner thunk type cannot.

5.2.1 Subtyping Relation

Sharing also allows the relaxation of annotations to subsume subtyping. The special case of sharing one type to a single other corresponds to a *subtyping relation*; we define the shorthand notation $A <: B$ to mean $\curlyvee(A \mid B)$. Inequalities over type annotations in rules ShareDat, ShareFun and ShareThunk allow potential annotations to decrease and cost annotations to increase. Informally, $A <: B$ implies not only that A and B have identical underlying types, but also that B has lower or equal potential and greater or equal cost than that of A. As usual in structural subtyping, this relation is contravariant in the left argument of functions (SHAREFUN).

5.2.2 Idempotent Types

We now define the notion that some types can be freely shared. Namely, if they observe the following definition:

Definition 5.1 (Idempotent Types and Idempotent Contexts)**.** We say type A (respectively context Γ) is *idempotent* iff $\curlyvee(A \mid A, A)$ (respectively $\curlyvee(\Gamma \mid \Gamma, \Gamma)$) holds.

This special case occurs when sharing a type or context to itself: because of non-negativity, $\curlyvee(A \mid A, A)$ (respectively $\curlyvee(\Gamma \mid \Gamma, \Gamma)$) requires the potential annotations in A (respectively Γ) to be zero for all data types outside of function types.

Note though that function types are unaffected by this special case of sharing. However, since function types do not carry potential per se (the potential required to execute the body of a function must come from its arguments), all types subject to such constraint carry no potential.

For example, types

$$\mathsf{T}^1(\mu X.\{\texttt{Unit}{:}(0,())\})$$
$$\mathsf{T}^1(\mathsf{T}^1(\mu X.\{\texttt{Unit}{:}(1,())\}) \xrightarrow{1} B)$$
$$\mathsf{T}^1(\mu X.\{\texttt{Cons}{:}(0,(\mathsf{T}^1(\mu X.\{\texttt{Unit}{:}(0,())\}), \mathsf{T}^1(X))) \mid \texttt{Nil}{:}(0,())\})$$

are idempotent, whereas types

$$\mathsf{T}^1(\mu X.\{\texttt{Unit}{:}(1,())\})$$
$$\mathsf{T}^1(\mu X.\{\texttt{Cons}{:}(1,(\mathsf{T}^1(\mu X.\{\texttt{Unit}{:}(0,())\}), \mathsf{T}^1(X))) \mid \texttt{Nil}{:}(0,())\})$$
$$\mathsf{T}^1(\mu X.\{\texttt{Cons}{:}(0,(\mathsf{T}^1(\mu X.\{\texttt{Unit}{:}(0,())\}), \mathsf{T}^1(X))) \mid \texttt{Nil}{:}(1,())\})$$

are not.

We use this property to impose a constraint that types or contexts carry no potential. A variant of this is $\curlyvee(A \mid A, A')$, which implies that A' is a subtype of A that holds no potential.

5.3 Typing Judgements

Our analysis for lazy evaluation is presented in Figures 5.4 and 5.5 as a proof system that derives judgements of the form $\Gamma \overset{q}{\vdash} \widehat{e} : A$, where Γ is a typing context, $\widehat{e}$ is an augmented expression, A is an annotated type and q (above the turnstile) is a non-negative rational number approximating the cost of evaluating $\widehat{e}$. For simplicity, we will omit turnstile annotations whenever they are not explicitly mentioned.

In the LET rule, the cost q' of evaluating $\widehat{e}$ is deferred by moving it to the thunk type of x in the type judgement of e. If x does not occur in e then its cost can be discarded, in accordance with lazy evaluation. Also, type A' is restricted to being idempotent in order to prevent the potential of x from being reused in the derivation of $\widehat{e}$, keeping potential from being obtained for free in the recursive definition. Finally, the overall cost of the let expression is 1 for the newly allocated heap cell (according to the cost model) plus the cost q of evaluating the body e and, if $\widehat{e}$ is a constructor, its potential p' is also added to the overall cost. Note that the thunk cost of x in the type judgement of $\widehat{e}$ is q', instead of always zero as in a previous presentation [SVF^{+}12]. This change allowed us to fix a minor problem in the soundness proof of the main theorem.

VAR moves the cost from the thunk type to the turnstile, ensuring that any cost in the thunk type is paid for at this point of access in a type derivation.

In the ABS rule, the cost of eventually applying the λ-abstraction is q, but the cost of evaluating the λ-abstraction itself is zero, since it is already a *whnf*. In order to avoid duplicating potential where a λ-abstraction is applied more than once, ABS ensures that Γ is idempotent, by forcing it to share with itself. While on the one hand this means functions can be reused arbitrarily without risking unsound duplication of potential, on the other hand functions must obtain all their required potential, other than a constant amount, from their input argument x alone and not from other variables in $\mathrm{dom}(\Gamma)$.

APP ensures that the argument and function types match and includes the cost of the function in the final result.

The CONS rule simply ensures consistency between the arguments and the result type. Since constructors cannot appear in source forms, the rule is used only when we need to assign types either to heap expressions or to evaluation results. Note that while rule LET

$$\frac{\begin{array}{c}\Gamma, x{:}\mathsf{T}^{q'}(A') \overset{q'}{\vdash} \widehat{e} : A \qquad \Delta, x{:}\mathsf{T}^{q'}(A) \overset{q}{\vdash} e : C \\ x \notin \mathrm{dom}(\Gamma, \Delta) \qquad \curlyvee(A \,|\, A, A') \qquad q' = 0 \text{ if } \widehat{e} \text{ is a } \textit{whnf} \\ p = \begin{cases} p', & \text{if } \widehat{e} \equiv c\,\vec{y} \text{ and } A = \mu X.\{\cdots \mid c : (p', \vec{B}) \mid \cdots\} \\ 0, & \text{otherwise} \end{cases}\end{array}}{\Gamma, \Delta \overset{1+q+p}{\vdash} \text{let } x = \widehat{e} \text{ in } e : C} \tag{LET}$$

$$\frac{}{x{:}\mathsf{T}^{q}(A) \overset{q}{\vdash} x : A} \tag{VAR}$$

$$\frac{\Gamma, x{:}A \overset{q}{\vdash} e : C \qquad x \notin \mathrm{dom}(\Gamma) \qquad \curlyvee(\Gamma \,|\, \Gamma, \Gamma)}{\Gamma \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C} \tag{ABS}$$

$$\frac{\Gamma \overset{q}{\vdash} e : A \xrightarrow{q'} C}{\Gamma, y{:}A \overset{q+q'}{\vdash} e\,y : C} \tag{APP}$$

$$\frac{B = \mu X.\{\cdots \mid c : (p, \vec{A}) \mid \cdots\}}{y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X] \overset{0}{\vdash} c\,\vec{y} : B} \tag{CONS}$$

$$\frac{\begin{array}{c}\Gamma \overset{q}{\vdash} e : B \qquad B = \mu X.\{c_1 : (p_1, \overrightarrow{A_1}) \mid \cdots \mid c_n : (p_n, \overrightarrow{A_n})\} \\ (\bigcup_{i=1}^{n}\{\overrightarrow{x_i}\}) \cap \mathrm{dom}(\Delta) = \emptyset \\ i = 1, \ldots, n \begin{cases} |\overrightarrow{A_i}| = |\overrightarrow{x_i}| = k_i \\ \Delta, x_{i_1}{:}A_{i_1}[B/X], \ldots, x_{i_{k_i}}{:}A_{i_{k_i}}[B/X] \overset{q'+p_i}{\vdash} e_i : C \end{cases}\end{array}}{\Gamma, \Delta \overset{q+q'}{\vdash} \text{case } e \text{ of } \{c_i\ \overrightarrow{x_i} \text{ -> } e_i\}_{i=1}^{n} : C} \tag{CASE}$$

Figure 5.4: Syntax directed type rules

must ensure that sufficient potential (p') is available for the constructor, the CONS rule does not — the former corresponds to allocating a constructor, the latter to merely referencing one.

The CASE rule deals with pattern-matching over an expression of a (possibly recursive) data type. The rule requires that all branches of the alternatives admit an identical result type and that part of the estimated cost of each alternative branch is the same; fulfilling such a condition may require the relaxation of type and/or cost information using the structural rules described below. The matching branch uses extra resources corresponding to the potential annotation on the matched constructor, previously set aside at the introduction of the constructor (LET).

The structural rules of Figure 5.5 allow the analysis to be relaxed in various ways. Rule WEAK allows the introduction of an extra hypothesis in the typing context and the side condition ensures type A must be structurally equivalent to any of $\Gamma\restriction_x$, if $\Gamma\restriction_x$ is not empty, preventing ill-formed contexts, such as $\{x{:}\texttt{Bool}, x{:}\texttt{List}\}$. RELAX allows argument costs to be relaxed. SUPERTYPE and SUBTYPE allow supertyping in a hypothesis and subtyping

$$\frac{\Gamma \overset{q}{\vdash} e : C \qquad \curlyvee(A' \mid (\Gamma, x{:}A)\upharpoonright_x)}{\Gamma, x{:}A \overset{q}{\vdash} e : C} \qquad (\textsc{Weak})$$

$$\frac{\Gamma \overset{q'}{\vdash} e : A \qquad q \geq q'}{\Gamma \overset{q}{\vdash} e : A} \qquad (\textsc{Relax})$$

$$\frac{\Gamma, x{:}B \overset{q}{\vdash} e : C \qquad A <: B}{\Gamma, x{:}A \overset{q}{\vdash} e : C} \qquad (\textsc{Supertype})$$

$$\frac{\Gamma \overset{q}{\vdash} e : B \qquad B <: C}{\Gamma \overset{q}{\vdash} e : C} \qquad (\textsc{Subtype})$$

$$\frac{\Gamma, x{:}A_1, x{:}A_2 \overset{q}{\vdash} e : C \qquad \curlyvee(A \mid A_1, A_2)}{\Gamma, x{:}A \overset{q}{\vdash} e : C} \qquad (\textsc{Share})$$

$$\frac{\Gamma, x{:}\mathsf{T}^{q'_0}(A) \ \overset{q}{\vdash} e : C}{\Gamma, x{:}\mathsf{T}^{q'_0+q'}(A) \ \overset{q+q'}{\vdash} e : C} \qquad (\textsc{Prepay})$$

Figure 5.5: Structural type rules

in the conclusion, respectively. SHARE allows the use of sharing to split potential in a hypothesis. Finally, PREPAY allows (part or all of) the cost of a thunk to be paid for, so reducing the cost of further uses.

It is important to note that a decrease of cost annotations for thunks (possibly down to zero) can only be achieved through the PREPAY structural rule and not through the sharing rules of Figure 5.2. Without PREPAY the system would model call-by-name, since each access of a variable would pay for the entire cost. Also, if we would force the use of PREPAY for the entire cost after each LET, we would be modelling call-by-value: pay in full once at introduction (LET) and pay zero at every access (VAR). It is the ability to selectively choose when to use PREPAY that enables the system to model call-by-need. Thus, “prepaying” is key to correctly modelling the reduced costs of lazy evaluation by allowing costs to be accounted only once for a thunk, if at all.

5.4 Example: Analysing Call-By-Need

We now present type derivations for the examples from Section 4.4 in order to illustrate how the type rules of Figures 5.4 and 5.5 reflect the costs of our operational semantics.

5.4.1 Non-Strict Evaluation

Recall example (4.1) which demonstrates that unneeded redexes are not reduced (i.e. that the semantics is non-strict):

$$\text{let } \mathtt{z} = \mathtt{z} \text{ in } (\lambda \mathtt{x}.\, \lambda \mathtt{y}.\, \mathtt{y})\, \mathtt{z}$$

Evaluation of this term in our operational semantics succeeds, requires one heap cell (for allocating the thunk named by $\mathtt{z}$) and the result is the identity function $\lambda \mathtt{y}.\, \mathtt{y}$:

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{1}{\vdash} \text{let } \mathtt{z} = \mathtt{z} \text{ in } (\lambda \mathtt{x}.\, \lambda \mathtt{y}.\, \mathtt{y})\, \mathtt{z} \Downarrow \lambda \mathtt{y}.\mathtt{y}, \mathcal{H}'$$

An analysis for this term is given in Figure 5.6 as an annotated type derivation.[*]

The final judgement is:

$$\emptyset \overset{1}{\vdash} \text{let } \mathtt{z} = \mathtt{z} \text{ in } (\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y})\, \mathtt{z} : \mathsf{T}^{q}(B) \xrightarrow{q} B$$

The annotation in the turnstile of this judgement gives a cost estimate of one heap cell, matching the exact cost of the operational semantics. The type annotation q represents the cost of the thunk bound to the concrete argument of the identity function $\lambda \mathtt{y}.\, \mathtt{y}$. The value of q can be arbitrary. So can type B. Note that type A' is similarly arbitrary, subject only to the side condition $\curlyveedownarrow(A' \mid A', A')$, forbidding circular data for having potential.

5.4.2 Lazy Evaluation

The second example (4.2) illustrates the sharing of normal forms, i.e. lazy evaluation:

$$\begin{aligned} &\text{let } \mathtt{f} = \text{let } \mathtt{z} = \mathtt{z} \text{ in } (\lambda \mathtt{x}.\, \lambda \mathtt{y}.\, \mathtt{y})\, \mathtt{z} \\ &\quad \text{in let } \mathtt{i} = \lambda \mathtt{x}.\, \mathtt{x} \text{ in let } \mathtt{v} = \mathtt{f}\ \mathtt{i} \text{ in } \mathtt{f}\ \mathtt{v} \end{aligned}$$

Evaluating `f v` forces the thunk named by `f`; following evaluation, the location associated with `f` is updated with a *whnf*. Subsequent evaluations of `f` re-use this result. Evaluation of

[*] For the complete derivation see Figure B.1 in Appendix B.

$$\dfrac{\dfrac{}{\mathtt{z}{:}\mathsf{T}^{q''}(A') \overset{q''}{\vdash} \mathtt{z} : A'}\,\textsc{Var} \qquad \dfrac{\cdots}{\mathtt{z}{:}\mathsf{T}^{q''}(A') \overset{0}{\vdash} (\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y})\, \mathtt{z} : \mathsf{T}^{q}(B) \xrightarrow{q} B}}{\emptyset \overset{1}{\vdash} \text{let } \mathtt{z} = \mathtt{z} \text{ in } (\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y})\, \mathtt{z} : \mathsf{T}^{q}(B) \xrightarrow{q} B}\,\textsc{Let}$$

$$\text{where } \curlyvee(A' \mid A', A')$$

Figure 5.6: Type derivation for a non-strict evaluation example

the overall expression therefore costs 4 heap cells (as seen in Figure 4.5, Chapter 4):

$$\emptyset, \emptyset, \emptyset \overset{4}{\vdash} \text{(4.2)} \Downarrow \lambda \mathtt{x}.\, \mathtt{x}, [\ell_0 \mapsto \lambda \mathtt{y}.\, \mathtt{y}, \ell_1 \mapsto \lambda \mathtt{x}.\, \mathtt{x}, \ell_2 \mapsto \lambda \mathtt{x}.\, \mathtt{x}, \ell_3 \mapsto \ell_3]$$

The type derivation in Figure 5.7 shows the analysis for this example.[†]

The final type judgement replicates the exact operational cost of 4 heap cells:

$$\emptyset \overset{4}{\vdash} \text{(4.2)} : B, \text{ where } B = \mathsf{T}^{q'}(C) \xrightarrow{q'} C$$

Note that we employ the structural type rule SHARE to allow the function `f` to be used twice. The duplication is justified since the type of `f` is idempotent (i.e. it shares to itself).

The crucial point in this type derivation that allows us to match the exact operational cost is the use of the structural rule PREPAY (below SHARE) to pay, precisely once, the cost of the thunk bound to `f`.

Also note that although the type derivation constrains $B = \mathsf{T}^{q'}(C) \xrightarrow{q'} C$ to be idempotent, i.e. $\curlyvee(B \mid B, B)$, it leaves type C unconstrained.

5.5 Soundness

This section establishes the soundness of our analysis for lazy evaluation with respect to the cost model of Section 4.3.

We begin by stating some auxiliary proof lemmas and preliminary definitions, notably formalising the notion of potential. We then define the principal invariants of our system, namely, type consistency and type compatibility relations between a heap configuration of

[†] For the complete derivation see Figure B.2 in Appendix B.

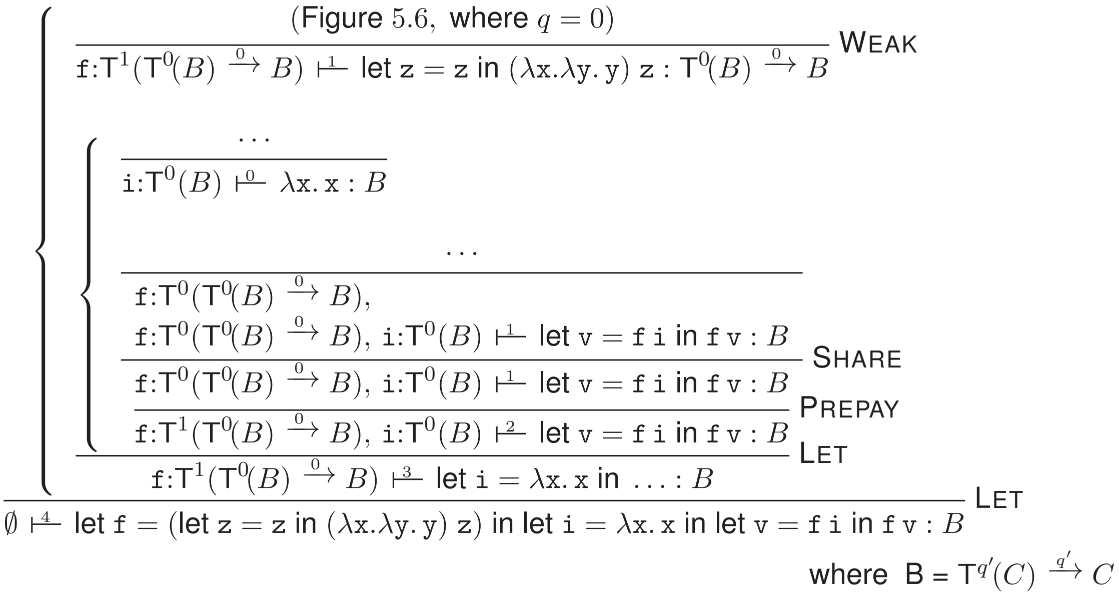
(Figure 5.6, where $q = 0$)
WEAK
$\mathtt{f}{:}\mathsf{T}^1(\mathsf{T}^0(B) \xrightarrow{0} B) \overset{1}{\vdash} \text{let } \mathtt{z} = \mathtt{z} \text{ in } (\lambda \mathtt{x}.\lambda \mathtt{y}.\,\mathtt{y})\ \mathtt{z} : \mathsf{T}^0(B) \xrightarrow{0} B$
$\mathtt{i}{:}\mathsf{T}^0(B) \overset{0}{\vdash} \lambda \mathtt{x}.\,\mathtt{x} : B$
$\mathtt{f}{:}\mathsf{T}^0(\mathsf{T}^0(B) \xrightarrow{0} B),$
$\mathtt{f}{:}\mathsf{T}^0(\mathsf{T}^0(B) \xrightarrow{0} B), \mathtt{i}{:}\mathsf{T}^0(B) \overset{1}{\vdash} \text{let } \mathtt{v} = \mathtt{f}\ \mathtt{i} \text{ in } \mathtt{f}\ \mathtt{v} : B$
SHARE
$\mathtt{f}{:}\mathsf{T}^0(\mathsf{T}^0(B) \xrightarrow{0} B), \mathtt{i}{:}\mathsf{T}^0(B) \overset{1}{\vdash} \text{let } \mathtt{v} = \mathtt{f}\ \mathtt{i} \text{ in } \mathtt{f}\ \mathtt{v} : B$
PREPAY
$\mathtt{f}{:}\mathsf{T}^1(\mathsf{T}^0(B) \xrightarrow{0} B), \mathtt{i}{:}\mathsf{T}^0(B) \overset{2}{\vdash} \text{let } \mathtt{v} = \mathtt{f}\ \mathtt{i} \text{ in } \mathtt{f}\ \mathtt{v} : B$
LET
$\mathtt{f}{:}\mathsf{T}^1(\mathsf{T}^0(B) \xrightarrow{0} B) \overset{3}{\vdash} \text{let } \mathtt{i} = \lambda \mathtt{x}.\,\mathtt{x} \text{ in } \ldots : B$
LET
$\emptyset \overset{4}{\vdash} \text{let } \mathtt{f} = (\text{let } \mathtt{z} = \mathtt{z} \text{ in } (\lambda \mathtt{x}.\lambda \mathtt{y}.\,\mathtt{y})\ \mathtt{z}) \text{ in let } \mathtt{i} = \lambda \mathtt{x}.\,\mathtt{x} \text{ in let } \mathtt{v} = \mathtt{f}\ \mathtt{i} \text{ in } \mathtt{f}\ \mathtt{v} : B$
where B = $\mathsf{T}^{q'}(C) \xrightarrow{q'} C$

Figure 5.7: Type derivation for a lazy-evaluation example

the operational semantics and global types, contexts and balance. We conclude with the soundness result proper (Theorem 5.13).

5.5.1 Auxiliary Lemmas

The first auxiliary lemma allows us to replace variables in type derivations. Note that because of the lazy evaluation semantics (and unlike the usual substitution lemma for the λ-calculus), we substitute only with variables but not with arbitrary expressions. Also, since our typing contexts are multisets, we need to ensure the simultaneous substitution of all typings of the variable in the context.

Lemma 5.2 (Substitution)**.** *If* $\Gamma, x{:}A_1, \ldots, x{:}A_n \overset{q}{\vdash} \widehat{e} : C$ *and* $x \notin \mathrm{dom}(\Gamma)$ *and* $y \notin \mathrm{dom}(\Gamma) \cup \mathrm{FV}(\widehat{e})$ *then also* $\Gamma, y{:}A_1, \ldots, y{:}A_n \overset{q}{\vdash} \widehat{e}[y/x] : C$*.*

Proof. By induction on the height of derivation of $\Gamma, x{:}A_1, \ldots, x{:}A_n \overset{q}{\vdash} \widehat{e} : C$, simply replacing any occurrences of x for y. □

The next two lemmas establish inversion properties for constructors and λ-abstractions.

Lemma 5.3 (CONS Inversion)**.** *If* $\Gamma \overset{0}{\vdash} c\ \vec{y} : B$ *then* $B = \mu X.\{\cdots \mid c : (p, \vec{A}) \mid \cdots\}$ *and* $\curlyvee(\Gamma \mid y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X])$*.*

Lemma 5.4 (ABS Inversion). *If $\Gamma \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C$ then there exists Γ' such that $\curlyvee(\Gamma \,|\, \Gamma')$, $\curlyvee(\Gamma' \,|\, \Gamma', \Gamma')$, $x \notin \mathrm{dom}(\Gamma')$ and $\Gamma', x{:}A \overset{q}{\vdash} e : C$.*

Proof Sketch (for both lemmas). A typing with conclusion $\Gamma \overset{0}{\vdash} c\,\vec{y} : B$ must result from axiom CONS followed by (possibly zero) uses of structural rules. Similarly, a typing $\Gamma \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C$ must result from an application of the rule ABS followed by (possibly zero) uses of structural rules. The proof follows by induction on the structural rules, considering each rule separately. For rules RELAX and PREPAY induction is trivial since both type judgements have zero on the turnstile. For the remaining structural rules the proof follows by transitivity of the sharing relation. See Section 5.5.6.2 and 5.5.6.3, respectively, for the detailed proofs. □

Note that, for a typing judgement with any number greater than zero on the turnstile, inversion in our type system would not hold in general. The reason is that two (mutually exclusive) rules might apply. For example $x{:}\mathsf{T}^1(A) \overset{1}{\vdash} e : C$ might have premise $x{:}\mathsf{T}^1(A) \overset{0}{\vdash} e : C$ through rule RELAX, or it might have premise $x{:}\mathsf{T}^0(A) \overset{0}{\vdash} e : C$ throught rule PREPAY. This is not a problem since the proofs we present in our system do not require inversion lemmas with a number other than zero on the turnstile of the typing judgements.

The final auxiliary lemma allows splitting contexts used for typing expressions in *whnf* according to a split of the result type.

Lemma 5.5 (Context Splitting). *If $\Gamma \overset{0}{\vdash} w : A$, where w is an expression in* whnf *and $\curlyvee(A \,|\, A_1, A_2)$; then there exist Γ_1, Γ_2 such that $\curlyvee(\Gamma \,|\, \Gamma_1, \Gamma_2)$, $\Gamma_1 \overset{0}{\vdash} w : A_1$ and $\Gamma_2 \overset{0}{\vdash} w : A_2$.*

Proof Sketch. The proof follows from an application of Lemma 5.3 (if w is a constructor) or Lemma 5.4 (if w is an abstraction) together with the definition of sharing. See Section 5.5.6.4 for the detailed proof. □

5.5.2 Global Types, Contexts and Balance

We now define some auxiliary mappings that will be necessary for formulating the soundness of our type system.

The mapping $\mathcal{M}$ from locations to types, written $\{\ell_1 \mapsto A_1, \ldots, \ell_n \mapsto A_n\}$, records the *global type* of a location, which accounts for all potential in all references to that location.

We extend subtyping to global types in the natural way, namely $\mathcal{M} <: \mathcal{M}'$ if and only if $\mathrm{dom}(\mathcal{M}) \subseteq \mathrm{dom}(\mathcal{M}')$ and for all $\ell \in \mathrm{dom}(\mathcal{M})$ we have $\mathcal{M}(\ell) <: \mathcal{M}'(\ell)$. This relation will be used to assert that the potential assigned to global types is always non-increasing during execution.

The mapping $\mathcal{C}$ from locations to typing contexts, written $\{\ell_1 \mapsto \Gamma_1, \ldots, \ell_n \mapsto \Gamma_n\}$, associates each location with its *global context* that justifies its global type.

We also extend the projection operation from (local) contexts to global contexts in the natural way:

$$\mathcal{C}\restriction_\ell = \{\ell_1 \mapsto \Gamma_1, \ldots, \ell_n \mapsto \Gamma_n\}\restriction_\ell \overset{\text{def}}{=} (\Gamma_1, \ldots, \Gamma_n)\restriction_\ell$$

Furthermore, we introduce an auxiliary *balance* (or *lazy potential*) mapping $\mathcal{B}$ from locations to non-negative rational numbers.

The balance mapping will be used to keep track of the partial costs of thunks that have been paid in advance by applications of the PREPAY rule.

Note that these auxiliary mappings are needed only in the soundness proof of the analysis for bookkeeping purposes, but are not part of the operational semantics — in particular, they do not incur run-time costs.

5.5.3 Potential

We now define the potential of an augmented expression with respect to a heap and an annotated type.

Definition 5.6 (Potential)**.** The potential assigned to an augmented expression $\widehat{e}$ of type A under heap $\mathcal{H}$, written $\phi_{\mathcal{H}}(\widehat{e}{:}A)$, is defined in (5.1) within Figure 5.8.

The potential of data constructors is obtained by summing the type annotation with the (possibly recursive) potential contributed by each of the arguments. Note how the potential of data constructors is unwrapped from thunk types. The potential of expressions other than data constructors is always zero.

Equation (5.2) extends the definition to typing contexts in the natural way. Equation (5.3) defines potential for global contexts, but considers only thunks that are not under evaluation.

$$\phi_{\mathcal{H}}(\widehat{e}{:}A) \overset{\mathsf{def}}{=} \begin{cases} p + \sum_i \phi_{\mathcal{H}}(\mathcal{H}(\ell_i){:}B_i[A/X]) & \text{if } A = \mu X.\{\cdots \mid c{:}(p, \vec{B}) \mid \cdots\} \text{ and } \widehat{e} = c\,\vec{\ell} \\ \phi_{\mathcal{H}}(\widehat{e}{:}B) & \text{if } A = \mathsf{T}^q(B) \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

$$\phi_{\mathcal{H}}(\Gamma) \overset{\mathsf{def}}{=} \sum \{\phi_{\mathcal{H}}(\mathcal{H}(x){:}A) \mid x{:}A \in \Gamma\} \tag{5.2}$$

$$\Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{C}) \overset{\mathsf{def}}{=} \sum \left\{ \phi_{\mathcal{H}}(\mathcal{C}(\ell)) \,\middle|\, \ell \in \mathrm{dom}(\mathcal{H}) \text{ and } \ell \notin \mathcal{L} \text{ and } \mathcal{H}(\ell) \text{ is not a } \textit{whnf} \right\} \tag{5.3}$$

$$\Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B}) \overset{\mathsf{def}}{=} \sum \left\{ \quad \mathcal{B}(\ell) \quad \,\middle|\, \ell \in \mathrm{dom}(\mathcal{H}) \text{ and } \ell \notin \mathcal{L} \text{ and } \mathcal{H}(\ell) \text{ is not a } \textit{whnf} \right\} \tag{5.4}$$

Figure 5.8: Potential

Finally, (5.4) defines a convenient shorthand notation for a similar summation over the balance.

Note that for cyclic data structures, the potential is only defined if all the type annotations of all nodes encountered along a cycle are zero (the overall potential must therefore also be zero). For example, consider the heap $\mathcal{H} = [\ell_0 \mapsto \texttt{True}, \ell_1 \mapsto \texttt{Cons}\ \ell_0\ \ell_1]$ where ℓ_1 represents an infinite list of booleans `True` as a cyclic list of length 1. Potential

$$\phi_{\mathcal{H}}\big(\ell_1{:}\mathsf{T}^1(\mu X.\{\texttt{Cons}{:}(0, (\mathsf{T}^1(\mu Y.\{\texttt{True}{:}(0, ()) \mid \texttt{False}{:}(1, ())\}), \mathsf{T}^1(X))) \mid \texttt{Nil}{:}(1, ())\})\big)$$

is zero, whereas potentials

$$\phi_{\mathcal{H}}\big(\ell_1{:}\mathsf{T}^1(\mu X.\{\texttt{Cons}{:}(1, (\mathsf{T}^1(\mu Y.\{\texttt{True}{:}(0, ()) \mid \texttt{False}{:}(1, ())\}), \mathsf{T}^1(X))) \mid \texttt{Nil}{:}(1, ())\})\big)$$
$$\phi_{\mathcal{H}}\big(\ell_1{:}\mathsf{T}^1(\mu X.\{\texttt{Cons}{:}(0, (\mathsf{T}^1(\mu Y.\{\texttt{True}{:}(1, ()) \mid \texttt{False}{:}(1, ())\}), \mathsf{T}^1(X))) \mid \texttt{Nil}{:}(1, ())\})\big)$$

are not defined.

The next lemma formalises the intuition that sharing splits the potential of a type.

Lemma 5.7 (Potential Splitting). *If $\curlyvee(A \mid A_1, \ldots, A_n)$ then for all $\widehat{e}$ such that the potentials are defined, we have $\phi_{\mathcal{H}}(\widehat{e}{:}A) \geq \sum_i \phi_{\mathcal{H}}(\widehat{e}{:}A_i)$.*

Proof Sketch. First note that the results follow immediately if $\widehat{e}$ is not in *whnf* or is a λ-abstraction (because potentials are zero in those cases). The potential is also zero if $\widehat{e}$ is a constructor that is part of a cycle (since otherwise it would be undefined). The remaining case is for a constructor with no cycles, i.e. a directed acyclic graph (DAG). The proof is then by induction on the length of the longest path. See Section 5.5.6.5 for the detailed proof. □

This lemma has an important corollary when A occurs as one of the types on the right hand side.

Corollary 5.8 (Potential Remaining)**.** *If* $\curlyvee(A \,|\, A, B_1, \ldots, B_n)$ *then for all* $\widehat{e}$ *such that the potentials are defined, we have* $\phi_{\mathcal{H}}(\widehat{e}{:}B_i) = 0$ *for all* i*.*

Proof. This is a direct corollary of Lemma 5.7. □

It also follows as corollary that a supertype of a type A has potential that is no greater than that of A.

Corollary 5.9 (Potential Subtype)**.** *If* $A <: B$ *then for all* $\widehat{e}$ *such that the potentials are defined, we have* $\phi_{\mathcal{H}}(\widehat{e}{:}A) \geq \phi_{\mathcal{H}}(\widehat{e}{:}B)$*.*

Proof. By the definition of subtyping, this is a direct corollary of Lemma 5.7 for the case when $n = 1$. □

5.5.4 Consistency and Compatibility

We now define the principal invariants for proving the soundness of our analysis, namely, *consistency* and *compatibility* relations between a heap configuration and the global types, contexts and balance.

We proceed by first defining type consistency of a single location and then extending it to a whole heap.

Definition 5.10 (Type Consistency of Locations)**.** We say that location ℓ admits type $\mathsf{T}^q(A)$ under context Γ, balance $\mathcal{B}$ and heap configuration $(\mathcal{H}, \mathcal{L})$, and write $\Gamma, \mathcal{B}; \mathcal{H}, \mathcal{L} \vdash_{\mathsf{LOC}} \ell : \mathsf{T}^q(A)$, if one of the following cases holds:

(Loc1) $\mathcal{H}(\ell)$ is in *whnf* and $\Gamma \xmapsto{0} \mathcal{H}(\ell) : A$

(Loc2) $\mathcal{H}(\ell)$ not in *whnf* and $\ell \notin \mathcal{L}$ and $\Gamma \xmapsto{q \,+\, \mathcal{B}(\ell)} \mathcal{H}(\ell) : A$

(Loc3) $\mathcal{H}(\ell)$ not in *whnf* and $\ell \in \mathcal{L}$ and $\Gamma = \emptyset$

The three cases in the above definition are mutually exclusive: Loc1 applies when the expression in the heap is already in *whnf*; otherwise Loc2 or Loc3 apply, depending on whether the thunk is or is not under evaluation.

Note that for LOC2 the balance $\mathcal{B}(\ell)$ associated with location ℓ is added to the available resources for typing the thunk $\mathcal{H}(\ell)$, effectively reducing its cost by the prepaid amount. Once evaluation has begun (LOC3), or once it has completed (LOC1), the balance is considered spent. However, we never lower or reset the balance, since it is simply ignored in such cases.

Definition 5.11 (Type Consistency of Heaps)**.** We say that a heap state $(\mathcal{H}, \mathcal{L})$ is consistent with global contexts, global types and balance, and write $\mathcal{C}, \mathcal{B} \vdash_{\textsf{MEM}} (\mathcal{H}, \mathcal{L}) : \mathcal{M}$, if and only if for all $\ell \in \mathrm{dom}(\mathcal{H})$: $\mathcal{C}(\ell), \mathcal{B}; \mathcal{H}, \mathcal{L} \vdash_{\textsf{LOC}} \ell : \mathcal{M}(\ell)$ holds.

Definition 5.12 (Global Compatibility)**.** We say that a global type $\mathcal{M}$ is *compatible* with context Γ and a global context $\mathcal{C}$, written $\curlyvee(\mathcal{M} \,|\, \Gamma, \mathcal{C})$, if and only if $\curlyvee(\mathcal{M}(\ell) \,|\, \Gamma\restriction_{\ell}, \mathcal{C}\restriction_{\ell})$ for all $\ell \in \mathrm{dom}(\mathcal{M})$.

Definition 5.11 requires the type consistency of each specific location. Definition 5.12 requires for each location that the global type accounts for the joint potential of all references (in both the local and global contexts).

5.5.5 Soundness of the Proof System

We can now state the soundness of our analysis as an augmented type preservation result.

Theorem 5.13 (Soundness)**.** *If the following statements hold*

$$\Gamma \overset{q}{\longmapsto} e : A \tag{5.5}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash e \Downarrow w, \mathcal{H}' \tag{5.6}$$

$$\mathcal{C}, \mathcal{B} \vdash_{\textsf{MEM}} (\mathcal{H}, \mathcal{L}) : \mathcal{M} \tag{5.7}$$

$$\curlyvee(\mathcal{M} \,|\, (\Gamma, \Theta), \mathcal{C}) \tag{5.8}$$

then for all $t \in \mathbb{Q}_0^+$ *and* $m \in \mathbb{N}$ *with*

$$m \geq t + q + \phi_{\mathcal{H}}(\Gamma) + \phi_{\mathcal{H}}(\Theta) + \Phi_{\mathcal{H}}^{\mathcal{L}}(\mathcal{C}) + \Phi_{\mathcal{H}}^{\mathcal{L}}(\mathcal{B}) \tag{5.9}$$

there exist Γ'*,* $\mathcal{C}'$*,* $\mathcal{B}'$*,* $\mathcal{M}'$ *and* $m', m'' \in \mathbb{N}$ *such that the following statements also hold*

$$\Gamma' \overset{0}{\longmapsto} w : A \tag{5.10}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{m''}{\longmapsto} e \Downarrow w, \mathcal{H}' \tag{5.11}$$

$$\mathcal{M} <: \mathcal{M}' \tag{5.12}$$

$$\mathcal{C}', \mathcal{B}' \vdash_{\textsf{MEM}} (\mathcal{H}', \mathcal{L}) : \mathcal{M}' \tag{5.13}$$

$$\curlyveedownarrow(\mathcal{M}' \,|\, (\Gamma', \Theta), \mathcal{C}') \tag{5.14}$$

$$m' \geq t + \phi_{\mathcal{H}'}(w{:}A) + \phi_{\mathcal{H}'}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}'}\big(\mathcal{C}'\big) + \Phi^{\mathcal{L}}_{\mathcal{H}'}\big(\mathcal{B}'\big) \tag{5.15}$$

$$m - m' \geq m'' \tag{5.16}$$

Informally, the soundness theorem reads as follows: if an expression e admits a type A (5.5), its evaluation is successful (5.6) and the heap can be consistently typed (5.7) (5.8), then the resulting *whnf* also admits type A (5.10). Furthermore, the resulting heap can also be typed (5.12) (5.13) (5.14) and the static bounds that are obtained from the typing of e give safe resource estimates for evaluation (5.9) (5.11) (5.15) (5.16).

The arbitrary value t is used to carry over excess potential which is not used for the immediate evaluation but will be needed in subsequent ones (e.g. for the argument of an application). Similarly, the context Θ is used to preserve types for variables that are not in the current scope but that are necessary for subsequent evaluations (e.g. the alternatives of the case).

Note that type preservation — i.e. the fact that expression e and its *whnf* w both have judgements with the same type A — could be proven separately from the main theorem, but obviously only if there were no resource related type annotations in the relevant statements.

We present here a proof sketch of our main theorem; a detailed proof is available in Section 5.5.6.

Proof Sketch. The proof follows by induction on the lengths of the derivations of (5.6) and (5.5) ordered lexicographically, with the derivation of the evaluation taking priority over the typing derivation. We proceed by case analysis of the typing rule used in premise (5.5), considering just some representative cases.

Case Var**:** The typing premise $\ell{:}\mathsf{T}^{q}(A) \overset{q}{\vdash} \ell : A$ is an axiom. By inversion of the evaluation premise, we obtain $\mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash \mathcal{H}(\ell) \Downarrow w, \mathcal{H}'$. In order to apply induction to the evaluation of the thunk $\mathcal{H}(\ell)$, we take the typing context from the hypothesis of type consistency for the location ℓ. We apply induction to a typing with the global type $\mathcal{M}(\ell)$ rather than the local type $\mathsf{T}^{q}(A)$ in the local context. This gives us a stronger conclusion with a context that we can then split using Lemma 5.7 to justify type consistency for the heap update and the local context for the answer. Finally, we require an auxiliary result to ensure that if the update introduces a cycle, the locations on the cycle can be assigned a type with zero potential (Lemma 5.17 in Section 5.5.6).

Case Let**:** The typing premise is $\Gamma, \Delta \overset{1+q+p}{\vdash}$ let $x = \widehat{e}$ in $e : C$ and evaluation premise gives us $\mathcal{H}_0, \mathcal{S}, \mathcal{L} \vdash e[\ell/x] \Downarrow w, \mathcal{H}'$ where $\mathcal{H}_0 = \mathcal{H}[\ell \mapsto \widehat{e}[\ell/x]]$ is the heap extended with a new location ℓ and thunk. To apply induction to the evaluation of $e[\ell/x]$ we reestablish the consistency to the new location ℓ; this is done using Γ from the typing hypothesis together with an idempotent type for self-references to ℓ. Applying induction then yields all required conclusions.

Case Case**:** The typing premise is $\Gamma, \Delta \overset{q+q'}{\vdash}$ case e of $\{c_i\ \overrightarrow{x_i}$ -> $e_i\}_{i=1}^{n} : C$ and, by inversion of the type rule, we get a typing $\Gamma \overset{q}{\vdash} e : B$ for e, where $B = \mu X.\{c_1 : (p_1, \overrightarrow{A_1}) | \cdots | c_n : (p_n, \overrightarrow{A_n})\}$ is some data type for constructors $c_1, \ldots, c_n$. We apply induction to the evaluation of e. We then apply induction to $e_k[\vec{\ell}/\overrightarrow{x_k}]$ and obtain the proof obligation. To establish the premise (5.9) on m', we use the definition of potential: $\phi_{\mathcal{H}}(c_k\ \vec{\ell}{:}B) = p_k + \sum_i \phi_{\mathcal{H}}(\mathcal{H}(\ell_i){:}A_i[B/X])$, i.e. the potential of the constructor c_k is the sum of the type annotation p_k plus the potential of its context.

Case Prepay**:** The typing premise is $\Gamma, \ell{:}\mathsf{T}^{q'_0+q'}(A) \overset{q+q'}{\vdash} e : C$. We want to apply induction to the typing $\Gamma, \ell{:}\mathsf{T}^{q'_0}(A) \overset{q}{\vdash} e : C$, obtained by inversion of the type rule, and we use the same evaluation premise, since Prepay is a structural rule. We reflect the prepayment of q' in the global balance $\mathcal{B}' = \mathcal{B}[\ell \mapsto q' + \mathcal{B}(\ell)]$. Let $\mathsf{T}^{r}(A') = \mathcal{M}(\ell)$, $k = \max(r - q', 0)$ and $\mathcal{M}' = \mathcal{M}[\ell \mapsto \mathsf{T}^{k}(A')]$, i.e. we want to show that we can lower (by q') the global cost of thunk types for location ℓ. In order to do so, we need to reestablish type consistency and global compatibility for the new $\mathcal{B}'$ and $\mathcal{M}'$. The important part is to show that $\curlyvee\Big(\mathsf{T}^{k}(A') \Big| \mathsf{T}^{q'_0}(A)\Big)$, in particular $k \leq q'_0$, but this is equivalent to show that $\max(r – q', 0) \leq q'_0 \iff r – q' \leq q'_0 \wedge 0 \leq$

q'_0, where the latter follows from the non-negativity of q'_0, and $r - q' \leq q'_0 \iff r \leq q'_0 + q'$, which holds by the compatibility premise $\curlyvee\left(\mathsf{T}^{r}(A') \,\middle|\, \mathsf{T}^{q'_0+q'}(A)\right)$.

□

The soundness proof presented in this thesis does not require co-induction for proving memory consistency. This contrasts with previous amortised analyses that deal with recursive closures [JLHH10, Jos10]. It should be noted that the same reason, i.e. proving the consistency of recursive closures, also caused Milner and Tofte [MT91] to resort to co-induction. The analyses presented in this thesis, however, do not need a co-inductively defined consistency for recursive closures, but instead rely upon the convention that all variable names are sufficiently fresh, hence a single, global, environment suffices. Therefore, checking all locations for this global environment once suffices since a function value does not need a recursive check for its consistency with a new environment.

The next section includes the detailed proof of the soundness theorem for our analysis.

5.5.6 Detailed Proofs

We begin with an auxiliary definition and lemma that will be used in the proof of the soundness of the analysis in the case VAR for updating a location with a *whnf*.

5.5.6.1 Minor Lemmas

Lemma 5.14 (Subtyping is a partial order)**.** $<:$ *is a partial order.*

Proof. Straightforward by induction on the type structure and the definition of sharing (Figure 5.2). □

Lemma 5.15 (Idempotent Subtypes)**.** *If* $\curlyvee(A \mid A, A')$ *then* $\curlyvee(A' \mid A', A')$ *as well.*

Proof. Straightforward by induction on the type structure and the definition of sharing (Figure 5.2). □

We now present the proof of Lemma 5.3 (CONS Inversion), followed by the proof of Lemma 5.4 (ABS Inversion).

5.5.6.2 Inversion Lemma for Constructors

Lemma 5.3 (CONS Inversion). *If* $\Gamma \overset{0}{\vdash} c\ \vec{y} : B$ *then* $B = \mu X.\{\cdots \mid c : (p, \vec{A}) \mid \cdots\}$ *and* $\curlyvee(\Gamma \mid y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X])$.

Proof. A typing with conclusion $\Gamma \overset{0}{\vdash} c\ \vec{y} : B$ must result from axiom CONS followed by (possibly zero) uses of structural rules. The proof follows by induction on the structural rules, considering each rule separately. For rules RELAX and PREPAY induction is trivial since both type judgements have zero on the turnstile. For the remaining structural rules the proof follows by transitivity of the sharing relation. We now consider each of the remaining structural rules.

Case WEAK: We have $\Gamma, x_{n+1}{:}C_{n+1} \overset{0}{\vdash} c\ \vec{y} : B$. Applying induction to the premise of rule WEAK $\Gamma \overset{0}{\vdash} c\ \vec{y} : B$ we obtain

$$B = \mu X.\{\cdots \mid c : (p, \vec{A}) \mid \cdots\}$$

as required for the conclusion, and

$$\curlyvee(\Gamma \mid y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X])$$

Let $\Gamma = \{x_1{:}C_1, \ldots, x_n{:}C_n\}$. By the definition of sharing (Figure 5.3) we know that

$$\curlyvee(x_1{:}C_1, \ldots, x_n{:}C_n \mid y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X])$$

iff there is a partition $\Delta_1, \ldots, \Delta_n$ of $\{y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X]\}$ such that $\curlyvee(x_i{:}C_i \mid \Delta_i)$ holds and $\mathrm{dom}(\Delta_i) \subseteq \{x_i\}$, for $(1 \leq i \leq n)$.

Let $\Delta_{n+1} = \emptyset$. Since $\curlyvee(x_{n+1}{:}C_{n+1} \mid \Delta_{n+1})$ holds (by SHAREEMPTYCTX) and $\mathrm{dom}(\Delta_{n+1}) \subseteq \{x_{n+1}\}$, again by definition of sharing we have

$$\curlyvee(\Gamma, x_{n+1}{:}C_{n+1} \mid y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X])$$

as required.

Case Supertype**:** We have $\Gamma, x_{n+1}{:}C'_{n+1} \vdash^{0} c\,\vec{y} : B$. The premises of rule Supertype are $\Gamma, x_{n+1}{:}C_{n+1} \vdash^{0} c\,\vec{y} : B$ and $\curlyvee\big(C'_{n+1} \,|\, C_{n+1}\big)$. Applying induction to $\Gamma, x_{n+1}{:}C_{n+1} \vdash^{0} c\,\vec{y} : B$ we obtain

$$B = \mu X.\{\cdots \mid c : (p, \vec{A}) \mid \cdots\}$$

as required for the conclusion, and

$$\curlyvee(\Gamma, x_{n+1}{:}C_{n+1} \,|\, y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X]\,)$$

Let $\Gamma = \{x_1{:}C_1, \ldots, x_n{:}C_n\}$. By the definition of sharing (Figure 5.3) we know that

$$\curlyvee(x_1{:}C_1, \ldots, x_n{:}C_n, x_{n+1}{:}C_{n+1} \,|\, y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X]\,)$$

iff there is a partition $\Delta_1, \ldots, \Delta_n, \Delta_{n+1}$ of $\{y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X]\}$ such that $\curlyvee(x_i{:}C_i \,|\, \Delta_i\,)$ holds and $\mathrm{dom}(\Delta_i) \subseteq \{x_i\}$, for $(1 \leq i \leq n+1)$.

From $\curlyvee\big(C'_{n+1} \,|\, C_{n+1}\big)$ and $\curlyvee(x_{n+1}{:}C_{n+1} \,|\, \Delta_{n+1}\,)$ by the transitivity of sharing we have

$$\curlyvee\big(x_{n+1}{:}C'_{n+1} \,|\, \Delta_{n+1}\big)$$

Thus by definition of sharing we have

$$\curlyvee\big(\Gamma, x_{n+1}{:}C'_{n+1} \,|\, y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X]\big)$$

as required.

Case Subtype**:** We have $\Gamma \vdash^{0} c\,\vec{y} : C$. The premises of rule Subtype are $\Gamma \vdash^{0} c\,\vec{y} : B$ and $\curlyvee(B \,|\, C\,)$. Applying induction to $\Gamma \vdash^{0} c\,\vec{y} : B$ we obtain

$$B = \mu X.\{\cdots \mid c : (p, \vec{A}) \mid \cdots\}$$

and

$$\curlyvee(\Gamma \,|\, y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X]\,)$$

From $\curlyvee(B \,|\, C\,)$ we know

$$C = \mu X.\{\cdots \mid c : (p', \vec{A}') \mid \cdots\}$$

where $p \leq p'$ and $\curlyvee\left(\vec{A} \middle| \vec{A'}\right)$. Also, $\curlyvee(y_i{:}A_i[B/X] \,|\, y_i{:}A'_i[C/X])$ for $(1 \leq i \leq k)$. By the transitivity of sharing we obtain

$$\curlyvee(\Gamma \,|\, y_1{:}A'_1[C/X], \ldots, y_k{:}A'_k[C/X])$$

as required.

Case SHARE**:** We have $\Gamma, x{:}C' \vdash^{0} c\ \vec{y} : B$. Applying induction to the premise of rule SHARE $\Gamma, x{:}C'_1, x{:}C'_2 \vdash^{0} c\ \vec{y} : B$ we obtain

$$B = \mu X.\{\cdots \mid c : (p, \vec{A}) \mid \cdots\}$$

as required for the conclusion, and

$$\curlyvee(\Gamma, x{:}C'_1, x{:}C'_2 \,|\, y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X])$$

Let $\Gamma = \{x_1{:}C_1, \ldots, x_n{:}C_n\}$. By the definition of sharing (Figure 5.3) we know that

$$\curlyvee(x_1{:}C_1, \ldots, x_n{:}C_n, x{:}C'_1, x{:}C'_2 \,|\, y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X])$$

iff there is a partition $\Delta_1, \ldots, \Delta_n, \Delta'_1, \Delta'_2$ of $\{y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X]\}$ such that $\curlyvee(x_i{:}C_i \,|\, \Delta_i)$ holds and $\mathrm{dom}(\Delta_i) \subseteq \{x_i\}$, for $(1 \leq i \leq n)$, and $\curlyvee(x{:}C'_1 \,|\, \Delta'_1)$, $\curlyvee(x{:}C'_2 \,|\, \Delta'_2)$ hold and $\mathrm{dom}(\Delta'_1 \cup \Delta'_2) \subseteq \{x\}$.

From $\curlyvee(x{:}C'_1 \,|\, \Delta'_1)$ and $\curlyvee(x{:}C'_2 \,|\, \Delta'_2)$ we have $\curlyvee(x{:}C'_1, x{:}C'_2 \,|\, \Delta'_1, \Delta'_2)$. From $\curlyvee(C' \,|\, C'_1, C'_2)$ (also premise of rule SHARE) and the transitivity of sharing we have $\curlyvee(x{:}C' \,|\, \Delta'_1, \Delta'_2)$. By definition of sharing we have

$$\curlyvee(\Gamma, x{:}C' \,|\, y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X])$$

as required.

This concludes the proof of the CONS Inversion. □

5.5.6.3 Inversion Lemma for λ-abstractions

Lemma 5.4 (Abs Inversion)**.** *If* $\Gamma \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C$ *then there exists* Γ' *such that* $\curlyvee(\Gamma \,|\, \Gamma')$*,* $\curlyvee(\Gamma' \,|\, \Gamma', \Gamma')$*,* $x \notin \mathrm{dom}(\Gamma')$ *and* $\Gamma', x{:}A \overset{q}{\vdash} e : C$*.*

Proof. A typing $\Gamma \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C$ must result from an application of the rule Abs followed by (possibly zero) uses of structural rules. The proof follows by induction on the structural rules, considering each rule separately. For rules Relax and Prepay induction is trivial since both type judgements have zero on the turnstile. For the remaining structural rules the proof follows by transitivity of the sharing relation. We now consider each of the remaining structural rules.

Case Weak**:** We have $\Gamma, y_{n+1}{:}B_{n+1} \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C$ and, as a premise of rule Weak,

$$\Gamma \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C$$

Applying induction to $\Gamma \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C$ we obtain Γ' such that $\curlyvee(\Gamma \,|\, \Gamma')$, $\curlyvee(\Gamma' \,|\, \Gamma', \Gamma')$, $x \notin \mathrm{dom}(\Gamma')$ and $\Gamma', x{:}A \overset{q}{\vdash} e : C$.

Let $\Gamma = \{y_1{:}B_1, \ldots, y_n{:}B_n\}$. By the definition of sharing (Figure 5.3) we know that

$$\curlyvee(y_1{:}B_1, \ldots, y_n{:}B_n \,|\, \Gamma')$$

iff there is a partition $\Delta_1, \ldots, \Delta_n$ of Γ' such that $\curlyvee(y_i{:}B_i \,|\, \Delta_i)$ holds and $\mathrm{dom}(\Delta_i) \subseteq \{y_i\}$, for $(1 \leq i \leq n)$.

Let $\Delta_{n+1} = \emptyset$. From ShareEmptyCtx we have $\curlyvee(y_{n+1}{:}B_{n+1} \,|\, \Delta_{n+1})$. By the definition of sharing we have

$$\curlyvee(\Gamma, y_{n+1}{:}B_{n+1} \,|\, \Gamma')$$

as required.

Case Supertype**:** We have $\Gamma, y_{n+1}{:}B'_{n+1} \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C$ and, as a premise of rule Supertype,

$$\Gamma, y_{n+1}{:}B_{n+1} \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C$$

where $\curlyvee\big(B'_{n+1} \,|\, B_{n+1}\big)$. Applying induction to $\Gamma, y_{n+1}{:}B_{n+1} \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C$ we obtain Γ' such that $\curlyvee(\Gamma, y_{n+1}{:}B_{n+1} \,|\, \Gamma'\,)$, $\curlyvee(\Gamma' \,|\, \Gamma', \Gamma'\,)$, $x \notin \mathrm{dom}(\Gamma')$ and $\Gamma', x{:}A \overset{q}{\vdash} e : C$.

Let $\Gamma = \{y_1{:}B_1, \ldots, y_n{:}B_n\}$. By the definition of sharing (Figure 5.3) we know that

$$\curlyvee(y_1{:}B_1, \ldots, y_n{:}B_n, y_{n+1}{:}B_{n+1} \,|\, \Gamma'\,)$$

iff there is a partition $\Delta_1, \ldots, \Delta_n, \Delta_{n+1}$ of Γ' such that $\curlyvee(y_i{:}B_i \,|\, \Delta_i\,)$ holds and $\mathrm{dom}(\Delta_i) \subseteq \{y_i\}$, for $(1 \le i \le n+1)$.

From $\curlyvee\big(B'_{n+1} \,|\, B_{n+1}\big)$ and $\curlyvee(y_{n+1}{:}B_{n+1} \,|\, \Delta_{n+1}\,)$ by the transitivity of sharing we have

$$\curlyvee\big(y_{n+1}{:}B'_{n+1} \,|\, \Delta_{n+1}\big)$$

Thus, by the definition of sharing we obtain

$$\curlyvee\big(\Gamma, y_{n+1}{:}B'_{n+1} \,|\, \Gamma'\,\big)$$

as required.

Case SUBTYPE**:** We have $\Gamma \overset{0}{\vdash} \lambda x.e : A' \xrightarrow{q'} C'$ and, as a premise of rule SUBTYPE,

$$\Gamma \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C$$

where $\curlyvee\Big(A \xrightarrow{q} C \,\Big|\, A' \xrightarrow{q'} C'\Big)$. Applying induction to $\Gamma \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C$ we obtain Γ' such that $\curlyvee(\Gamma \,|\, \Gamma'\,)$, $\curlyvee(\Gamma' \,|\, \Gamma', \Gamma'\,)$, $x \notin \mathrm{dom}(\Gamma')$ and $\Gamma', x{:}A \overset{q}{\vdash} e : C$.

From $\curlyvee\Big(A \xrightarrow{q} C \,\Big|\, A' \xrightarrow{q'} C'\Big)$ we know $q \le q'$, $\curlyvee(A' \,|\, A\,)$ and $\curlyvee(C \,|\, C'\,)$.

From $\Gamma', x{:}A \overset{q}{\vdash} e : C$ applying rules SUPERTYPE (with $\curlyvee(A' \,|\, A)$), RELAX (with $q \le q'$) and SUBTYPE (with $\curlyvee(C \,|\, C'\,)$) we obtain

$$\Gamma', x{:}A' \overset{q'}{\vdash} e : C'$$

as required.

Case Share**:** We have $\Gamma, y{:}B' \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C$ and, as a premise of rule Share,

$$\Gamma, y{:}B'_1, y{:}B'_2 \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C$$

where $\curlyvee(B' \,|\, B'_1, B'_2\,)$. Applying induction to $\Gamma, y{:}B'_1, y{:}B'_2 \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C$ we obtain Γ' such that $\curlyvee(\Gamma, y{:}B'_1, y{:}B'_2 \,|\, \Gamma'\,)$, $\curlyvee(\Gamma' \,|\, \Gamma', \Gamma'\,)$, $x \notin \mathrm{dom}(\Gamma')$ and $\Gamma', x{:}A \overset{q}{\vdash} e : C$.

Let $\Gamma = \{y_1{:}B_1, \ldots, y_n{:}B_n\}$. By the definition of sharing (Figure 5.3) we know that

$$\curlyvee(y_1{:}B_1, \ldots, y_n{:}B_n, y{:}B'_1, y{:}B'_2 \,|\, \Gamma'\,)$$

iff there is a partition $\Delta_1, \ldots, \Delta_n, \Delta'_1, \Delta'_2$ of Γ' such that $\curlyvee(y_i{:}B_i \,|\, \Delta_i\,)$ holds and $\mathrm{dom}(\Delta_i) \subseteq \{y_i\}$, for $(1 \leq i \leq n)$, and $\curlyvee(y{:}B'_1 \,|\, \Delta'_1\,)$ and $\curlyvee(y{:}B'_2 \,|\, \Delta'_2\,)$ hold, and $\mathrm{dom}(\Delta'_1 \cup \Delta'_2) \subseteq \{y\}$.

From $\curlyvee(y{:}B'_1 \,|\, \Delta'_1\,)$ and $\curlyvee(y{:}B'_2 \,|\, \Delta'_2\,)$ we have $\curlyvee(y{:}B'_1, y{:}B'_2 \,|\, \Delta'_1, \Delta'_2\,)$. From $\curlyvee(B' \,|\, B'_1, B'_2\,)$ and the transitivity of sharing we have $\curlyvee(y{:}B' \,|\, \Delta'_1, \Delta'_2\,)$. By definition of sharing we have

$$\curlyvee(\Gamma, y{:}B' \,|\, \Gamma'\,)$$

as required.

This concludes the proof of the Abs Inversion. □

We now present the proof of Lemma 5.5 (Context Splitting), followed by the proof of Lemma 5.7 (Potential Splitting).

5.5.6.4 Context Splitting Lemma

Lemma 5.5 (Context Splitting). *If $\Gamma \overset{0}{\vdash} w : A$, where w is an expression in* whnf *and $\curlyvee(A \,|\, A_1, A_2\,)$; then there exist Γ_1, Γ_2 such that $\curlyvee(\Gamma \,|\, \Gamma_1, \Gamma_2\,)$, $\Gamma_1 \overset{0}{\vdash} w : A_1$ and $\Gamma_2 \overset{0}{\vdash} w : A_2$.*

Proof. Expression w is either a constructor application or a λ-abstraction. The proof follows by considering the two cases separately.

Z/A A/B B/Z

Case $w = c\,\vec{y}$**:** We have $\Gamma \overset{0}{\vdash} c\,\vec{y} : A$ and $\curlyvee(A \,|\, A_1, A_2\,)$. By applying Lemma 5.3 we obtain $A = \mu X.\{\cdots \,|\, c : (p, \vec{B}) \,|\, \cdots\}$ and $\curlyvee(\Gamma \,|\, y_1{:}B_1[A/X], \ldots, y_k{:}B_k[A/X]\,)$. From $\curlyvee(A \,|\, A_1, A_2\,)$ we also obtain

$$A_1 = \mu X.\{\cdots \,|\, c : (p', \vec{B'}) \,|\, \cdots\}$$
$$A_2 = \mu X.\{\cdots \,|\, c : (p'', \vec{B''}) \,|\, \cdots\}$$

Applying rule Cons we obtain

$$y_1{:}B'_1[A_1/X], \ldots, y_k{:}B'_k[A_1/X] \overset{0}{\vdash} c\,\vec{y} : A_1$$
$$y_1{:}B''_1[A_2/X], \ldots, y_k{:}B''_k[A_2/X] \overset{0}{\vdash} c\,\vec{y} : A_2$$

as required, by considering

$$\Gamma_1 = y_1{:}B'_1[A_1/X], \ldots, y_k{:}B'_k[A_1/X]$$
$$\Gamma_2 = y_1{:}B''_1[A_2/X], \ldots, y_k{:}B''_k[A_2/X]$$

We are left to prove $\curlyvee(\Gamma \,|\, \Gamma_1, \Gamma_2\,)$. Note that by definition of sharing and $\curlyvee(A \,|\, A_1, A_2\,)$

$$\curlyvee(y_i{:}B_i[A/X] \,|\, y_i{:}B'_i[A_1/X], y_i{:}B''_i[A_2/X]\,)$$

for $(1 \leq i \leq k)$. Thus, we have $\curlyvee(y_1{:}B_1[A/X], \ldots, y_k{:}B_k[A/X] \,|\, \Gamma_1, \Gamma_2\,)$. By transitivity of sharing we obtain $\curlyvee(\Gamma \,|\, \Gamma_1, \Gamma_2\,)$ as required.

Case $w = \lambda x.e$**:** We have $\Gamma \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C$ and $\curlyvee\big(A \xrightarrow{q} C \,\big|\, A_1 \xrightarrow{q_1} C_1, A_2 \xrightarrow{q_2} C_2\big)$. By applying Lemma 5.4 we obtain Γ' such that $\curlyvee(\Gamma \,|\, \Gamma'\,)$, $\curlyvee(\Gamma' \,|\, \Gamma', \Gamma'\,)$, $x \notin \mathrm{dom}(\Gamma')$ and $\Gamma', x{:}A \overset{q}{\vdash} e : C$.

From $\curlyvee\big(A \xrightarrow{q} C \,\big|\, A_1 \xrightarrow{q_1} C_1, A_2 \xrightarrow{q_2} C_2\big)$ we also obtain $\curlyvee(A_1 \,|\, A\,)$, $q \leq q_1$, $\curlyvee(C \,|\, C_1\,)$, $\curlyvee(A_2 \,|\, A\,)$, $q \leq q_2$ and $\curlyvee(C \,|\, C_2\,)$.

Let $\Gamma_1 = \Gamma_2 = \Gamma'$. From $\Gamma_1, x{:}A \overset{q}{\vdash} e : C$ applying rules Supertype (with $\curlyvee(A_1 \,|\, A\,)$), Relax (with $q \leq q_1$), Subtype (with $\curlyvee(C \,|\, C_1\,)$) and Abs (with $\curlyvee(\Gamma_1 \,|\, \Gamma_1, \Gamma_1\,)$ and $x \notin \mathrm{dom}(\Gamma_1)$) we obtain $\Gamma_1 \overset{0}{\vdash} \lambda x.e : A_1 \xrightarrow{q_1} C_1$ as required. Also from $\Gamma_2, x{:}A \overset{q}{\vdash} e : C$ applying rules Supertype (with $\curlyvee(A_2 \,|\, A\,)$), Relax (with $q \leq q_2$), Subtype (with $\curlyvee(C \,|\, C_2\,)$) and Abs (with $\curlyvee(\Gamma_2 \,|\, \Gamma_2, \Gamma_2\,)$ and $x \notin \mathrm{dom}(\Gamma_2)$) we obtain $\Gamma_2 \overset{0}{\vdash} \lambda x.e : A_2 \xrightarrow{q_2} C_2$ as required.

We are left to prove $\curlyvee(\Gamma \,|\, \Gamma_1, \Gamma_2\,)$. This is equivalent to $\curlyvee(\Gamma \,|\, \Gamma', \Gamma'\,)$ and follows from $\curlyvee(\Gamma \,|\, \Gamma'\,)$

and $\curlyvee(\Gamma' \,|\, \Gamma', \Gamma'\,)$ by the transitivity of sharing.

We thus conclude the proof of Context Splitting. □

5.5.6.5 Potential Splitting Lemma

Lemma 5.7 (Potential Splitting). *If $\curlyvee(A \,|\, A_1, \ldots, A_n\,)$ then for all $\widehat{e}$ such that the potentials are defined, we have $\phi_{\mathcal{H}}(\widehat{e}{:}A) \geq \sum_i \phi_{\mathcal{H}}(\widehat{e}{:}A_i)$.*

Proof. First note that the results follow immediately if $\widehat{e}$ is not in *whnf* or is a λ-abstraction (because potentials are zero in those cases). The potential is also zero if $\widehat{e}$ is a constructor that is part of a cycle (since otherwise it would be undefined). The remaining case is for a constructor with no cycles, i.e. a directed acyclic graph (DAG). The proof is then by induction on the length of the longest path. We have $\curlyvee(A \,|\, A_1, \ldots, A_n\,)$ and $\widehat{e} \equiv c\ \vec{y}$. Also $\phi_{\mathcal{H}}(c\ \vec{y}{:}A)$ is defined.

If $A = \mathsf{T}^q(B)$ then $A_i = \mathsf{T}^{q_i}(B_i)$ for $(1 \leq i \leq n)$ and we would proceed to proving $\phi_{\mathcal{H}}(c\ \vec{y}{:}B) \geq \sum_i \phi_{\mathcal{H}}(c\ \vec{y}{:}B_i)$.

Otherwise, $A = \mu X.\{\cdots \mid c{:}(p, \vec{B}) \mid \cdots\}$ and $A_i = \mu X.\{\cdots \mid c{:}(p_i, \vec{B}'_i) \mid \cdots\}$ for $(1 \leq i \leq n)$. We have to prove $\phi_{\mathcal{H}}(c\ \vec{y}{:}A) \geq \sum_i \phi_{\mathcal{H}}(c\ \vec{y}{:}A_i)$ or in this case the equivalent inequality

$$p + \sum_j \phi_{\mathcal{H}}(\mathcal{H}(\ell_j){:}B_j[A/X]) \geq \sum_i (p_i + \sum_j \phi_{\mathcal{H}}\big(\mathcal{H}(\ell_j){:}B'_{ij}[A_i/X]\big))$$

By induction on the shorter paths $\mathcal{H}(\ell_j)$, we know

$$\sum_j \phi_{\mathcal{H}}(\mathcal{H}(\ell_j){:}B_j[A/X]) \geq \sum_i \sum_j \phi_{\mathcal{H}}\big(\mathcal{H}(\ell_j){:}B'_{ij}[A_i/X]\big)$$

From the non-negativity of potential annotations, all that remains to prove is

$$p \geq \sum_i p_i$$

and that follows from $\curlyvee(A \,|\, A_1, \ldots, A_n\,)$ by the definition of sharing.

This concludes the proof of Potential Splitting.

□

5.5.6.6 Idempotent Cycles

Definition 5.16 (Reachability)**.** The *one-step reachability* relation $\ell \rightsquigarrow_{\mathcal{H}} \ell'$ between two locations ℓ, ℓ' in a heap $\mathcal{H}$ holds if and only if $\mathcal{H}(\ell) = c\,\vec{\ell''}$ and $\ell' \in \vec{\ell''}$. The *many-step reachability* relation $\rightsquigarrow^{+}_{\mathcal{H}}$ is defined as the transitive closure of the one-step reachability relation.

Note that reachability only traverses constructors, but not unevaluated locations nor λ-abstractions. This mimics the definition of potential (Definition 5.6).

The following lemma shows that, in a consistent configuration, locations within cycles can be assigned global types with zero potential. Because of the way the invariants were defined, any cycles having positive potential must keep this potential within the cycle in order to justify the typing of each subsequent location. Therefore, since this potential cannot affect the types of locations outside the cycle, we can always set the potential within a cycle to zero.

Lemma 5.17 (Idempotent Cycles)**.** *Let* $(\mathcal{H}, \mathcal{L})$ *be a heap configuration consistent with global types, contexts and balance* $\mathcal{M}, \mathcal{C}, \mathcal{B}$*, that is, such that* $\mathcal{C}, \mathcal{B} \vdash_{\mathsf{MEM}} (\mathcal{H}, \mathcal{L}) : \mathcal{M}$ *and* $\curlyvee(\mathcal{M} \,|\, \Gamma, \mathcal{C})$*. Then there exist* $\mathcal{C}', \mathcal{M}'$ *such that* $\mathcal{M} <: \mathcal{M}'$ *with* $\mathcal{C}', \mathcal{B} \vdash_{\mathsf{MEM}} (\mathcal{H}, \mathcal{L}) : \mathcal{M}'$ *and* $\curlyvee(\mathcal{M}' \,|\, \Gamma, \mathcal{C}')$ *such that for all* ℓ *with* $\ell \rightsquigarrow^{+}_{\mathcal{H}} \ell$ *we have* $\curlyvee(\mathcal{M}'(\ell) \,|\, \mathcal{M}'(\ell), \mathcal{M}'(\ell))$ *as well.*

Proof. Consider a cycle consisting of the locations $\ell_0, \ldots, \ell_{n+1}$ with $\ell_i \rightsquigarrow_{\mathcal{H}} \ell_{i+1}$ and $\ell_{n+1} = \ell_0$. By Definition 5.16 (Reachability) each $\mathcal{H}(\ell_i)$ must be a constructor of the form $c_i(\ldots, \ell_{i+1}, \ldots)$. The type consistency of locations (Definition 5.10) for each ℓ_i must hold by case LOC1, because constructors are *whnfs*. Since thunk annotations are irrelevant for LOC1, we omit them in the following for readability.

Let $\mathcal{M}(\ell_i) = \mathsf{T}(A_i)$, hence $\mathcal{C}(\ell_i) \overset{0}{\vdash} c_i(\ldots, \ell_{i+1}, \ldots) : A_i$ by LOC1. By our assumption that recursive types are non-interleaving, the type for the position of ℓ_{i+1} within the constructor c_i must be the μ-bound type variable, i.e. $A_i = \mu X.\{\cdots | c_i : (p_i, \ldots \mathsf{T}(X) \ldots) | \cdots\}$. Applying Lemma 5.3 (CONS Inversion) we obtain

$$\curlyvee(\mathcal{C}(\ell_i) \,|\, \ell_{i+1}{:}\mathsf{T}(A_i), \ldots) \tag{5.17}$$

From (5.17) by the definition of context sharing and subtyping we conclude that there exists A'_i such that $\mathsf{T}(A'_i) \in \mathcal{C}{\upharpoonright}_{\ell_{i+1}}$ and $A'_i <: A_i$. By the definition of global compatibility (Definition 5.12), we have $\curlyvee\big(\mathcal{M}(\ell_{i+1}) \;\big|\; \Gamma{\upharpoonright}_{\ell_{i+1}}, \mathcal{C}{\upharpoonright}_{\ell_{i+1}}\big)$; again by definition of subtyping this implies $A_{i+1} <: A'_i$; combining with $A'_i <: A_i$ established earlier, we obtain

$$A_{i+1} <: A'_i <: A_i \quad \text{for } 0 \leq i \leq n \tag{5.18}$$

Because $<:$ is a partial order (Lemma 5.14) and $A_{n+1} = A_0$ by definition, it follows from (5.18) that the A_i, A'_i must all be equal. Let A be this common type of the cycle locations, i.e. $\mathcal{M}(\ell_i) = \mathsf{T}(A)$ for all $0 \leq i \leq n$. The compatibility hypothesis for location ℓ_i now instantiates as follows:

$$\curlyvee\big(\mathsf{T}(A) \;\big|\; \Gamma{\upharpoonright}_{\ell_i}, \mathsf{T}(A), \ldots\big) \tag{5.19}$$

Because each location occurs at least once in the cycle with exactly the global type $\mathsf{T}(A)$ we know that any other references in Γ or $\mathcal{C}$ must occur with an idempotent subtype of A, i.e. A' such that $A <: A'$ and $\curlyvee(A' \mid A', A')$. We can thus set the global type for all locations in the cycle to this self-sharing type A' without disrupting type consistency. □

5.5.6.7 Proof of the Soundness Theorem

The proof of Theorem 5.13 follows by induction on the lengths of the derivations of (5.6) and (5.5) 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 unevaluated locations, which prolongs the length of the typing derivation by a typing judgement for the thunk, granted through the type consistency hypothesis. On the other hand, the length of the derivation for the term evaluation never increases, but may remain unchanged where the last step of the typing derivation was obtained by a structural rule. In these cases, the length of the typing derivation does decrease, allowing an induction over the lexicographically ordered lengths of both derivations. We proceed by case analysis of the typing rule used in premise (5.5).

Case VAR**:** We have $\ell{:}\mathsf{T}^q(A) \overset{q}{\vdash} \ell : A$ from the typing hypothesis (5.5). From the compatibility hypothesis (5.8) we then obtain $\curlyvee\big(\mathcal{M}(\ell) \;\big|\; \mathsf{T}^q(A), \Theta{\upharpoonright}_\ell, \mathcal{C}{\upharpoonright}_\ell\big)$ which implies $\mathcal{M}(\ell) = \mathsf{T}^{q'}(\widehat{A})$ and $\curlyvee(\widehat{A} \mid A, \bar{A})$ for some types $\widehat{A}, \bar{A}$ and annotation q' with $q \geq q'$.

The evaluation premise (5.6) reads as $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \ell \Downarrow w, \mathcal{H}'[\ell \mapsto w]$ for some intermediate heap $\mathcal{H}'$; by inversion of the only applicable evaluation rule $\textsc{Var}_{\Downarrow}$, we obtain $\ell \notin \mathcal{L}$ and

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash \mathcal{H}(\ell) \Downarrow w, \mathcal{H}' \tag{5.20}$$

By (5.7) for location ℓ we obtain $\mathcal{C}(\ell), \mathcal{B}; \mathcal{H}, \mathcal{L} \vdash_{\text{LOC}} \ell : \mathsf{T}^{q'}(\widehat{A})$. We proceed by case analysis of the rule used for type consistency of ℓ. Note that LOC3 cannot apply because $\ell \notin \mathcal{L}$ by the premise of the VAR rule. The remaining cases are then LOC1 and LOC2, which apply according to whether $\mathcal{H}(\ell)$ is in *whnf* or not, respectively.

If $\mathcal{H}(\ell)$ is in *whnf*: The evaluation (5.20) terminates immediately by $\textsc{Whnf}_{\Downarrow}$ and we have $w = \mathcal{H}(\ell)$ and $\mathcal{H} = \mathcal{H}' = \mathcal{H}'[\ell \mapsto w]$, i.e. the update is without effect. By LOC1 we obtain $\mathcal{C}(\ell) \overset{0}{\vdash} w : \widehat{A}$. By $\curlyvee(\widehat{A} \,|\, A, \bar{A})$ established earlier and Lemma 5.5 we obtain $\curlyvee(\mathcal{C}(\ell) \,|\, \Gamma_1', \Gamma_2')$ and $\Gamma_1' \overset{0}{\vdash} w{:}A$ as required for (5.10), as well as $\Gamma_2' \overset{0}{\vdash} w{:}\bar{A}$. Let $\mathcal{M}' = \mathcal{M}[\ell \mapsto \mathsf{T}^0(\bar{A})]$ and $\mathcal{C}' = \mathcal{C}[\ell \mapsto \Gamma_2']$. By the previous results together with (5.7) we obtain $\mathcal{C}', \mathcal{B} \vdash_{\text{MEM}} (\mathcal{H}, \mathcal{L}) : \mathcal{M}'$ as required for (5.13). From the compatibility premise (5.8) together with $\curlyvee(\mathcal{C}(\ell) \,|\, \Gamma_1', \Gamma_2')$ established earlier we can conclude $\curlyvee(\mathcal{M}' \,|\, \Gamma_1', \Theta, \mathcal{C}')$ as required for (5.14). Conclusions (5.11) and (5.16) with $m' = m$ follow directly from an application of rule $\textsc{Whnf}_{\Downarrow\mathsf{C}}$. It remains to show that the bound (5.15) is satisfied for the choice $m' = m$. Our proof obligation is:

$$t + q + \phi_{\mathcal{H}}(\ell{:}\mathsf{T}^q(A)) + \phi_{\mathcal{H}}(\Theta) + \Phi_{\mathcal{H}}^{\mathcal{L}}(\mathcal{C}) + \Phi_{\mathcal{H}}^{\mathcal{L}}(\mathcal{B}) \;\geq t + \phi_{\mathcal{H}}(w{:}A) + \phi_{\mathcal{H}}(\Theta) + \Phi_{\mathcal{H}}^{\mathcal{L}}(\mathcal{C}') + \Phi_{\mathcal{H}}^{\mathcal{L}}(\mathcal{B})$$

The above inequality holds because: q is non-negative; $\phi_{\mathcal{H}}(\ell{:}\mathsf{T}^q(A)) = \phi_{\mathcal{H}}(\mathcal{H}(\ell){:}A) = \phi_{\mathcal{H}}(w{:}A)$ by Def. 5.6 (potential); and $\Phi_{\mathcal{H}}^{\mathcal{L}}(\mathcal{C}') = \Phi_{\mathcal{H}}^{\mathcal{L}}(\mathcal{C})$ because $\mathcal{C}'$ differs from $\mathcal{C}$ only for ℓ which is in *whnf*, and therefore its context does not contribute to the potential.

If $\mathcal{H}(\ell)$ is not in *whnf*: By LOC2 we obtain $\mathcal{C}(\ell) \overset{q' + \mathcal{B}(\ell)}{\vdash\!\!-\!\!-\!\!-} \mathcal{H}(\ell) : \widehat{A}$. Let $\mathcal{M}' = \mathcal{M}[\ell \mapsto \mathsf{T}^{q'}(\bar{A})]$ and $\mathcal{C}' = \mathcal{C}[\ell \mapsto \emptyset]$. We observe that $\mathcal{C}', \mathcal{B} \vdash_{\text{MEM}} (\mathcal{H}, \mathcal{L} \cup \{\ell\}) : \mathcal{M}'$ must hold by hypothesis (5.7) together with the case for LOC3 for location ℓ. Furthermore $\curlyvee(\mathcal{M}' \,|\, \mathcal{C}(\ell), \Theta, \mathcal{C}')$ holds, since for all ℓ' we have $\mathcal{C}{\restriction_{\ell'}} = \mathcal{C}(\ell){\restriction_{\ell'}} \cup \mathcal{C}'{\restriction_{\ell'}}$ by definition.

We will now apply the induction hypothesis to the evaluation of $\mathcal{H}(\ell)$ with type $\widehat{A}$. We first show that m can be chosen as required for the induction; the proof obligation is:

$$m \geq t + (q' + \mathcal{B}(\ell)) + \phi_{\mathcal{H}}(\mathcal{C}(\ell)) + \phi_{\mathcal{H}}(\Theta) + \Phi_{\mathcal{H}}^{\mathcal{L}\cup\{\ell\}}(\mathcal{C}') + \Phi_{\mathcal{H}}^{\mathcal{L}\cup\{\ell\}}(\mathcal{B}) \tag{5.21}$$

Starting from the hypothesis (5.9) we obtain:

$$\begin{aligned} m &\geq t + q + \phi_{\mathcal{H}}(\ell{:}\mathsf{T}^q(A)) + \phi_{\mathcal{H}}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{C}) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B}) \\ &\geq t + q' + 0 + \big(\phi_{\mathcal{H}}(\mathcal{C}(\ell)) + \phi_{\mathcal{H}}(\Theta) + \Phi^{\mathcal{L}\cup\{\ell\}}_{\mathcal{H}}(\mathcal{C}')\big) + \big(\Phi^{\mathcal{L}\cup\{\ell\}}_{\mathcal{H}}(\mathcal{B}) + \mathcal{B}(\ell)\big) \\ &= t + \big(q' + \mathcal{B}(\ell)\big) + \phi_{\mathcal{H}}(\mathcal{C}(\ell)) + \phi_{\mathcal{H}}(\Theta) + \Phi^{\mathcal{L}\cup\{\ell\}}_{\mathcal{H}}(\mathcal{C}') + \Phi^{\mathcal{L}\cup\{\ell\}}_{\mathcal{H}}(\mathcal{B}) \end{aligned}$$

The inequality follows, since $q' \leq q$ from above; $\phi_{\mathcal{H}}(\ell{:}\mathsf{T}^q(A)) = 0$ because $\mathcal{H}(\ell)$ is not in *whnf*; $\Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{C}) = \phi_{\mathcal{H}}(\mathcal{C}(\ell)) + \Phi^{\mathcal{L}\cup\{\ell\}}_{\mathcal{H}}(\mathcal{C}')$ by definition; and $\Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B}) = \mathcal{B}(\ell) + \Phi^{\mathcal{L}\cup\{\ell\}}_{\mathcal{H}}(\mathcal{B})$ by definition.

We can now apply the induction hypothesis to the evaluation of $\mathcal{H}(\ell)$ with type $\widehat{A}$ and obtain:

$$\Gamma' \overset{0}{\vdash} w : \widehat{A} \tag{5.22}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{\ell\} \overset{m''}{\vdash} \mathcal{H}(\ell) \Downarrow w, \mathcal{H}' \tag{5.23}$$

$$\mathcal{M}' <: \mathcal{M}'' \tag{5.24}$$

$$\mathcal{C}'', \mathcal{B}' \vdash_{\mathsf{MEM}} (\mathcal{H}', \mathcal{L} \cup \{\ell\}) : \mathcal{M}'' \tag{5.25}$$

$$\curlyvee(\mathcal{M}'' \mid \Gamma', \Theta, \mathcal{C}'') \tag{5.26}$$

$$m' \geq t + \phi_{\mathcal{H}'}(w{:}\widehat{A}) + \phi_{\mathcal{H}'}(\Theta) + \Phi^{\mathcal{L}\cup\{\ell\}}_{\mathcal{H}'}(\mathcal{C}'') + \Phi^{\mathcal{L}\cup\{\ell\}}_{\mathcal{H}'}(\mathcal{B}') \tag{5.27}$$

$$m - m' \geq m'' \tag{5.28}$$

By applying the induction hypothesis to the global type, we obtained a stronger typing (5.22) for the resulting *whnf* as well. We now recover the required typing for A by the lemma for splitting contexts and the remaining potential associated through $\bar{A}$ allows us to establish memory consistency for the remaining aliases. So by $\curlyvee(\widehat{A} \mid A, \bar{A})$ from above, (5.22) and Lemma 5.5 we obtain $\curlyvee(\Gamma' \mid \Gamma'_1, \Gamma'_2)$ and $\Gamma'_1 \overset{0}{\vdash} w{:}A$ as required for (5.10), as well as $\Gamma'_2 \overset{0}{\vdash} w{:}\bar{A}$; this together with (5.25) and the case Loc1 of Def. 5.10 gives us $\mathcal{C}''[\ell \mapsto \Gamma'_2], \mathcal{B}' \vdash_{\mathsf{MEM}} (\mathcal{H}'[\ell \mapsto w], \mathcal{L}) : \mathcal{M}''$ as required for (5.13).

Conclusion (5.12) follows by the transitivity of subtyping from $\mathcal{M} <: \mathcal{M}'$ by the definition of $\mathcal{M}'$ and (5.24).

From (5.26) we have $\curlyvee(\mathcal{M}'' \mid \Gamma'_1, \Gamma'_2, \Theta, \mathcal{C}'')$ which by definition is equivalent to

$$\curlyvee(\mathcal{M}'' \mid \Gamma'_1, \Theta, \mathcal{C}''[\ell \mapsto \Gamma'_2])$$

as required for (5.14).

Conclusion (5.11) follows directly from (5.23) by application of $\text{VAR}_{\Downarrow}$.

It remains to show that m' obtained from (5.27) satisfies the requirements of (5.15); our proof obligation is:

$$t + \phi_{\mathcal{H}'}(w{:}\widehat{A}) + \phi_{\mathcal{H}'}(\Theta) + \Phi_{\mathcal{H}'}^{\mathcal{L}\cup\{\ell\}}(\mathcal{C}'') + \Phi_{\mathcal{H}'}^{\mathcal{L}\cup\{\ell\}}(\mathcal{B}') \geq \\ t + \phi_{\mathcal{H}'[\ell\mapsto w]}(w{:}A) + \phi_{\mathcal{H}'[\ell\mapsto w]}(\Theta) + \Phi_{\mathcal{H}'[\ell\mapsto w]}^{\mathcal{L}}(\mathcal{C}''[\ell \mapsto \Gamma_2']) + \Phi_{\mathcal{H}'[\ell\mapsto w]}^{\mathcal{L}}(\mathcal{B}')$$

First note that $\Phi_{\mathcal{H}'}^{\mathcal{L}\cup\{\ell\}}(\mathcal{B}') = \Phi_{\mathcal{H}'[\ell\mapsto w]}^{\mathcal{L}}(\mathcal{B}')$ since the balance ignores both *whnfs* and locations under evaluation. It remains to show that

$$\phi_{\mathcal{H}'}(w{:}\widehat{A}) + \phi_{\mathcal{H}'}(\Theta) + \Phi_{\mathcal{H}'}^{\mathcal{L}\cup\{\ell\}}(\mathcal{C}'') \geq \phi_{\mathcal{H}'[\ell\mapsto w]}(w{:}A) + \phi_{\mathcal{H}'[\ell\mapsto w]}(\Theta) + \Phi_{\mathcal{H}'[\ell\mapsto w]}^{\mathcal{L}}(\mathcal{C}''[\ell \mapsto \Gamma_2']) \tag{5.29}$$

We first argue that we can assume without loss of generality that the potentials above are all defined (i.e. finite): these could be undefined only if the update $\mathcal{H}'[\ell \mapsto w]$ introduced new cycles, but in that case we would apply Lemma 5.17 (Idempotent Cycles) and obtain new global contexts and global types that still satisfy the three conclusions (5.12), (5.13) and (5.14) proved earlier. Furthermore, any new cycles must include the updated location ℓ, for which the refined global type assigns zero potential by Lemma 5.5. That implies (5.29) is then an equality, since $\phi_{\mathcal{H}'}(w{:}\widehat{A})$, $\phi_{\mathcal{H}'[\ell\mapsto w]}(w{:}A)$ and $\Phi_{\mathcal{H}'[\ell\mapsto w]}^{\mathcal{L}}((\mathcal{C}''[\ell \mapsto \Gamma_2'])\restriction_{\ell})$ must then all be zero, and likewise $\Phi_{\mathcal{H}'}^{\mathcal{L}\cup\{\ell\}}(\mathcal{C}''\restriction_{\ell})$ by definition.

In the remaining case where the update did not introduce a cycle, we have $\phi_{\mathcal{H}'}(w{:}\widehat{A}) = \phi_{\mathcal{H}'[\ell\mapsto w]}(w{:}\widehat{A})$ and $\phi_{\mathcal{H}'}(\Theta) = \phi_{\mathcal{H}'[\ell\mapsto w]}(\Theta)$. Recall that $\curlyvee(\widehat{A}\,|\,A, \bar{A})$, so by Lemma 5.7 we have $\phi_{\mathcal{H}'[\ell\mapsto w]}(w{:}\widehat{A}) \geq \phi_{\mathcal{H}'[\ell\mapsto w]}(w{:}A) + \phi_{\mathcal{H}'[\ell\mapsto w]}(w{:}\bar{A})$. Thus it remains to be shown that

$$\phi_{\mathcal{H}'}(w{:}\bar{A}) \;\geq\; \Phi_{\mathcal{H}'[\ell\mapsto w]}^{\mathcal{L}}(\mathcal{C}''[\ell \mapsto \Gamma_2']) \;-\; \Phi_{\mathcal{H}'}^{\mathcal{L}\cup\{\ell\}}(\mathcal{C}'') \;=\; \Phi_{\mathcal{H}'[\ell\mapsto w]}^{\mathcal{L}}((\mathcal{C}''[\ell \mapsto \Gamma_2'])\restriction_{\ell}) \tag{5.30}$$

which follows by the compatibility concluded earlier, and applying Lemma 5.9 for

$$\mathsf{T}^{q'}(\bar{A}) <: \mathcal{M}''(\ell)$$

This concludes the proof of the VAR case.

Case LET**:** The typing and evaluation premises (5.5) and (5.6) instantiate as

$$\Gamma, \Delta \overset{1+q+p}{\vdash} \mathsf{let}\ x = \widehat{e}\ \mathsf{in}\ e : C$$

and $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \mathsf{let}\ x = \widehat{e}\ \mathsf{in}\ e \Downarrow w, \mathcal{H}'$, respectively. By inversion of rules LET and LET$_\Downarrow$ we obtain

$$x \notin \operatorname{dom}(\Gamma, \Delta) \tag{5.31}$$

$$\curlyvee(A \mid A, A') \tag{5.32}$$

$$\Gamma, x{:}\mathsf{T}^{q'}(A') \overset{q'}{\vdash} \widehat{e} : A \tag{5.33}$$

$$\Delta, x{:}\mathsf{T}^{q'}(A) \overset{q}{\vdash} e : C \tag{5.34}$$

$$p = \begin{cases} p', & \text{if } \widehat{e} \equiv c\ \vec{y} \text{ and } A = \mu X.\{\cdots \mid c : (p', \vec{B}) \mid \cdots\} \\ 0, & \text{otherwise} \end{cases} \tag{5.35}$$

$$\ell \text{ is fresh} \tag{5.36}$$

$$\mathcal{H}[\ell \mapsto \widehat{e}[\ell/x]], \mathcal{S}, \mathcal{L} \vdash e[\ell/x] \Downarrow w, \mathcal{H}' \tag{5.37}$$

Applying Lemma 5.2 (substitution) to (5.33) and (5.34) we obtain

$$\Gamma, \ell{:}\mathsf{T}^{q'}(A') \overset{q'}{\vdash} \widehat{e}[\ell/x] : A \tag{5.38}$$

$$\Delta, \ell{:}\mathsf{T}^{q'}(A) \overset{q}{\vdash} e[\ell/x] : C \tag{5.39}$$

We intend to apply the induction hypothesis over subterm $e[\ell/x]$ using (5.39) and (5.37), so we must establish the required premises first. Note that we do not invoke the induction hypothesis for subterm $\widehat{e}$, since it is not executed at this point, but just stored within the heap.

Let $\mathcal{H}_0 = \mathcal{H}\big[\ell \mapsto \widehat{e}[\ell/x]\big]$. In order to establish global compatibility and type consistency for the extended memory $\mathcal{H}_0$, we set $\mathcal{B}_0 = \mathcal{B}[\ell \mapsto 0]$, $\mathcal{M}_0 = \mathcal{M}[\ell \mapsto \mathsf{T}^{q'}(A)]$ and $\mathcal{C}_0 = \mathcal{C}[\ell \mapsto \Gamma, \ell{:}\mathsf{T}^{q'}(A')]$. Type consistency for existing locations is unaffected by these extensions, since ℓ is a fresh location.

The required global compatibility $\curlyvee\big(\mathcal{M}_0 \bigm| \Delta, \ell{:}\mathsf{T}^{q'}(A), \Theta, \mathcal{C}_0\big)$ follows from (5.8) and $\curlyvee\big(\mathsf{T}^{q'}(A) \bigm| \mathsf{T}^{q'}(A), \mathsf{T}^{q'}(A')\big)$, where the latter follows from typing premise (5.32).

Type consistency for the new location ℓ requires $\Gamma, \ell{:}\mathsf{T}^{q'}(A'), \mathcal{B}_0; \mathcal{H}_0, \mathcal{L} \vdash_{\mathsf{LOC}} \ell : \mathsf{T}^{q'}(A)$ to hold. We now distinguish whether $\widehat{e}[\ell/x]$ happens to be in *whnf* or not. In the case that

$\widehat{e}[\ell/x]$ is not in *whnf*, (LOC2) applies, since a fresh location is not contained in $\mathcal{L}$ and the required typing (5.38) holds.

In the case that $\widehat{e}[\ell/x]$ is in *whnf*, (LOC1) applies since there is a type judgement for expression $\widehat{e}[\ell/x]$ with zero on the turnstile as required by (LOC1), either by inversion of ABS (Lemma 5.4) or CONS (Lemma 5.3), followed by an immediate application of ABS or CONS, depending on whether the *whnf* is a λ-expression or a constructor application, respectively. (Note that instead of $\Gamma, \ell{:}\mathsf{T}^{q'}(A')$, inversion might require an altered context. If this is the case, then $\mathcal{C}_0$ is chosen above to deliver the altered context in the first place.)

This establishes the required type consistency for ℓ and thus together with (5.7) also $\mathcal{C}_0, \mathcal{B}_0 \vdash_{\mathsf{MEM}} (\mathcal{H}_0, \mathcal{L}) : \mathcal{M}_0$.

In order to establish the remaining premise (5.9), we proceed by case analysis on whether expression $\widehat{e}[\ell/x]$ is a constructor application (and consequently on whether we need to consider the potential p').

If $\widehat{e} \equiv c\,\vec{y}$: Premise (5.9) reads as

$$m + 1 \geq t + 1 + q + p' + \phi_{\mathcal{H}}(\Gamma, \Delta) + \phi_{\mathcal{H}}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{C}) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B})$$

By definition $\phi_{\mathcal{H}_0}\Big(\ell{:}\mathsf{T}^{q'}(A)\Big) \leq p' + \phi_{\mathcal{H}_0}\Big(\Gamma, \ell{:}\mathsf{T}^{q'}(A')\Big) = p' + \phi_{\mathcal{H}}(\Gamma)$, where the inequality is due to the context possibly containing unneeded or needlessly strong references and the equality follows by $\ell \notin \mathrm{dom}(\Gamma)$ from the freshness of ℓ and type A' being idempotent by $\curlyvee(A \,|\, A, A')$ and Lemma 5.15; $\Phi^{\mathcal{L}}_{\mathcal{H}_0}(\mathcal{C}_0) = \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{C})$ since ℓ points to a *whnf*; furthermore $\Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B}) = \Phi^{\mathcal{L}}_{\mathcal{H}_0}(\mathcal{B}_0)$ by definition. Combining these three with the previous inequality yields as required

$$m \geq t + q + \phi_{\mathcal{H}_0}\big(\Delta, \ell{:}\mathsf{T}^{q'}(A)\big) + \phi_{\mathcal{H}_0}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}_0}(\mathcal{C}_0) + \Phi^{\mathcal{L}}_{\mathcal{H}_0}(\mathcal{B}_0)$$

since the other statements are unaffected by the fresh ℓ extending $\mathcal{H}$ to $\mathcal{H}_0$.

If $\widehat{e} \not\equiv c\,\vec{y}$: Premise (5.9) reads as

$$m + 1 \geq t + 1 + q + 0 + \phi_{\mathcal{H}}(\Gamma, \Delta) + \phi_{\mathcal{H}}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{C}) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B})$$

Since $\widehat{e} \not\equiv c\ \vec{y}$, $\mathcal{H}_0(\ell)$ is either a λ-expression or not in *whnf*, hence $\phi_{\mathcal{H}_0}\big(\ell{:}\mathsf{T}^{q'}(A)\big) = 0$; by subtyping $\phi_{\mathcal{H}_0}\big(\ell{:}\mathsf{T}^{q'}(A')\big) = 0$ and thus $\phi_{\mathcal{H}}(\Gamma) = \phi_{\mathcal{H}_0}(\mathcal{C}_0(\ell))$ by definition of $\mathcal{C}_0$, and hence $\Phi^{\mathcal{L}}_{\mathcal{H}_0}(\mathcal{C}_0) \leq \phi_{\mathcal{H}}(\Gamma) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{C})$ (note that this is an equality if $\mathcal{H}_0(\ell)$ is not in *whnf*, and Γ is minimal, and strict inequality if $\mathcal{H}_0(\ell)$ is a λ-expression and $\phi_{\mathcal{H}}(\Gamma) > 0$); furthermore $\Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B}) = \Phi^{\mathcal{L}}_{\mathcal{H}_0}(\mathcal{B}_0)$ by definition. Combining these three with the inequality before yields as required

$$m \geq t + q + \phi_{\mathcal{H}_0}\big(\Delta, \ell{:}\mathsf{T}^{q'}(A)\big) + \phi_{\mathcal{H}_0}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}_0}(\mathcal{C}_0) + \Phi^{\mathcal{L}}_{\mathcal{H}_0}(\mathcal{B}_0)$$

since the other statements are unaffected by the fresh ℓ extending $\mathcal{H}$ to $\mathcal{H}_0$.

Regardless of whether expression $\widehat{e}[\ell/x]$ is a constructor application, once premise (5.9) is established, applying the induction hypothesis then yields all required conclusions directly without any alterations, except for (5.12) which follows by the transitivity of subtyping and $\mathcal{M} <: \mathcal{M}_0$ by definition. This concludes the proof of the LET case.

Case ABS**:** The typing premise (5.5) is $\Gamma \overset{0}{\vdash} \lambda x.e : A \xrightarrow{q} C$.

The evaluation premise (5.6) is $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \lambda x.e \Downarrow \lambda x.e, \mathcal{H}$. Assume m satisfying (5.9); let $\Gamma' = \Gamma$, $\mathcal{C}' = \mathcal{C}$, $\mathcal{M}' = \mathcal{M}$, $\mathcal{B}' = \mathcal{B}$, $m' = m$ and $m'' = 0$; we trivially obtain (5.12), (5.10), (5.13), (5.14), (5.11), (5.16).

It remains to show that the bound (5.15) is satisfied when $m' = m$. From the premise (5.9) we know

$$m \geq t + \phi_{\mathcal{H}}(\Gamma) + \phi_{\mathcal{H}}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{C}) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B}) \tag{5.40}$$

By inversion of rule ABS we obtain $\curlyvee(\Gamma \,|\, \Gamma, \Gamma)$ which by Lemma 5.7 implies $\phi_{\mathcal{H}}(\Gamma) = 0$. By Def. 5.6 (potential) we also obtain $\phi_{\mathcal{H}}\big(\lambda x.e{:}A \xrightarrow{q} C\big) = 0$, and therefore also $\phi_{\mathcal{H}}(\Gamma) = \phi_{\mathcal{H}}\big(\lambda x.e{:}A \xrightarrow{q} C\big)$; substituting in (5.40) gives us

$$m' \geq t + \phi_{\mathcal{H}}\big(\lambda x.e{:}A \xrightarrow{q} C\big) + \phi_{\mathcal{H}}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{C}) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B})$$

as required. This concludes the proof of the ABS case.

Case APP**:** The typing and evaluation premises (5.5) and (5.6) instantiate as

$$\Gamma, \ell{:}A \overset{q + q'}{\vdash} e\ \ell : C$$

and $\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash e\ \ell \Downarrow w, \mathcal{H}''$, respectively. By inversion of rules App and App$_{\Downarrow}$ we obtain

$$\Gamma \overset{q}{\vdash} e : A \xrightarrow{q'} C \tag{5.41}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash e \Downarrow \lambda x.e', \mathcal{H}' \tag{5.42}$$

$$\mathcal{H}', \mathcal{S}, \mathcal{L} \vdash e'[\ell/x] \Downarrow w, \mathcal{H}'' \tag{5.43}$$

By premise (5.9) we assume

$$\begin{aligned} m &\geq t + q + q' + \phi_{\mathcal{H}}(\Gamma, \ell{:}A) + \phi_{\mathcal{H}}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{C}) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B}) \\ &= (t + q') + q + \phi_{\mathcal{H}}(\Gamma) + \phi_{\mathcal{H}}(\ell{:}A, \Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{C}) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B}) \end{aligned} \tag{5.44}$$

Inequation (5.44) shows that the bound for m satisfies the requirements for applying induction for expression e using judgements (5.41) and (5.42); we obtain $m', \Gamma', \mathcal{C}', \mathcal{B}', \mathcal{M}'$ and m''_1 such that:

$$\Gamma' \overset{0}{\vdash} \lambda x.e' : A \xrightarrow{q'} C \tag{5.45}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{m''_1}{\vdash} e \Downarrow \lambda x.e', \mathcal{H}' \tag{5.46}$$

$$\mathcal{M} <: \mathcal{M}' \tag{5.47}$$

$$\mathcal{C}', \mathcal{B}' \vdash_{\text{MEM}} (\mathcal{H}', \mathcal{L}) : \mathcal{M}' \tag{5.48}$$

$$\curlyvee(\mathcal{M}' \,|\, (\Gamma',\ \ell{:}A,\ \Theta, \mathcal{C}') \tag{5.49}$$

$$m' \geq t + q' + \phi_{\mathcal{H}'}\Big(\lambda x.e'{:}A \xrightarrow{q'} C\Big) + \phi_{\mathcal{H}'}(\ell{:}A, \Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}'}(\mathcal{C}') + \Phi^{\mathcal{L}}_{\mathcal{H}'}(\mathcal{B}') \tag{5.50}$$

$$m - m' \geq m''_1 \tag{5.51}$$

By Lemma 5.4 (Abs inversion) applied to judgement (5.45) we can assume without loss of generality that $\curlyvee(\Gamma' \,|\, \Gamma', \Gamma')$ and $\Gamma', x{:}A \overset{q'}{\vdash} e' : C$; applying Lemma 5.2 (Substitution) we obtain

$$\Gamma', \ell{:}A \overset{q'}{\vdash} e'[\ell/x] : C \tag{5.52}$$

In order to apply the induction hypothesis to $e'[\ell/x]$ it remains to show that the bound (5.50) for m' satisfies the premise (5.9). By $\curlyvee(\Gamma' \,|\, \Gamma', \Gamma')$ established earlier and Lemma 5.7 (Potential Splitting) we know $\phi_{\mathcal{H}'}(\Gamma') = 0$ and therefore $\phi_{\mathcal{H}'}(\Gamma', \ell{:}A) = \phi_{\mathcal{H}'}(\ell{:}A)$.

By Def. 5.6 (Potential) we know $\phi_{\mathcal{H}'}\Big(\lambda x.e'{:}A \xrightarrow{q'} C\Big) = 0$; substituting in (5.50) gives us:

$$\begin{aligned} m' &\geq t + q' + \phi_{\mathcal{H}'}\Big(\lambda x.e'{:}A \xrightarrow{q'} C\Big) + \phi_{\mathcal{H}'}(\ell{:}A, \Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}'}\big(\mathcal{C}'\big) + \Phi^{\mathcal{L}}_{\mathcal{H}'}\big(\mathcal{B}'\big) \\ &= t + q' + \phi_{\mathcal{H}'}(\ell{:}A, \Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}'}\big(\mathcal{C}'\big) + \Phi^{\mathcal{L}}_{\mathcal{H}'}\big(\mathcal{B}'\big) \\ &= t + q' + \phi_{\mathcal{H}'}\big(\Gamma', \ell{:}A\big) + \phi_{\mathcal{H}'}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}'}\big(\mathcal{C}'\big) + \Phi^{\mathcal{L}}_{\mathcal{H}'}\big(\mathcal{B}'\big) \end{aligned}$$

Hence we are able to apply induction on $e'[\ell/x]$ and obtain:

$$\Gamma'' \overset{0}{\vdash} w : C \tag{5.53}$$

$$\mathcal{H}', \mathcal{S}, \mathcal{L} \overset{m''_2}{\vdash} e'[\ell/x] \Downarrow w, \mathcal{H}'' \tag{5.54}$$

$$\mathcal{M}' <: \mathcal{M}'' \tag{5.55}$$

$$\mathcal{C}'', \mathcal{B}'' \vdash_{\textsf{MEM}} (\mathcal{H}'', \mathcal{L}) : \mathcal{M}'' \tag{5.56}$$

$$\curlyvee(\mathcal{M}'' \,|\, (\Gamma'', \Theta), \mathcal{C}'') \tag{5.57}$$

$$m'' \geq t + \phi_{\mathcal{H}''}(w{:}C) + \phi_{\mathcal{H}''}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}''}\big(\mathcal{C}''\big) + \Phi^{\mathcal{L}}_{\mathcal{H}''}\big(\mathcal{B}''\big) \tag{5.58}$$

$$m' - m'' \geq m''_2 \tag{5.59}$$

From (5.47) and (5.55) and the transitivity of subtyping we conclude $\mathcal{M} <: \mathcal{M}''$. From (5.46) and (5.54) and rule $\text{APP}_{\Downarrow\mathsf{C}}$ we obtain $\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{m''_1 + m''_2}{\vdash} e\,\ell \Downarrow w, \mathcal{H}''$. From (5.51) and (5.59) we establish proof obligation (5.16), i.e. $m - m'' = m + (-m' + m') - m'' = (m - m') + (m' - m'') \geq m''_1 + m''_2$. Equations (5.53), (5.56), (5.57) and (5.58) establish the remaining proof obligations. This concludes the proof of the APP case.

Case CONS**:** This case cannot occur because the theorem applies only to initial expressions (not augmented expressions).

Case CASE**:** The typing and evaluation premises are

$$\Gamma, \Delta \overset{q\,+\,q'}{\vdash} \texttt{case}\ e\ \texttt{of}\ \{c_i\ \overrightarrow{x_i}\ \texttt{->}\ e_i\}^n_{i=1} : C \tag{5.60}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \texttt{case}\ e\ \texttt{of}\ \{c_i\ \overrightarrow{x_i}\ \texttt{->}\ e_i\}^n_{i=1} \Downarrow w, \mathcal{H}'' \tag{5.61}$$

From (5.61) by inversion of rule $\text{C}\textsc{ase}_{\Downarrow}$ we obtain:

$$\mathcal{H}, \mathcal{S} \cup \bigcup_{i=1}^{n} (\{\overrightarrow{x_i}\} \cup \mathrm{BV}(e_i)), \mathcal{L} \vdash e \Downarrow c_k\ \vec{\ell}, \mathcal{H}' \tag{5.62}$$

$$\mathcal{H}', \mathcal{S}, \mathcal{L} \vdash e_k[\vec{\ell}/\overrightarrow{x_k}] \Downarrow w, \mathcal{H}'' \tag{5.63}$$

From (5.60) by inversion of the typing rule $\text{C}\textsc{ase}$ we obtain:

$$\Gamma \overset{q}{\vdash} e : B \tag{5.64}$$

$$B = \mu X.\{c_1 : (p_1, \overrightarrow{A_1}) \mid \cdots \mid c_n : (p_n, \overrightarrow{A_n})\} \tag{5.65}$$

$$(\bigcup_{i=1}^{n} \{\overrightarrow{x_i}\}) \cap \mathrm{dom}(\Delta) = \emptyset \tag{5.66}$$

$$|\overrightarrow{A_k}| = |\overrightarrow{x_k}| = j \tag{5.67}$$

$$\Delta, x_{k_1}{:}A_{k_1}[B/X], \ldots, x_{k_j}{:}A_{k_j}[B/X] \overset{q' + p_k}{\vdash} e_k : C \tag{5.68}$$

From (5.68), (5.66) and (5.67) together with Lemma 5.2 (substitution) we obtain

$$\Delta, \ell_1{:}A_{k_1}[B/X], \ldots, \ell_j{:}A_{k_j}[B/X] \overset{q' + p_k}{\vdash} e_k[\vec{\ell}/\overrightarrow{x_k}] : C \tag{5.69}$$

Let m be such that

$$\begin{aligned} m &\geq t + q + q' + \phi_{\mathcal{H}}(\Gamma, \Delta) + \phi_{\mathcal{H}}(\Theta) + \Phi_{\mathcal{H}}^{\mathcal{L}}(\mathcal{C}) + \Phi_{\mathcal{H}}^{\mathcal{L}}(\mathcal{B}) \\ &= (t + q') + q + \phi_{\mathcal{H}}(\Gamma) + \phi_{\mathcal{H}}(\Delta, \Theta) + \Phi_{\mathcal{H}}^{\mathcal{L}}(\mathcal{C}) + \Phi_{\mathcal{H}}^{\mathcal{L}}(\mathcal{B}) \end{aligned} \tag{5.70}$$

We are now able to apply the induction hypothesis for expression e using (5.64) and (5.62) and obtain:

$$\Gamma' \overset{0}{\vdash} c_k\ \vec{\ell} : B \tag{5.71}$$

$$\mathcal{H}, \mathcal{S} \cup \bigcup_{i=1}^{n} (\{\overrightarrow{x_i}\} \cup \mathrm{BV}(e_i)), \mathcal{L} \overset{m_1''}{\vdash} e \Downarrow c_k\ \vec{\ell}, \mathcal{H}' \tag{5.72}$$

$$\mathcal{M} <: \mathcal{M}' \tag{5.73}$$

$$\mathcal{C}', \mathcal{B}' \vdash_{\textsf{Mem}} (\mathcal{H}', \mathcal{L}) : \mathcal{M}' \tag{5.74}$$

$$\curlyvee(\mathcal{M}' \mid (\Gamma', \Delta, \Theta), \mathcal{C}') \tag{5.75}$$

$$m' \geq (t + q') + \phi_{\mathcal{H}'}(c_k\ \vec{\ell}{:}B) + \phi_{\mathcal{H}'}(\Delta, \Theta) + \Phi_{\mathcal{H}'}^{\mathcal{L}}(\mathcal{C}') + \Phi_{\mathcal{H}'}^{\mathcal{L}}(\mathcal{B}') \tag{5.76}$$

$$m - m' \geq m_1'' \tag{5.77}$$

From (5.71), by Lemma 5.3 (inversion), we have

$$\curlyvee(\Gamma' \mid l_1{:}A_{k_1}[B/X], \ldots, l_j{:}A_{k_j}[B/X]\,) \tag{5.78}$$

From (5.75) and (5.78), global compatibility can be relaxed to

$$\curlyvee(\mathcal{M}' \mid (l_1{:}A_{k_1}[B/X], \ldots, l_j{:}A_{k_j}[B/X], \Delta, \Theta), \mathcal{C}'\,) \tag{5.79}$$

We now apply induction again, this time for expression $e_k[\vec{\ell}/\vec{x_k}]$ using (5.69), (5.63), (5.74) and (5.79). It remains to show that the bound (5.76) satisfies premise (5.9). By Def. 5.6 (potential) and (5.65) we know $\phi_{\mathcal{H}'}(c_k\ \vec{\ell}{:}B) = p_k + \sum_{i=1}^{j} \phi_{\mathcal{H}'}(\ell_i{:}A_{k_i}[B/X])$; substituting in (5.76) yields:

$$\begin{aligned} m' &\geq t + q' + p_k + \textstyle\sum_{i=1}^{j} \phi_{\mathcal{H}'}(\ell_i{:}A_{k_i}[B/X]) + \phi_{\mathcal{H}'}(\Delta, \Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}'}(\mathcal{C}') + \Phi^{\mathcal{L}}_{\mathcal{H}'}(\mathcal{B}') \\ &= t + q' + p_k + \phi_{\mathcal{H}'}\big(\Delta, \ell_1{:}A_{k_1}[B/X], \ldots, \ell_j{:}A_{k_j}[B/X]\big) + \phi_{\mathcal{H}'}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}'}\big(\mathcal{C}'\big) + \Phi^{\mathcal{L}}_{\mathcal{H}'}\big(\mathcal{B}'\big) \end{aligned}$$

Hence we can apply induction and obtain:

$$\Gamma'' \overset{0}{\vdash} w : C \tag{5.80}$$

$$\mathcal{H}', \mathcal{S}, \mathcal{L} \overset{m_2''}{\vdash} e_k[\vec{\ell}/\vec{x_k}] \Downarrow w, \mathcal{H}'' \tag{5.81}$$

$$\mathcal{M}' <: \mathcal{M}'' \tag{5.82}$$

$$\mathcal{C}'', \mathcal{B}'' \vdash_{\textsf{MEM}} (\mathcal{H}'', \mathcal{L}) : \mathcal{M}'' \tag{5.83}$$

$$\curlyvee(\mathcal{M}'' \mid (\Gamma'', \Theta), \mathcal{C}''\,) \tag{5.84}$$

$$m'' \geq t + \phi_{\mathcal{H}''}(w{:}C) + \phi_{\mathcal{H}''}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}''}\big(\mathcal{C}''\big) + \Phi^{\mathcal{L}}_{\mathcal{H}''}\big(\mathcal{B}''\big) \tag{5.85}$$

$$m' - m'' \geq m_2'' \tag{5.86}$$

From (5.73) and (5.82) and the transitivity of subtyping we conclude $\mathcal{M} <: \mathcal{M}''$. From (5.72) and (5.81) and rule CASE$_{\Downarrow\text{C}}$ we obtain

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{m_1'' + m_2''}{\vdash} \textsf{case}\ e\ \textsf{of}\ \{c_i\ \vec{x_i}\ \texttt{->}\ e_i\}_{i=1}^{n} \Downarrow w, \mathcal{H}''$$

From (5.77) and (5.86) we establish proof obligation (5.16), i.e. $m - m'' = m + (-m' + m') - m'' = (m - m') + (m' - m'') \geq m_1'' + m_2''$. Equations (5.80), (5.83), (5.84) and (5.85) establish the remaining proof obligations. This concludes the proof of the CASE case.

Case WEAK**:** The typing premise (5.5) reads $\Gamma, x{:}A \overset{q}{\vdash} e : C$. By inversion of rule WEAK we obtain $\Gamma \overset{q}{\vdash} e : C$. In order to apply the induction hypothesis for this judgement, we note that premise (5.7) (type consistency) holds unchanged; and because $\curlyvee(\mathcal{M} \mid \Gamma, x{:}A, \Theta, \mathcal{C})$ implies $\curlyvee(\mathcal{M} \mid \Gamma, \Theta, \mathcal{C})$ so does (5.8) (global compatibility). The bound (5.9) for the induction also holds because $\phi_{\mathcal{H}}(\Gamma, x{:}A) \geq \phi_{\mathcal{H}}(\Gamma)$. We can therefore apply induction to e with the typing $\Gamma \overset{q}{\vdash} e : C$ and obtain all required results for this case.

Case RELAX**:** By the second premise of RELAX follows $q - q' \geq 0$ and thus we can choose $t' = t + q - q'$. We apply the induction hypothesis to $\Gamma \overset{q'}{\vdash} e : A$ for this t'. Since RELAX is a structural rule, all statements apart from (5.5) and (5.9) remain unchanged. The induction hypothesis thus yields all required conclusions verbatim, except for (5.15). Instead, the induction yields $m' \geq t' + \phi_{\mathcal{H}'}(w{:}A) + \phi_{\mathcal{H}'}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}'}(\mathcal{C}') + \Phi^{\mathcal{L}}_{\mathcal{H}'}(\mathcal{B}')$. Unfolding our choice for t' yields $m' \geq (t + q - q') + \phi_{\mathcal{H}'}(w{:}A) + \phi_{\mathcal{H}'}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}'}(\mathcal{C}') + \Phi^{\mathcal{L}}_{\mathcal{H}'}(\mathcal{B}')$. By the second premise of RELAX follows $q - q' \geq 0$ and thus $m' \geq t + \phi_{\mathcal{H}'}(w{:}A) + \phi_{\mathcal{H}'}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}'}(\mathcal{C}') + \Phi^{\mathcal{L}}_{\mathcal{H}'}(\mathcal{B}')$ as required to conclude this case.

Case PREPAY**:** The typing premise is

$$\Gamma, \ell{:}\mathsf{T}^{q'_0 + q'}(A) \overset{q\,+\,q'}{\vdash} e : C$$

By inversion of the rule PREPAY we obtain

$$\Gamma, \ell{:}\mathsf{T}^{q'_0}(A) \overset{q}{\vdash} e : C \tag{5.87}$$

Let $\mathcal{B}' = \mathcal{B}[\ell \mapsto q' + \mathcal{B}(\ell)]$, i.e. $\mathcal{B}'$ is equal to $\mathcal{B}$ except for location ℓ where it increases by q'. Assuming m as in premise (5.9), we show that it satisfies the requirements for applying induction to (5.87) with the modified $\mathcal{B}'$:

$$\begin{aligned} m &\geq t + q + q' + \phi_{\mathcal{H}}(\Gamma, \ell{:}\mathsf{T}^{q'_0 + q'}(A)) + \phi_{\mathcal{H}}(\Theta) \; + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{C}) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B}) \\ &\geq t + q + \phi_{\mathcal{H}}(\Gamma, \ell{:}\mathsf{T}^{q'_0}(A)) + \phi_{\mathcal{H}}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{C}) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B}') \end{aligned}$$

The last inequality holds because $\phi_{\mathcal{H}}(\ell{:}\mathsf{T}^{q'_0 + q'}(A)) = \phi_{\mathcal{H}}(\ell{:}\mathsf{T}^{q'_0}(A))$ by Def. 5.6 (potential) and $q' + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B}) \geq \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B}')$; note that the latter is an equality when $\mathcal{H}(\ell)$ is not a *whnf*.

We need to reestablish both global compatibility and type consistency in order to apply the induction hypothesis. Let $\mathsf{T}^r(A') = \mathcal{M}(\ell)$. By the definition of sharing and global compatibility (5.8) we have $\curlyvee(\mathsf{T}^r(A') \mid \mathsf{T}^{q'_0+q'}(A))$ and hence $q'_0 + q' \geq r$. Define $k = \max(r - q', 0)$, and $\mathcal{M}' = \mathcal{M}[\ell \mapsto \mathsf{T}^k(A')]$.

To establish consistency for $\mathcal{M}'$, note that only the global type of location ℓ changes. Assume that (Loc2) applies, i.e. $\mathcal{H}(\ell)$ is not in *whnf* and $\ell \notin \mathcal{L}$, since otherwise the claim is trivial. From the consistency premise (5.7) we have

$$\mathcal{C}(\ell) \vdash^{r + \mathcal{B}(\ell)} \mathcal{H}(\ell) : A' \tag{5.88}$$

By the definition of k we have $k+q' = \max(r-q', 0)+q' \geq r$. Hence we can apply rule Relax to (5.88) and obtain

$$\mathcal{C}(\ell) \vdash^{k + q' + \mathcal{B}(\ell)} \mathcal{H}(\ell) : A'$$

By definition of $\mathcal{B}'$ this is equivalent to the required

$$\mathcal{C}(\ell) \vdash^{k + \mathcal{B}'(\ell)} \mathcal{H}(\ell) : A' \; .$$

To establish compatibility for $\mathcal{M}'$ we need to show

$$\curlyvee\Big(\mathsf{T}^k(A') \,\Big|\, \Gamma{\restriction_\ell}, \mathsf{T}^{q'_0}(A), \mathcal{C}{\restriction_\ell}\Big)$$

From the compatibility premise (5.8) we know

$$\curlyvee\Big(\mathsf{T}^r(A') \,\Big|\, \Gamma{\restriction_\ell}, \mathsf{T}^{q'_0+q'}(A), \mathcal{C}{\restriction_\ell}\Big) \tag{5.89}$$

First we show that $\curlyvee\Big(\mathsf{T}^k(A') \,\Big|\, \mathsf{T}^{q'_0}(A)\Big)$; by definition of sharing, we need to show $q'_0 \geq k$. By definition of k, we have $q'_0 \geq k \iff q'_0 \geq \max(r - q', 0) \iff q'_0 \geq r - q' \wedge q'_0 \geq 0 \iff q'_0 + q' \geq r \wedge q'_0 \geq 0$; the latter holds by non-negativity assumption, while the former holds by the compatibility premise above.

For other types $\mathsf{T}^s(A'')$ in either $\Gamma{\restriction_\ell}$ or $\mathcal{C}{\restriction_\ell}$, observe that $\mathsf{T}^k(A') <: \mathsf{T}^r(A')$ by construction and $\mathsf{T}^r(A') <: \mathsf{T}^s(A'')$ by the original compatibility (5.89). By transitivity we obtain the desired result.

Since the other premises remain unchanged, we can therefore apply induction and obtain precisely the results required for the conclusion of this case.

Case Share**:** The typing hypothesis is $\Gamma, \ell{:}A \overset{q}{\vdash} e : C$. By inversion of rule Share we obtain $\Gamma, \ell{:}A_1, \ell{:}A_2 \overset{q}{\vdash} e : C$ and $\curlyvee(A \mid A_1, A_2)$. Assuming m as in premise (5.9), we obtain:

$$\begin{aligned} m &\geq t + \phi_{\mathcal{H}}(\Gamma, \ell{:}A) + \phi_{\mathcal{H}}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{C}) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B}) \\ &\geq t + \phi_{\mathcal{H}}(\Gamma, \ell{:}A_1, \ell{:}A_2) + \phi_{\mathcal{H}}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{C}) + \Phi^{\mathcal{L}}_{\mathcal{H}}(\mathcal{B}) \end{aligned}$$

The last inequality holds by Lemma 5.7 (Potential Splitting) $\phi_{\mathcal{H}}(\mathcal{H}(\ell){:}A) \geq \phi_{\mathcal{H}}(\mathcal{H}(\ell){:}A_1) + \phi_{\mathcal{H}}(\mathcal{H}(\ell){:}A_2)$. We can therefore apply the induction hypothesis to e with typing premise $\Gamma, \ell{:}A_1, \ell{:}A_2 \overset{q}{\vdash} e : C$ and obtain as result the required conclusions for the case Share. This concludes the proof of this case.

Case Supertype**:** The type rule gives us $\Gamma, x{:}A \overset{q}{\vdash} e : C$ and $A <: B$. We show that we can apply induction for the premise $\Gamma, x{:}B \overset{q}{\vdash} e : C$. Type consistency holds unchanged for the induction; by $A <: B$ and the compatibility premise (5.8) $\curlyvee(\mathcal{M} \mid \Gamma, x{:}A, \Theta, \mathcal{C})$, we have $\curlyvee(\mathcal{M} \mid \Gamma, x{:}B, \Theta, \mathcal{C})$. The bound (5.9) also holds because $\phi_{\mathcal{H}}(x{:}A) \geq \phi_{\mathcal{H}}(x{:}B)$ by $A <: B$ and Lemma 5.9. Applying the induction gives us the required conclusions for the case Supertype.

Case Subtype**:** The type rule gives us $\Gamma \overset{q}{\vdash} e : C$; by inversion we obtain $\Gamma \overset{q}{\vdash} e : B$ and $B <: C$. Because the context is unchanged, we can apply induction hypothesis directly and obtain:

$$\Gamma' \overset{0}{\vdash} w : B \tag{5.90}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{m''}{\vdash} e \Downarrow w, \mathcal{H}' \tag{5.91}$$

$$\mathcal{M} <: \mathcal{M}' \tag{5.92}$$

$$\mathcal{C}', \mathcal{B}' \vdash_{\textsf{Mem}} (\mathcal{H}', \mathcal{L}) : \mathcal{M}' \tag{5.93}$$

$$\curlyvee(\mathcal{M}' \mid (\Gamma', \Theta), \mathcal{C}') \tag{5.94}$$

$$m' \geq t + \phi_{\mathcal{H}'}(w{:}B) + \phi_{\mathcal{H}'}(\Theta) + \Phi^{\mathcal{L}}_{\mathcal{H}'}(\mathcal{C}') + \Phi^{\mathcal{L}}_{\mathcal{H}'}(\mathcal{B}') \tag{5.95}$$

$$m - m' \geq m'' \tag{5.96}$$

Applying Subtype to (5.90) gives us $\Gamma' \overset{0}{\vdash} w : C$ as required for (5.10). Lemma 5.9 with $B <: C$ gives us $\phi_{\mathcal{H}'}(w{:}B) \geq \phi_{\mathcal{H}'}(w{:}C)$; substituting in (5.95) establishes the bound (5.15). Results (5.91), (5.92), (5.93), (5.94) and (5.96) directly establish the remaining proof obligations for this case.

5.6 A System for Eager Evaluation

This section emphasises the key points of the analysis for lazy evaluation developed in this thesis by contrast to the minimal changes needed to derive an analysis for eager evaluation. The complete definitions and figures of the eager system can be seen in Appendix A.

First of all, the analysis needs a cost model to be validated against. For that purpose we derive a cost model for eager evaluation from Figure 4.4 by replacing rule $\text{L}\scriptstyle{\text{ET}}_{\Downarrow\text{C}}$ with the following:

$$\frac{\ell \text{ is fresh} \qquad \mathcal{H}\big[\ell \mapsto \widehat{e}[\ell/x]\big], \mathcal{S}, \mathcal{L} \cup \{\ell\} \overset{m'}{\vdash} \widehat{e}[\ell/x] \Downarrow w', \mathcal{H}' \qquad \mathcal{H}'[\ell \mapsto w'], \mathcal{S}, \mathcal{L} \overset{m}{\vdash} e[\ell/x] \Downarrow w, \mathcal{H}''}{\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{1 + m' + m}{\vdash} \text{let } x = \widehat{e} \text{ in } e \Downarrow w, \mathcal{H}''} \; (\text{EagerLet}_{\Downarrow\text{C}})$$

The new rule $\text{EagerLet}_{\Downarrow\text{C}}$ forces evaluation of $\widehat{e}$ before evaluating the body of the let expression. Note that the cost m' of this forced evaluation is immediately added to the overall cost of the let expression while, in a lazy setting, an expression in a new location would only possibly incur a cost if its evaluation was needed indeed. Correspondingly, in rule $\text{Var}_{\Downarrow\text{C}}$ the cost m is zero since all locations introduced by $\text{EagerLet}_{\Downarrow\text{C}}$ map to *whnfs* in the heap if their evaluation terminates.

Although it is tempting to simplify the eager semantics (for example, in $\text{EagerLet}_{\Downarrow}$ we could avoid adding to $\mathcal{H}$ the mapping for ℓ when evaluating $\widehat{e}[\ell/x]$ or we could alter rule $\text{Var}_{\Downarrow}$ to remove the update since $\mathcal{H}' = \mathcal{H}'[\ell \mapsto w]$) we must refrain from doing so, remembering that the purpose of presenting an eager system in this thesis is to be able to contrast it with the lazy system. The fewer the changes, the simpler the contrast.

With respect to the type system, from Figures 5.4 and 5.5 we derive a type system suitable for eager evaluation by removing the now unneeded rule Prepay and by replacing rules Let and Var with

$$\frac{\begin{array}{c}\Gamma, x{:}A' \overset{q'}{\vdash} \widehat{e} : A \qquad \Delta, x{:}A \overset{q}{\vdash} e : C \\ x \notin \mathrm{dom}(\Gamma, \Delta) \qquad \curlyvee(A \,|\, A, A') \qquad q' = 0 \text{ if } \widehat{e} \text{ is a } \textit{whnf} \\ p = \begin{cases} p', & \text{if } \widehat{e} \equiv c\,\vec{y} \text{ and } A = \mu X.\{\cdots \,|\, c : (p', \vec{B}) \,|\, \cdots\} \\ 0, & \text{otherwise} \end{cases}\end{array}}{\Gamma, \Delta \overset{1+q'+q+p}{\vdash} \text{let } x = \widehat{e} \text{ in } e : C} \qquad (\text{EAGERLET})$$

and

$$\frac{}{x{:}A \overset{0}{\vdash} x : A} \qquad (\text{EAGERVAR})$$

respectively.

Since we removed all explicit references to thunk types from the type system, we can also derive for the eager system both a new syntax of allowed types (by removing the thunk types from Figure 5.1) and a new sharing relation (by removing rule SHARETHUNK from Figure 5.2).

In order to validate the analysis for eager evaluation against its respective cost model, we alter the invariants needed for the proof of the soundness theorem. We start by removing the now unneeded balance $\mathcal{B}$ (lazy potential). Moreover, since we no longer have references to thunk types and there is no need to account for expressions that are simultaneously not in *whnf* and not under evaluation (set $\mathcal{L}$), we can simplify the definition of potential (Figure 5.8) with respect to thunk types (also removing the auxiliary definitions of potential for global contexts $\mathcal{C}$ and balance $\mathcal{B}$) and furthermore remove case LOC2 from the definition of type consistency of locations (Definition 5.10).

The soundness theorem (Theorem 5.13) is restated according to the changes introduced for the eager system in this section:

Theorem 5.18 (Soundness of the Eager System)**.** *If the following statements hold*

$$\Gamma \overset{q}{\vdash} e : A \tag{5.97}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash e \Downarrow w, \mathcal{H}' \tag{5.98}$$

$$\mathcal{C} \vdash_{\text{MEM}} (\mathcal{H}, \mathcal{L}) : \mathcal{M} \tag{5.99}$$

$$\curlyvee(\mathcal{M} \,|\, (\Gamma, \Theta), \mathcal{C}) \tag{5.100}$$

then for all $t \in \mathbb{Q}_0^+$ *and* $m \in \mathbb{N}$ *with*

$$m \geq t + q + \phi_{\mathcal{H}}(\Gamma) + \phi_{\mathcal{H}}(\Theta) \tag{5.101}$$

there exist Γ', $\mathcal{C}'$, $\mathcal{M}'$ *and* $m', m'' \in \mathbb{N}$ *such that the following statements also hold*

$$\Gamma' \overset{0}{\vdash} w : A \tag{5.102}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{m''}{\vdash} e \Downarrow w, \mathcal{H}' \tag{5.103}$$

$$\mathcal{M} <: \mathcal{M}' \tag{5.104}$$

$$\mathcal{C}' \vdash_{\text{MEM}} (\mathcal{H}', \mathcal{L}) : \mathcal{M}' \tag{5.105}$$

$$\curlyvee(\mathcal{M}' \,|\, (\Gamma', \Theta), \mathcal{C}') \tag{5.106}$$

$$m' \geq t + \phi_{\mathcal{H}'}(w{:}A) + \phi_{\mathcal{H}'}(\Theta) \tag{5.107}$$

$$m - m' \geq m'' \tag{5.108}$$

Except for EAGERLET, the proof of the eager system is omitted since all cases are similar to (or simpler than) the ones presented in the soundness proof of the lazy system (in Section 5.5.6.7). The proof of the eager system can be seen in the Appendix A.2.

Note that, apart from the expected changes to the operational semantics (EAGERLET$_{\Downarrow}$) and its corresponding type rules (EAGERLET and EAGERVAR), the fundamental difference between the lazy and the eager systems presented in this chapter is rule PREPAY, that allows the lazy system to prepay or otherwise defer the costs of thunks. Without rule PREPAY, the eager system does not need thunk types nor *lazy potential* (global balance $\mathcal{B}$) and consequently there is no need to handle those in the definitions of sharing, potential and type consistency.

5.7 Summary

In this chapter we have presented a type-based amortised analysis of total heap allocations for lazily evaluated programs and proved that its statically determined bounds are not exceeded during run-time. We have also emphasised the key elements needed in the development of this analysis for lazy evaluation by contrasting the lazy system with a specifically tailored eager system.

The eager system implicitly forces PREPAY when allocating new heap cells (EAGERLET), while the lazy system is flexible enough to allow prepaying part of the cost of a named expression (all of the cost, none or something in between) at allocation (LET) and defer the remainder to the references of the expression (VAR).

Note that the difference of modelling call-by-name would be not prepaying at all and instead defering the cost to *every* reference, and although we would avoid paying the cost of expressions that are not referenced, we might have to pay the full cost multiple times (corresponding to the number of references to the named expression).

The next chapter discusses the applicability of our analysis for lazy evaluation with some concrete program examples.

6. Experimental Results

This chapter illustrates the strenghts and limitations of our approach through a series of examples. We first analyse a higher-order function, `map`. Then, we use the bounds obtained for `map` in a list fusion example in order to show that our analysis can capture intensional behaviour, by comparing two programs and their respective bounds as given by our analysis. Afterwards, we show the accurate bounds predicted for an infinite list constructed as a cycle of a finite non-empty list. We then analyse a function `concat` for an example that deals with nested data structures. Finally, we describe an interesting limitation that we have found for our analysis.

In this chapter we abbreviate the type of lists of A as:

$$\mathtt{L}^{q_t}(p_c, p_n, A) \overset{\text{def}}{=} \mu X.\{\, \mathtt{Cons} : (p_c,\ (A, \mathsf{T}^{q_t}(X))) \mid \mathtt{Nil} : (p_n,\ ())\,\}$$

where q_t is an upper bound on the maximum of the costs of evaluating to *whnf* each of the tails of the `Cons` nodes of the list, and p_c and p_n are the potentials assigned to each `Cons` node and `Nil` node of the list, respectively. Also, whenever we have $\mathsf{T}^{q_0}(\mathtt{L}^{q_t}(p_c, p_n, A))$, the q_0 represents an upper bound on the cost of evaluating the list to *whnf*.

Note that type derivations show *one* possible solution. In most examples the best solution would depend on the context. We provide alternative solutions for the first example that try to take advantage of more concrete use cases, since these types will also help us explain two other examples in this chapter. In particular, we present two alternative types for `map`: one when applied to lists with potential and the other when applied to lists with zero potential such as circular lists (which the soundness of our system prevents from having potential).

6.1 Higher-Order Functions: `map`

The first example we will analyse using our system is the function `map` which takes as arguments a function `f` and a list `xs` and returns a new list constructed from the results of applying `f` to every element of `xs`.

$$\begin{aligned}
&\text{let } \mathtt{map} = \lambda\mathtt{f}.\lambda\mathtt{xs}.\,\text{case } \mathtt{xs} \text{ of } \mathtt{Nil} \text{ -> } \text{let } \mathtt{nil} = \mathtt{Nil} \text{ in } \mathtt{nil},\\
&\qquad\qquad\qquad\qquad\qquad \mathtt{Cons\ x\ xs'} \text{ -> } \text{let } \mathtt{y} = \mathtt{f\ x} \text{ in}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{let } \mathtt{ys'} = \mathtt{map\ f\ xs'} \text{ in}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{let } \mathtt{ys} = \mathtt{Cons\ y\ ys'} \text{ in } \mathtt{ys} \text{ in}\\
&\mathtt{map}
\end{aligned}$$

Assuming q_f is an upper bound on the cost of applying `f` to an element of `xs`, and p'_c and p'_n are the potentials associated to the `Cons` nodes and the `Nil` node of the output list, respectively, we can derive the following informative type for `map`:*

$$\mathsf{T}^0\Big(\mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L_{in}}) \xrightarrow{q_0} \mathtt{L_{out}}\Big) \text{ where } \begin{aligned}\mathtt{L_{in}} &= \mathtt{L}^{q_t}(3{+}q_f{+}q_l{+}p'_c, 1{+}p'_n, A)\\ \mathtt{L_{out}} &= \mathtt{L}^0(p'_c, p'_n, \mathsf{T}^0(B))\\ q_l &= \mathtt{max}(q_0, q_t)\end{aligned}$$

According to this type, once applied to a function `f` and a list `xs` of length n, the cost of `map` is bounded by $q_0 + n(3{+}q_f{+}q_l{+}p'_c) + 1{+}p'_n$, i.e. the cost of `map` is bounded by q_0 — the cost over the arrow type — plus $n(3{+}q_f{+}q_l{+}p'_c)$ — n times the potential required for each `Cons` node of the input list — plus $1{+}p'_n$ — the potential required for the `Nil` node of the input list. Also, observe that any potential carried to the output list (p'_c and p'_n) imposes an extra requirement on the potentials of the input list. Note that in general when applying a function, we can use rule PREPAY to shift to the caller (or turnstile) the costs of evaluating the arguments to *whnf*. For example, when applying `map` to its list argument we can prepay q_0 and make subsequent costs of evaluating the tail of the `Cons` nodes depend on q_t instead of q_l.

In terms of quality, provided q_f, q_0, q_t and q_l are actual costs (and not just upper bounds) and `map` is evaluated in a context that demands all elements of the output list to be in *whnf* (and, for simplicity, the output list has zero potential), the analysis of `map` gives an exact

[*] See Figure B.3 in Appendix B for the type derivation.

match to its operational cost. Dividing the cost into three parts — q_0, 1 and $n(3{+}q_f{+}q_l)$ — we can see that the first part corresponds to evaluating xs in order to determine which of the case alternatives apply. The second part corresponds to the cost of allocating a heap cell for the nil. Finally $n(3{+}q_f{+}q_l)$ corresponds to, for every Cons node in xs, the cost of allocating three heap cells (for y, ys′ and ys) plus the cost of applying f to x plus q_l, the cost of applying map to f and xs′, where we have to take the maximum between q_0 (the cost that map expects for its input list) and q_t (the cost of evaluating xs′ to *whnf*).

The above type for map has the advantage that if the length of the input list is known as well as the other parameters q_f, q_0, q_t, p'_c and p'_n, then an upper bound on the cost of map is determined by a simple linear formula. However, with lazy evaluation, the actual cost largely depends on the context in which evaluation takes places. For instance, suppose that we knew for a fact that we would not need the *whnfs* of the elements in the output list. Then, a more suitable type for reasoning about map (also derivable in our system) would have been to have $\mathtt{L_{in}} = \mathtt{L}^{q_t}(3{+}q_l{+}p'_c, 1{+}p'_n, A)$ and $\mathtt{L_{out}} = \mathtt{L}^0(p'_c, p'_n, \mathsf{T}^{q_f}(B))$. The corresponding cost formula would then be $q_0 + n(3{+}q_l{+}p'_c) + 1{+}p'_n$, and although each element of the output list had a latent cost q_f, we could ignore those, since we had known that we would not need to evaluate them.

Note that we have considered types for map that are valid only when applied to a non-circular list, since they rely on the input list having positive potential (and the soundness of our type system only allows circular data structures without potential). However, it is important to note that we can similarly derive a type for map when the input list has zero potential:[†]

$$\mathsf{T}^0\Big(\mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\, p'_n-2)} \mathtt{L_{out}}\Big)$$
$$\text{where } \mathtt{L}'_{\mathtt{in}} = \mathtt{L}^{q_l}(0, 0, A'),\ \text{with } \curlyvee(A' \mid A', A')$$
$$\mathtt{L_{out}} = \mathtt{L}^{3+q_f+q_l+p'_c}(p'_c, p'_n, \mathsf{T}^0(B))$$

Note that since q_0 and q_t do not contribute independently to the cost formula for this type, we use the maximum between them (q_l) throughout instead. In this case, the cost formula is expressed in terms of how many elements, say m, of the output list are demanded from the context of the evaluation of map. The corresponding cost formula is now $3+q_f+q_l+\mathtt{max}(p'_c, p'_n{-}2) + m(3{+}q_f{+}q_l{+}p'_c)$. When comparing with the cost formula of the initial type presented for map, it is interesting to realise that the higher bound now obtained is due to the inability of having separate potential for the Cons nodes and the Nil node of the input list,

[†] See Figure B.6 in Appendix B for the type derivation.

and for that reason there is a fixed cost $(3{+}q_f{+}q_l{+}\mathtt{max}(p'_c, p'_n{-}2))$ covering both alternatives.

Note the change in the expected type for function `f`: since the input list has zero potential, the argument to `f` cannot have potential as well.

Having discussed `map` we now move to another example that benefits from the types just presented.

6.2 List Fusion: `map/map`

The following two Haskell programs are equivalent in the sense that, given the same input, produce the same output:

- `progA f g = map f . map g`
- `progB f g = map (\x -> f (g x))`

Translating to our Fun language:

- let $\mathtt{progA} = \lambda \mathtt{f}.\lambda \mathtt{g}.\lambda \mathtt{xs}.$let $\mathtt{ys} = \mathtt{map1}\ \mathtt{g}\ \mathtt{xs}$ in $\mathtt{map2}\ \mathtt{f}\ \mathtt{ys}$ in $\mathtt{progA}$
- let $\mathtt{progB} = \lambda \mathtt{f}.\lambda \mathtt{g}.\lambda \mathtt{xs}.$let $\mathtt{h} = (\lambda \mathtt{x}.$let $\mathtt{y} = \mathtt{g}\ \mathtt{x}$ in $\mathtt{f}\ \mathtt{y})$ in $\mathtt{map}\ \mathtt{h}\ \mathtt{xs}$ in $\mathtt{progB}$

where `map1` and `map2` are two copies of the code for `map`, as defined in the previous section, in order to enable our analysis to annotate the types of each copy differently and successfully handle the first of the above programs. *Resource parametricity* as previously established [JLHH10] would avoid the need for such code duplication and its adoption here is suggested as further work.

Conceptually, `progA` constructs an intermediate list `ys` from applying `g` to every element of `xs` and then returns a final list from applying `f` to every element of `ys`, while `progB` starts by creating a function, that, given `x` as an argument, creates an intermediate value `y` from applying `g` to `x` and proceeds by applying `f` to `y`, and then returns a final list from applying the newly defined function `h` to every element of `xs`. Note though that under a call-by-need semantics these intermediate structures are created as needed and not up front as this conceptual description would seem to imply.

In this section we want to show that our analysis can express intensional behaviour of programs. In particular, we want to show that operationally `progB` allocates fewer heap cells than `progA` and that our analysis captures this difference in cost, despite the fact that both programs produce the same output if given the same input.

The types given by our analysis for `progA` and `progB` are, respectively:

- $\mathsf{T}^0\Big(\mathsf{T}^0(\mathsf{T}^0(A) \xrightarrow{q_g} B) \xrightarrow{0} \mathsf{T}^0(\mathsf{T}^0(B) \xrightarrow{q_f} C) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L}_{\mathtt{inA}}) \xrightarrow{1+q_0} \mathtt{L}_{\mathtt{out}}\Big)$
- $\mathsf{T}^0\Big(\mathsf{T}^0(\mathsf{T}^0(A) \xrightarrow{q_g} B) \xrightarrow{0} \mathsf{T}^0(\mathsf{T}^0(B) \xrightarrow{q_f} C) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L}_{\mathtt{inB}}) \xrightarrow{1+q_0} \mathtt{L}_{\mathtt{out}}\Big)$

where[‡]

$$\begin{aligned}
\mathtt{L}_{\mathtt{inA}} &= \mathtt{L}^{q_t}(6+q_g+q_f+q_t+p'_c, 2+p'_n, \mathsf{T}^0(A)) \\
\mathtt{L}_{\mathtt{inB}} &= \mathtt{L}^{q_t}(4+q_g+q_f+q_t+p'_c, 1+p'_n, \mathsf{T}^0(A)) \\
\mathtt{L}_{\mathtt{out}} &= \mathtt{L}^{0}(p'_c, p'_n, \mathsf{T}^0(C))
\end{aligned}$$

First note that the only two differences between the above types are that 6 and 2 in $\mathtt{L}_{\mathtt{inA}}$ are replaced by 4 and 1 in $\mathtt{L}_{\mathtt{inB}}$. Now recall from the previous section that, assuming q_f (and q_g), q_0 and q_t are actual costs (and not just upper bounds), the analysis of `map` gives an exact match to its operational cost. We also assume for simplicity that we are not interested in having potential in the resulting list ($p'_c = p'_n = 0$).

We start by showing that the type given for `progA` corresponds to its expected cost. According to its type, `progA` has the following cost when applied to two functions and a list of length n

$$1+q_0+n(6+q_g+q_f+q_t)+2$$

Let us divide the cost formula into three parts, $1+q_0$, $n(6+q_g+q_f+q_t)$ and 2, and explain each of them. The first part corresponds to a fixed cost of $1+q_0$, where the 1 corresponds to allocating a heap cell for binding `ys` to the thunk `map1 g xs` and the q_0 is prepayment for evaluating `xs` to *whnf*. In the second and third parts the $6+q_g+q_f+q_t$ and the 2 correspond to the cost of processing each `Cons` node and the `Nil` node of `xs` in `progA`, respectively. In order to understand the origin of these costs, we have to consider what types do `map1` and `map2` have, in this concrete example. Given that q_0 has been prepaid in the first part as a fixed cost, the type for the input list that `map1` is expecting is $\mathsf{T}^0\big(\mathtt{L}^{q_t}(3+q_g+q_t+p_c, 1+p_n, \mathsf{T}^0(A))\big)$, where p_c and p_n are extra amounts that are returned in the output list of `map1`. Also, since

[‡] Note that the cost of zero in the thunk type $\mathsf{T}^0(A)$ assumes that the eventual cost of the thunk has been shifted to q_g. Similarly, for $\mathsf{T}^0(B)$ and q_f.

q_0 has been prepaid, the cost of applying `map1` to `g` and `xs` is zero, which is also the cost of `ys`. Since the output of `map1` has type $\mathtt{L}^0(p_c, p_n, \mathsf{T}^0(B))$ and `ys` has cost zero, the type for the input list of `map2` is $\mathsf{T}^0\big(\mathtt{L}^0(p_c, p_n, \mathsf{T}^0(B))\big)$, the cost of applying `map2` to `f` and `ys` is zero and the type of the output of `map2` is $\mathtt{L}^0(0, 0, \mathsf{T}^0(C))$, the same type as the output of `progA`. Note from the type of `map` that the extra amounts p_c and p_n are $3{+}q_f{+}0{+}0$ and $1{+}0$, respectively. Now we can see where the costs for the second and third parts come from, since $3{+}q_g{+}q_t{+}p_c = 3{+}q_g{+}q_t{+}(3{+}q_f) = 6{+}q_g{+}q_f{+}q_t$ and $1{+}p_n = 1{+}1 = 2$. Since the cost of `map` is an exact match to its operational cost and no expression in `progA` is unaccounted for, we conclude that the cost formula of `progA` shown above is accurate.

We use a similar argument to demonstrate that the type given for `progB` also corresponds to its expected cost. The cost formula is now

$$1{+}q_0{+}n(4{+}q_g{+}q_f{+}q_t){+}1$$

which we also divide in three parts: $1{+}q_0$, $n(4{+}q_g{+}q_f{+}q_t)$ and 1. The first part corresponds again to a fixed cost of $1{+}q_0$, but this time the 1 corresponds to allocating a heap cell for binding `h` to the λ-abstraction (λ`x`.let `y = g x` in `f y`), while the q_0 is still prepayment for evaluating `xs` to *whnf*. Before looking at the second and third parts, it is useful to reason about the cost of applying function `h`. We know function `h` allocates a heap cell for the binding of `y` to the application of `g` to argument `x`, and returns `f` applied to `y`. So, we know `h` costs $q_h = 1{+}q_g{+}q_f$ to apply. Now going back to the second and third parts of the cost formula of `progB`, since q_0 has been prepaid, the specific type for `map` in `progB` is

$$\mathsf{T}^0\Big(\mathsf{T}^0(\mathsf{T}^0(A) \xrightarrow{q_h} C) \xrightarrow{0} \mathsf{T}^0(\mathtt{L}^{q_t}(3{+}q_h{+}q_t, 1, \mathsf{T}^0(A))) \xrightarrow{0} \mathtt{L}^0(0, 0, \mathsf{T}^0(C))\Big)$$

and it is clear now that the costs for the second and third parts of the cost formula of `progB` come from the potential assigned to the input list of `map`, in particular the `Cons` nodes: $3{+}q_h{+}q_t = 3{+}(1{+}q_g{+}q_f){+}q_t = 4{+}q_g{+}q_f{+}q_t$. Again, since the cost of `map` is an exact match to its operational cost and no expression in `progB` is unaccounted for, we conclude that the cost formula of `progB` shown above is accurate, indeed showing that our analysis is able to measure deforestation benefits.

6.3 Infinite Data Structures: `cycle`

This section serves to demonstrate that our static analysis can obtain accurate bounds when applied to definitions of infinite data structures.

Consider the following program

```
let append′ = λys.λxs. case xs of Nil -> ys,
                             Cons x xs′ -> let ws′ = append′ ys xs′ in
                                           let ws = Cons x ws′ in
                                           ws in
let cycle = λzs. let zs′ = append′ zs′ zs in zs′ in
cycle
```

where `cycle` is a function that, given a finite non-empty list as its argument, generates an infinite list by constructing a copy of the input list and connecting its end back with the beginning, effectively creating a circular list. The function `cycle` uses the auxiliary `append′`, which is defined as the classical `append`, except for having its argument order reversed. This change is necessary since our system only allows potential in the innermost argument of a function (rule Abs of our type system forces context Γ to be idempotent) and thus, if we want the cost of applying `append` to be paid from the potential in the recursive argument, we must swap the order of arguments (as in this example) or use an *uncurried* version of the function (as we will see in the next section).

In our type system we can derive the following type for `cycle`:

$$
\begin{aligned}
&\mathtt{T}^{0}\Big(\mathtt{T}^{q_0}(\mathtt{L_{in}}) \xrightarrow{1+q_0} \mathtt{L'_{out}}\Big)\\
&\qquad \text{where } \mathtt{L_{in}} = \mathtt{L}^{q_t}(2+q_t, 0, A)\\
&\qquad\qquad\quad \mathtt{L'_{out}} = \mathtt{L}^{0}(0, 0, A'), \text{ with } \curlyvee(\mathtt{L'_{out}} \mid \mathtt{L'_{out}}, \mathtt{L'_{out}}) \text{ and } \curlyvee(A \mid A, A')
\end{aligned}
$$

Note that, since the outermost argument `ys` of `append′` cannot have potential, the output list of `append′` cannot have potential as well, since for the case alternative of the `Nil` branch, the returning expression is `ys`. However, given that `cycle` outputs a circular list and our system does not allow circular data structures with (positive) potential[§], the restriction on `append′` does not negatively affect the type of `cycle`, since we would not expect its output

[§] Note in Figure 5.4 the use of an idempotent type A' in the recursive typing of rule Let.

list $\mathtt{L}'_{\mathtt{out}}$ to have potential anyway.

We now show that the bounds given by our analysis are tight. According to the type of `cycle`, we have the following cost formula

$$1{+}q_0{+}n(2{+}q_t)$$

where n is the length of `zs` with $n \geq 1$ (otherwise, `cycle` applied to the empty list would fail to terminate). We divide the cost formula into two parts: a fixed part $1{+}q_0$ and a part that depends on the length of the input list $n(2{+}q_t)$. In the first part, the 1 corresponds to the heap allocation for the let-binding of `zs′`, while the q_0 corresponds to a prepayment of the cost of evaluating `zs` to *whnf*. The second part corresponds to, for each `Cons` node of `xs` in `append′`, the cost of allocating two heap cells for the let-bindings of `ws′` and `ws` plus a prepayment for the evaluation of `xs′` to *whnf*. Note that `ys` acts as a reference to a copy of `zs` and can be seen as a thunk with zero cost, provided the cost of constructing a copy of the `Cons` nodes of `zs` has been prepaid for, as in this case. Since we have covered the cost of all the expressions in the program, we conclude that the cost formula shown above is tight, as long as q_0 and q_t are actual costs and not just upper bounds.

6.4 Nested Data Structures: `concat`

In this section we show the applicability of our analysis to nested data structures, using a function `concat`. The classical list concatenation function is defined as taking a list of lists as its argument and creating a single list by appending each of the inner lists to the previous one. Here, we define the following alternative version to the classical list concatenation, using an auxiliary function `appendp`:

```
let appendp = λp. case p of Pair xs ys -> case xs of Nil -> ys,
                                                     Cons x xs′ -> let p′ = Pair xs′ ys in
                                                                   let zs′ = appendp p′ in
                                                                   let zs = Cons x zs′ in
                                                                   zs in
```

```
let concat = λxss. case xss of Nil -> let nil = Nil in nil,
                         Cons xs xss′ -> let ys = concat xss′ in
                                         let p = Pair xs ys in
                                         appendp p in
concat
```

We choose to define `concat` with `appendp` and not with the `append′` seen in the previous section. While this allows us to show another alternative version of `append` successfully handled by our analysis and avoids imposing unnecessary constraints on the output list (since the output list can now have potential, unlike the output of `append′`), `appendp` does have a higher cost due to the construction of a pair for each call of this *uncurried* version and this is reflected on the following type for $concat$

$$
\begin{aligned}
\texttt{concat}:\ & \mathsf{T}^0\big(\mathsf{T}^{q_{o0}}(\mathtt{L_{outer}}) \xrightarrow{q_{o0}} \mathtt{L_{final}}\big)\\
\text{where } \mathtt{L_{outer}} &= \mathtt{L}^{q_{ot}}(2{+}q_{ol}{+}q_{i0}, 1{+}p'_n, \mathsf{T}^{q_{i0}}(\mathtt{L_{inner}}))\\
\mathtt{L_{inner}} &= \mathtt{L}^{q_{it}}(3{+}q_{it}{+}p'_c, 0, A)\\
\mathtt{L_{final}} &= \mathtt{L}^{0}(p'_c, p'_n, A)\\
\mathtt{q_{ol}} &= \mathtt{max}(q_{o0}, q_{ot})
\end{aligned}
$$

whose derivation uses the following type for `appendp`

$$
\begin{aligned}
& \mathsf{T}^0\Big(\mathtt{P}(\mathsf{T}^0(L_{inner}), \mathsf{T}^0(\mathtt{L'_{final}})) \xrightarrow{0} \mathtt{L'_{final}}\Big)\\
& \quad \text{where } \mathtt{P}(A,B) = \mathsf{T}^0(\mu X.\{\, \mathtt{Pair} : (0,\ (A,B))\,\})\\
& \quad \mathtt{L'_{final}} = \mathtt{L}^0(p'_c, 0, A)
\end{aligned}
$$

where q_{o0} and q_{ot} are the usual costs of a list (as defined in the introduction of the current chapter), in this case for the outer list of `concat`, and q_{i0} and q_{it} are the usual costs applied to the inner lists, but taking the maximum of such costs for each of the inner lists of `concat`, i.e. q_{i0} is an upper bound on the maximum of the costs of evaluating to *whnf* each of the inner lists (`xs`) and q_{it} is an upper bound on the maximum of the costs of evaluating to *whnf* each of the tails of the `Cons` nodes of the inner lists (`xs′`).

Assuming, for simplicity, that we are not interested in the potential of the output list ($p'_c = 0$), the cost formula extracted from the type of `appendp` is

$$l(3+q_{it})$$

where l is the length of the first list of the input. We start by explaining how the cost formula relates to the definition of `appendp`. First note that the type assigned by our analysis assumes that the first list of the input pair costs zero to evaluate to a *whnf* (or assumes that this cost has been prepaid). Now, looking at the program definition, the cost of the case expression for the pair is equal to the cost of the case expression for the list `xs`, since `p`, from its type, costs nothing to evaluate to *whnf*. We can also see in the type that `xs` and `ys` cost zero to evaluate to *whnf*, and thus, the cost of the case expression for the list is equal to the cost of the `Cons` case alternative, which in turn, for each $Cons$ node of `xs`, corresponds to a cost of 3 for the three heap cells storing the thunks referenced by `p`′, `zs`′ and `zs`, plus the cost of prepaying q_{it} for `xs`′ since `appendp` expects a pair of lists that cost zero to evaluate to *whnf* (or have those costs prepaid, as in this case). Note that applying `appendp` to `p`′ has no extra cost and that not only `zs` is in whnf, but also evaluating each of its `Cons` nodes also has no extra cost, according to the type $\mathtt{L}'_{\mathtt{final}}$ of the application `appendp p`′. We have thus related each expression in the definition of `appendp` to the cost formula expressed by its type.

We now do the same with respect to `concat`. According to its type, and ignoring, for simplicity, p'_c and p'_n, we have the following cost formula

$$q_{o0}+1+n(2+q_{ol}+q_{i0})+m(3+q_{it})$$

where n is the length of outer list passed as input to `concat` and m is the sum of the lengths of the inner lists ($m = l_1, \ldots, l_n$). Connecting the cost formula to the definition of `concat`, we can see that, once applied to a list of lists `xss`, `concat` evaluates the case discriminant (costing q_{o0}). When `concat` reaches the end of the outer list, it costs 1 for the heap cell allocated by the let-binding for `nil`. Meanwhile, we have to consider the cost of each `Cons` node of the outer list, and it useful to consider its part on the cost formula $n(2+q_{ol}+q_{i0})+m(3+q_{it})$ as

$$\sum_{i=1}^{n}(2+q_{ol}+q_{i0}+l_i(3+q_{it}))$$

So, for each inner list of the input to `concat`, it costs 2 heap cells to create the two let-bindings for `ys` and `p`. We also have to pay q_{ol}, as the worst case between the cost q_{ot} of evaluating $\texttt{xss}'$ to *whnf* and the cost q_{o0} that the type of `concat` expects for its input list. Furthermore, we prepay the cost q_{i0} of `xs`, since the type of `appendp` expects a list with no cost, and pay the cost $l_i(3+q_{it})$ of applying `appendp` to `p`. Thus, we have related each expression in the definition of `concat` to the cost formula expressed by its type.

Note that m is the sum of the lengths of the inner lists, taking each length separately into account, and thus it does not introduce a source of relaxation on the cost, unlike, for example, if we had considered m as $n \times \texttt{max}(l_1, \ldots, l_n)$. The cost formula for `concat` is an exact match to its operational cost, provided q_{o0}, q_{ot}, q_{i0}, q_{it} and q_{ol} are exact values and not just upper bounds, and therefore the quality of the bounds is the best we could hope for[¶].

6.5 Known Limitation with Co-Recursive Definitions: `fibs`

Non-strict functional languages allow the use of an idiom that consists of concisely defining an infinite list where, other than a finite number of initial elements, each element depends on previous ones. The classical definition of the *Fibonacci series* is an example of such idiom and is written in Haskell as

$$\texttt{fibs} = 0 : 1 : \texttt{zipWith}\ (+)\ \texttt{fibs}\ (\texttt{tail}\ \texttt{fibs})$$

Unfortunately, although `fibs` has a linear cost with respect to the number of elements needed from this infinite list, our analysis cannot capture that fact and cannot find a solution for this example. In fact, we have found that our analysis cannot handle such examples and we discuss the difficulties in the remainder of this section.

In order to isolate the problem, we highlight the difficulties with what we believe to be one of the simplest examples of this idiom, concisely written in Haskell as

$$\texttt{bools} = \texttt{True} : \texttt{map}\ \texttt{not}\ \texttt{bools}$$

[¶] Note that, by definition, q_{ot}, q_{it} and q_{ol} are likely to be a source of relaxation of the cost. However, this loss of precision is expected of any static analysis, since it results from the need to create a single abstraction to represent an infinity of concrete data.

and translated into our Fun language as

```
let true = True in
let false = False in
let not = λb. case b of True -> false, False -> true in
let map = λf.λxs. case xs of Nil -> let nil = Nil in nil,
                             Cons x xs′ -> let y = f x in
                                           let ys′ = map f xs′ in
                                           let ys = Cons y ys′ in ys in
let bools = let bls′ = map not bools in
            let bls = Cons true bls′ in
            bls in
bools
```

The bools example defines an infinite list of alternating booleans (arbitrarily starting with True) where each element, other than the first, is defined as the negation of the preceeding element.

We start by observing that bools yields a constant cost for each successive element (and thus has a linear cost with respect to its length). Evaluating bools to a *whnf*, in order to access its first element, costs 2, corresponding to the allocation of two heap cells that hold the thunk map not bools and the *whnf* Cons true bls′. Each subsequent element costs 3 heap allocations, corresponding to the three *lets* in the Cons branch of function map. Given this reasoning, we would like to obtain a typing such as bools : $\mathsf{T}^2\big(\mathtt{L}^3(0, 0, \mathsf{T}^0(\mathtt{Bool}))\big)$, where $\mathtt{Bool} \overset{\text{def}}{=} \mu X.\{\, \mathtt{True} : (0,\ ())\ |\ \mathtt{False} : (0,\ ())\,\}$.

Recall the type of map, for input lists without potential, as shown in Section 6.1:

$$\mathsf{T}^0\Big(\mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\, p'_n-2)} \mathtt{L}_{\mathtt{out}}\Big)$$
$$\text{where } \mathtt{L}'_{\mathtt{in}} = \mathtt{L}^{q_l}(0, 0, A'),\ \text{ with } \curlyvee(A' \,|\, A', A')$$
$$\mathtt{L}_{\mathtt{out}} = \mathtt{L}^{3+q_f+q_l+p'_c}(p'_c, p'_n, \mathsf{T}^0(B))$$

Applying to this concrete example: $A' = \mathsf{T}^0(\mathtt{Bool})$, $q_f = 0$ (since applying not to a boolean has zero cost), $B = \mathtt{Bool}$ and, for simplicity, assuming we are not interested in having potential in the output list, $p_c' = 0$ and $p_n' = 0$. We thus have

$$\mathsf{T}^0\Big(\mathsf{T}^0(\mathsf{T}^0(\texttt{Bool}) \xrightarrow{0} \texttt{Bool}) \xrightarrow{0} \mathsf{T}^{q_l}(\texttt{L}'_{\texttt{in}}) \xrightarrow{3+q_l} \texttt{L}_{\texttt{out}}\Big)$$
$$\text{where } \texttt{L}'_{\texttt{in}} = \texttt{L}^{q_l}(0, 0, \mathsf{T}^0(\texttt{Bool}))$$
$$\texttt{L}_{\texttt{out}} = \texttt{L}^{3+q_l}(0, 0, \mathsf{T}^0(\texttt{Bool}))$$

Since in `bools` the output list of `map` is passed back again as the input list, the types $\texttt{L}_{\texttt{out}}$ and $\texttt{L}'_{\texttt{in}}$ must match, but then our analysis fails to produce a type for `bools` due to the impossibility of finding a finite solution to $q_l = 3+q_l$ (the costs in $\texttt{L}'_{\texttt{in}}$ and $\texttt{L}_{\texttt{out}}$).

However, the real problem with this co-recursive definition is that, because of lazy evaluation, the cost of the recursive call of `map` $(3+q_l)$ is shared with the cost of obtaining each element of the output list (also $3+q_l$). Unfortunately, the rules of our type system (including PREPAY) are not enough to track the circular sharing dependencies which would allow lowering the costs of thunk types. Therefore, we conclude that our analysis cannot handle such examples of co-recursive functions.

It is important to note that `bools` can be rewritten in a way for which our analysis obtains accurate results. For example, in the translation to our Fun language of the following Haskell code

$$\texttt{bools} = \texttt{iter not True}$$
$$\text{where } \texttt{iter f x} = \texttt{x} : \texttt{iter f } (\texttt{f x})$$

`bools` has type

$$\mathsf{T}^{3+p_c}(\texttt{L}^{3+p_c}(p_c, p_n, \mathsf{T}^0(\texttt{Bool})))$$

where any potential in the `Cons` nodes p_c must be paid for from the costs of evaluating `bools`, and subsequent tails, to its *whnf*, while the potential in the `Nil` node p_n has no restriction since `bools` never creates such node. We could do better and rewrite `bools` as a circular definition having constant overall cost, such as

```
let true = True in
let false = False in
let bools = let bls′ = Cons false bools in
            let bls = Cons true bls′ in
            bls in
bools
```

(in Haskell it could be written as $\texttt{bools} = \texttt{True} : \texttt{False} : \texttt{bools}$), which has type

$$\mathsf{T}^2\big(\mathtt{L}^0(0, 0, \mathsf{T}^0(\mathtt{Bool}))\big)$$

and thus, although here we could not have potential in `bools` if we wanted to[‖], this version has better cost.

While we believe the remaining examples with linear cost of this idiom can also be rewritten in a way our analysis can handle, such reformulations might not feel natural for some examples. We would like to avoid forcing programmers out of this style when using our analysis and we will pursue a solution to this problem as further work.

6.6 Summary

In this chapter we have shown how our analysis provides accurate cost bounds for functions such as `map`, `cycle` and `concat`, thus covering examples of higher-order functions and the use of infinite and nested data structures. We have also seen how our analysis can hint into which alternative program definition has better operational cost.

Remember though, that all static analyses are doomed to fail for some programs and we did show some examples that in particular our analysis finds problematic. Some limitations such as that of `append` have simple workarounds by swapping the order of arguments or using an *uncurried* version, but each has its drawbacks: restricted output potential or increased cost of *uncurrying* the input. Other limitations are left as further work, such as the one found on the co-recursive definitions of the previous section and the one that restricts our analysis to programs with linear costs with respect to the number of constructors in data structures.

[‖] Circular data structures in our system cannot have potential other than zero. This is similar to the restriction found in the output of function `cycle` in Section 6.3.

7. Conclusion

In this chapter we summarise the work described in this thesis and note the limitations of our approach together with a discussion of further work.

7.1 Assessment of Achievements

Analyses for lazily evaluated programs were restricted to first-order programs or were not automatic or depended on context information currently impractical to obtain or made the relation between costs and inputs more opaque by not expressing data-dependencies in the bounds.

This thesis has introduced the first automatic static analysis for accurately determining bounds on the execution costs of lazy functional programs. The analysis uses an amortised analysis technique that is capable of directly analysing higher-order lazy programs, without requiring defunctionalisation or other non-cost-preserving program transformations. Our analysis deals with user-defined (potentially infinite) data structures and data-dependencies are expressed in the produced bounds. We have presented a soundness proof, validating the analysis against an operational semantics derived from Launchbury's natural semantics of graph reduction, and analysed in detail some non-trivial examples of lazy evaluation using the rules of our system, while providing a *URL* to a web-prototype implementation of the analysis where more examples can be found and users can try their own. From our novel analysis for lazy evaluation we have derived with minimal changes an analysis for eager evaluation, clearly highlighting the key element of our result: a type rule (PREPAY) that allows costs to be deferred.

Although the examples in this thesis have only considered list and scalar data structures, previous work [JLH$^+$09, JLHH10] suggests that there should be no difficulties in finding examples that successfully deal with other forms of data structures. Also, even though we do not provide a formal guarantee for the predicted bounds of the publicly accessible web-prototype implementation (we have not worked on a proof connecting the analysis to the implementation), examples with other forms of data structures are available and we have confidence on the implementation results based on the similarity to other (three) implementations by Jost [Jos10] (that have been around for over 10 years now) and on the fact that we have used this same tool to help construct the complete type derivations found in Appendix B.

7.2 Limitations and Further Work

There are a number of limitations to the work presented here that would repay further investigation.

Deallocation and other resource metrics: While Jost et al. [JLH$^+$09, JLHH10] have previously constructed analyses that are capable of dealing with arbitrary countable resources for strict languages, for simplicity, in this thesis we have restricted our attention to the heap allocation metric only. There are two obvious ways to extend the analysis to handle deallocation: a *destructive pattern matching operator* similar to the one used in [HJ03] and a *deallocation primitive* such as `free(`ℓ`)`. In either case we would extend the type system with an extra annotation (on thunk types, function types and turnstile) to represent how many heap cells could be reused. For example, a type $\mathsf{T}^{q}_{q'}(A)$ would correspond to an expression that needs at most q heap cells to evaluate and frees at most q' heap cells once evaluated. Supporting deallocation would widen the range of successfully handled examples as seen for example in the analysis of insertion sort with destructive pattern matching [HJ03] that turned an otherwise quadratic heap usage function into a linear one. Once deallocation is supported, the same principle can be used to measure other non-monotonic resource usage such as stack information. Analysing countable resource metrics other than heap usage should then follow a similar structure, but might require a richer operational semantics than that given by Launchbury.

Co-recursive definitions: We would like to tackle the limitation found in Section 6.5. It seems that the cause for failure is related to our type system inability to capture sharing of costs in co-recursive examples such as the ones found in Section 6.5. Our first attempt at solving this problem would be to provide a way to unfold recursive types and see where it would lead us. Failing that, and since a circular definition can successfully reach a *whnf* if it only ever depends on parts of the data that can readily be in *whnf* themselves, we could try to change rule LET in order to allow the free reuse of the recursive binding by having zero on the costs of the thunk types for x, in the typing for $\widehat{e}$. After these attempts, we hope we would have more insight and be better positioned to suggest further alternatives to tackle this current limitation.

Super-linear bounds: It seems possible to combine our analysis for lazy evaluation with recent advances by Hoffmann et al. [HAH11] in the study of super-linear bounds. However, it is not clear how to keep higher-orderness since their system currently only applies to first-order functional languages. Another approach that could be taken (based on a suggestion found in Jost's thesis [Jos10]) is the combination of current sized type and automatic amortised analyses. The idea is to use information from a sized type analysis (such as the length of a list) to recharge the potential (of that same list) on an amortised analysis. Inference of super-linear bounds could still be efficient through the successive application of LP-solving. Note though that, to be effective, this alternative approach would have the extra difficulty of first extending a sized type analysis, since such analyses currently do not handle lazy evaluation.

Interleaving types: The analysis is limited to non-interleaving types [Mat98] , which exclude types such as finitely branching trees $(\mu Y.\{\texttt{FinT}{:}(\mu X.\{\texttt{Cons}{:}(Y,X)\,|\,\texttt{Nil}{:}()\})\})$, but include nested types such as lists of lists and trees of lists and most of the commonly user-defined data types. The requirement of non-interleaving types helps us prove a crucial lemma on cyclic structures (Lemma 5.17) in the key soundness proof (Theorem 5.13). However, this restriction feels arbitrary — we do not know if there is a fundamental reason for such requirement or if it results from our current inability to find a better argument for the proof, although we suspect the latter. In accordance with this intuition, note that the soundness proof of the eager system presented in this thesis also relies on the same lemma (and thus relies on having non-interleaving types only), while no other previous analyses for eager systems do so. We believe that, if removing the limitation of non-interleaving types is possible, it will require different invariants for the soundness proof, since that is the main difference between the eager system presented in this thesis and the previous automatic amortised analyses for eager evaluation.

Non-terminating programs: As Jost [Jos10] observes: "all programs that exceed the available free memory, even non-terminating ones, must do so already after a *finite* number of steps". The technique for handling non-terminating programs as a separate proof treatment in analyses following Hofmann and Jost's approach was introduced by Aspinall et al. [ABH$^+$07] and later used in other analyses [Cam08, Jos10, Hof11]. It consists of extending the core semantics with extra rules that force the termination of programs that exceed some arbitrary prescribed amount of resources during evaluation and then add a soundness theorem for non-terminating programs, stating that if a program is forced to terminate then the prescribed bound is lower than what the analysis predicted. This technique can be straightforwardly applied to our system, extending our analysis to handle non-terminating lazy programs.

Polymorphic functions and resource parametricity: The system we have described is restricted to monomorphic definitions. Jost et al. [JLHH10] describe an amortised analysis for polymorphic, higher-order but strict functions. Also, improving our system to allow function types to be *resource parametric* [JLHH10] would imply that instead of having the type system deal only with rational numbers as type annotations, it would have to deal also with so-called *resource variables* in order to directly manipulate constraints on the annotations. With this convenient feature we would not have to duplicate the code in order to produce the required different type annotations for function map when showing the deforestation benefits in Section 6.2. We believe the same techniques that enable polymorphic functions and resource parametricity can be straightforwardly adapted to a lazy setting.

7.3 Final Remark

Lazy evaluation, in practice being free from side-effects, is particularly well suited for parallelism [THLPJ98, MML$^+$10, KCL$^+$10, MNPJ11], and can offer important advantages in the current era of many-core processors. Recent work [DMMZ12] allows further research on call-by-need to simply focus on one formal system, e.g. a reduction semantics, an abstract machine or a natural semantics, since we can mechanically inter-derive the others. Combined with the continuous improvements on the automatic amortised analysis over the past few years [HJ03, HJ06, Cam08, JLH$^+$09, JLHH10, Jos10, HAH11, SVF$^+$12], we look forward to see our work on bridging lazy evaluation and automatic amortised analysis enjoy increasing success.

Bibliography

[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. 7.2

[ADM04] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. 2.1, 4

[AF97] Zena M. Ariola and Matthias Felleisen. The Call-by-Need Lambda Calculus. *Journal of Functional Programming*, 7:265–301, May 1997. 2.1

[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'09)*, pages 129–138, Dublin, Ireland, June 2009. ACM. 2.4

[AGG10] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Parametric Inference of Memory Requirements for Garbage Collected Languages. In *Proceedings of the International Symposium on Memory Management (ISMM'10)*, pages 121–130, Toronto, Ontario, Canada, June 2010. ACM. 2.4

[ASV03] Elvira Albert, Josep Silva, and Germán Vidal. Time Equations for Lazy Functional (Logic) Languages. In *Proceedings of the Joint Conference on Declarative Programming, AGP-2003*, pages 13–24, Reggio Calabria, Italy, September 2003. 2.2

[Bay72] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972. 3.2.1

[BFGY08] Víctor Braberman, Federico Fernández, Diego Garbervetsky, and Sergio Yovine. Parametric Prediction of Heap Memory Requirements. In *Proceedings of the International Symposium on Memory Management (ISMM'08)*, pages 141–150, Tucson, Arizona, USA, June 2008. ACM. 2.4

[BH89] Bror Bjerner and Sören Holmström. A compositional approach to time analysis of first order lazy functional programs. In *Proceedings of the ACM SIGPLAN Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, London, UK, September 1989. 2.2

[BHA86] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986. 2.1

[BR00] Adam Bakewell and Colin Runciman. A Model for Comparing the Space Usage of Lazy Evaluators. In *Proceedings of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'00)*, pages 151–162, Montreal, Quebec, Canada, September 2000. 2.1

[BR01] Adam Bakewell and Colin Runciman. A Space Semantics for Core Haskell. In Graham Hutton, editor, *ACM SIGPLAN Haskell Workshop 2000*, volume 41 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001. 2.1

[Cam08] Brian Campbell. *Type-based amortized stack memory prediction*. PhD thesis, Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh, UK, 2008. 2.3, 7.2, 7.3

[Cam09] Brian Campbell. Amortised Memory Analysis Using the Depth of Data Structures. In Giuseppe Castagna, editor, *Proceedings of the European Symposium on Programming (ESOP'09), York, UK, March, 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 190–204. Springer, 2009. 2.3, 3.2

[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'08)*, pages 151–160, Tucson, Arizona, USA, June 2008. ACM. 2.4

[CW00] Karl Crary and Stephanie Weirich. Resource Bound Certification. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

Languages (POPL'00), pages 184–198, Boston, Massachusetts, USA, January 2000. 2.3

[Dan08] Nils Anders Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 133–144, San Francisco, California, USA, January 2008. 2.2

[DM82] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'82)*, pages 207–212, Albuquerque, New Mexico, USA, January 1982. 1, 3.2.1

[DMMZ12] Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. On Inter-deriving Small-step and Big-step Semantics: A Case Study for Storeless Call-by-need Evaluation. *Theoretical Computer Science*, 435(0):21–42, 2012. 7.3

[Enn03] Robert Ennals. *Adaptive Evaluation of Non-Strict Programs*. PhD thesis, King's College, University of Cambridge, December 2003. 2.1

[EP02] Alberto de la Encina and Ricardo Peña. Proving the Correctness of the STG Machine. In Thomas Arts and Markus Mohnen, editors, *Selected papers of the International Workshop on Implementation of Functional Languages (IFL'01), Stockholm, Sweden, September, 2001*, volume 2312 of *Lecture Notes in Computer Science*, pages 88–104. Springer, 2002. 2.1, 4, 4.2, †, 4.2, 4.2

[EP03a] Alberto de la Encina and Ricardo Peña. Formally Deriving an STG Machine. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 102–112, Uppsala, Sweden, August 2003. ACM. 2.1, 4

[EP03b] Robert Ennals and Simon Peyton Jones. Optimistic Evaluation: an adaptive evaluation strategy for non-strict programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pages 287–298, Uppsala, Sweden, August 2003. 2.1

[EP09] Alberto de la Encina and Ricardo Peña. From Natural Semantics to C: a Formal Derivation of two STG Machines. *Journal of Functional Programming*, 19(1):47–94, 2009. 2.1, 4.1

[Fax00] Karl-Filip Faxén. Cheap eagerness: Speculative evaluation in a lazy functional language. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 150–161, Montreal, Canada, September 2000. 2.1

[GS99] Jörgen Gustavsson and David Sands. A Foundation for Space-Safe Transformations of Call-by-Need Programs. In Andrew D. Gordon and Andrew M. Pitts, editors, *Third International Workshop on Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999. 2.1

[HAH11] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate Amortized Resource Analysis. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*, pages 357–370, Austin, Texas, USA, January 2011. 1, 2.3, 3.2, 3.2.1, 5.1, 7.2, 7.3

[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 *Proceedings of the 7th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 13–18, Pisa, Italy, July 2007. 2.3

[HH10] Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polynomial Potential. In Giuseppe Castagna, editor, *Proceedings of the European Symposium on Programming (ESOP'10), Paphos, Cyprus, March, 2010*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010. 2.3, 3.2, 3.2.1

[HJ03] Martin Hofmann and Steffen Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 185–197, New Orleans, Louisiana, USA, January 2003. 1, 2.3, 3.2, 7.2, 7.3

[HJ06] Martin Hofmann and Steffen Jost. Type-Based Amortised Heap-Space Analysis. In Peter Sestoft, editor, *Proceedings of the European Symposium on Programming (ESOP'06), Vienna, Austria, March, 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006. 2.3, 7.3

[Hof11] Jan Hoffmann. *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis*. PhD thesis, LMU Munich, Germany, 2011. 7.2

[Hop08] Catherine Hope. *A Functional Semantics for Space and Time*. PhD thesis, University of Nottingham, UK, 2008. 2.2

[HR09] Martin Hofmann and Dulma Rodriguez. Efficient Type-Checking for Amortised Heap-Space Analysis. In *Proceedings of the CSL: Annual Conference of the European Association for Computer Science Logic, Coimbra, Portugal, September, 2009*, volume 5771 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2009. 2.3

[Hug89] John Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, 1989. 1

[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, Eindhoven, The Netherlands, November, 2009*, volume 5850 of *Lecture Notes in Computer Science*, pages 354–369. Springer, 2009. 1, 2.3, 3.2, 4.3, 7.1, 7.2, 7.3

[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 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*, pages 223–236, Madrid, Spain, January 2010. 1, 2.3, 3.2, 5.1, 5.5.5, 6.2, 7.1, 7.2, 7.2, 7.3

[Jon92] Simon Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. *Journal of Functional Programming*, 2(2):127–202, 1992. 4

[Jos89] Mark B. Josephs. The semantics of lazy functional languages. *Theoretical Computer Science*, 68(1):105–111, 1989. 2.1

[Jos10] Steffen Jost. *Automated Amortised Analysis*. PhD thesis, Faculty of Mathematics, Computer Science and Statistics, LMU Munich, Germany, 2010. 2.3, 5.5.5, 7.1, 7.2, 7.2, 7.3

[KCL$^+$10] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'10)*, pages 261–272, Baltimore, Maryland, USA, September 2010. 7.3

[Lau93] John Launchbury. A Natural Semantics for Lazy Evaluation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154, Charleston, South Carolina, USA, January 1993. 1, 1.1, 2.1, 4, 4.1, 4.2, 4.2

[LJ09] Hans-Wolfgang Loidl and Steffen Jost. Improvements to a Resource Analysis for Hume. In *Proceedings of the 1st International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA)*, Eindhoven, The Netherlands, November 2009. Springer. 3.2.1

[Mat98] Ralph Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Induction Types*. PhD thesis, LMU Munich, Germany, 1998. 5.1, 7.2

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978. 1, 3.2.1

[MML$^+$10] Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K. Aswad, and Phil Trinder. Seq no more: Better strategies for parallel haskell. In *Proceedings of the third ACM SIGPLAN Haskell Symposium*, pages 91–102, Baltimore, Maryland, USA, 2010. ACM. 7.3

[MN92] Alan Mycroft and Arthur Norman. Optimising compilation — lazy functional languages. In *Proceedings of the 19th Software Seminar (SOFSEM)*, Ždiar, Czechoslovakia, 1992. 2.1

[MNPJ11] Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the fourth ACM SIGPLAN Haskell Symposium*, pages 71–82, Tokyo, Japan, 2011. ACM. 7.3

[Mou98] Jon Mountjoy. The Spineless Tagless G-machine, naturally. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 163–173, Baltimore, Maryland, USA, September 1998. 2.1

[MOW98] John Maraist, Martin Odersky, and Philip Wadler. The Call-by-Need Lambda Calculus. *Journal of Functional Programming*, 8:275–317, May 1998. 2.1

[MS99] Andrew Moran and David Sands. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages 43–56, San Antonio, Texas, USA, January 1999. 2.1

[MT91] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991. 5.5.5

[Myc80] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the International Symposium on Programming, Paris, France, April, 1980*, volume 83 of *Lecture Notes in Computer Science*, pages 269–281. Springer, 1980. 2.1

[Myc81] Alan Mycroft. *Abstract interpretation and optimising transformations for applicative programs*. PhD thesis, Department of Computer Science, University of Edinburgh, UK, 1981. 2.1

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998. 2.3, 3.2

[PAB$^+$99] Simon Peyton Jones (editor), Lennart Augustsson, Brian Boutel, F. Warren Burton, Joseph H. Fasel, Andrew D. Gordon, Kevin Hammond, John Hughes, Paul Hudak, Thomas Johnsson, Mark P. Jones, John C. Peterson, Alastair Reid, and Philip Wadler. Report on the Non-Strict Functional Language, Haskell (Haskell98). Technical report, Yale University, 1999. 1

[PB10] Maciej Pirog and Dariusz Biernacki. A Systematic Derivation of the STG Machine Verified in Coq. In *Proceedings of the third ACM SIGPLAN Haskell Symposium*, pages 25–36, Baltimore, Maryland, USA, 2010. ACM. 2.1, 4

[Rey72] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM National Conference*, pages 717–740. ACM, August 1972. 2.2

[San90a] David Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, Imperial College, University of London, September 1990. 2.2

[San90b] David Sands. Complexity Analysis for a Lazy Higher-Order Language. In Neil Jones, editor, *Proceedings of the European Symposium on Programming (ESOP'90), Copenhagen, Denmark, May, 1990*, volume 432 of *Lecture Notes in Computer Science*, pages 361–376. Springer, 1990. 2.2

[San98] David Sands. Computing with contexts: A simple approach. In Andrew D. Gordon, Andrew M. Pitts, and Carolyn L. Talcott, editors, *Second Workshop on Higher-Order Operational Techniques in Semantics*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. 2.2

[Ses97] Peter Sestoft. Deriving a Lazy Abstract Machine. *Journal of Functional Programming*, 7(3):231–264, 1997. 2.1, 4, 4.1, 4.2, ∗, 4.2

[SHFV07] Hugo R. Simões, Kevin Hammond, Mário Florido, and Pedro Vasconcelos. Using Intersection Types for Cost-Analysis of Higher-Order Polymorphic Functional Programs. In Thorsten Altenkirch and Conor McBride, editors, *Revised Selected Papers of the International Workshop on Types for Proofs and Programs (TYPES'06), Nottingham, UK, April, 2006*, volume 4502 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2007. 1, 1.1

[SVF$^+$12] Hugo Simões, Pedro Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'12)*, pages 165–176, Copenhagen, Denmark, September 2012. 1.1, 2.3, 5.3, 7.3

[Svv07] Olha Shkaravska, Ron van Kesteren, and Marko van Eekelen. Polynomial Size Analysis of First-Order Functions. In *Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications (TLCA'07), Paris, France, June, 2007*, volume 4583 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2007. 2.3

[Tar85] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985. 2.3, 3.1, ∗, 3.2

[THLPJ98] Phil W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon Peyton Jones. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998. 7.3

[Vas08] Pedro Baltazar Vasconcelos. *Space cost analysis using sized types*. PhD thesis, School of Computer Science, University of St Andrews, November 2008. 1

[VH05] Pedro B. Vasconcelos and Kevin Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In Phil Trinder, Greg J. Michaelson, and Ricardo Peña, editors, *Revised Papers of the International Workshop on Implementation of Functional Languages (IFL'03), Edinburgh, UK, September, 2003*, volume 3145 of *Lecture Notes in Computer Science*, pages 88–101. Springer, 2005. 1

[Wad88] Philip Wadler. Strictness Analysis aids Time Analysis. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'88)*, pages 119–132, San Diego, California, USA, January 1988. 2.2

[Wad92] Philip Wadler. The Essence of Functional Programming. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92)*, pages 1–14, Albuquerque, New Mexico, USA, January 1992. 2.2

[WH87] Philip Wadler and John Hughes. Projections for Strictness Analysis. In *Proceedings of the ACM SIGPLAN Conference on Functional Programming Languages and Computer Architecture (FPCA'87)*, Portland, Oregon, USA, September 1987. 2.1, 2.2

A. A System for Eager Evaluation

A.1 Definitions and Figures

$$\frac{w \text{ is in } \textit{whnf}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash w \Downarrow w, \mathcal{H}} \qquad (\textsc{Whnf}_{\Downarrow})$$

$$\frac{\ell \notin \mathcal{L} \qquad \mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash \mathcal{H}(\ell) \Downarrow w, \mathcal{H}'}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \ell \Downarrow w, \mathcal{H}'[\ell \mapsto w]} \qquad (\textsc{Var}_{\Downarrow})$$

$$\frac{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash e \Downarrow \lambda x.\, e', \mathcal{H}' \qquad \mathcal{H}', \mathcal{S}, \mathcal{L} \vdash e'[\ell/x] \Downarrow w, \mathcal{H}''}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash e\,\ell \Downarrow w, \mathcal{H}''} \qquad (\textsc{App}_{\Downarrow})$$

$$\frac{\begin{array}{c} \ell \text{ is fresh} \qquad \mathcal{H}\big[\ell \mapsto \widehat{e}[\ell/x]\big], \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash \widehat{e}[\ell/x] \Downarrow w', \mathcal{H}' \\ \mathcal{H}'[\ell \mapsto w'], \mathcal{S}, \mathcal{L} \vdash e[\ell/x] \Downarrow w, \mathcal{H}'' \end{array}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \text{let } x = \widehat{e} \text{ in } e \Downarrow w, \mathcal{H}''} \qquad (\textsc{EagerLet}_{\Downarrow\mathrm{C}})$$

$$\frac{\begin{array}{c} \mathcal{H}, \mathcal{S} \cup \bigcup_{i=1}^{n} \left(\{\overrightarrow{x_i}\} \cup \mathrm{BV}(e_i)\right), \mathcal{L} \vdash e \Downarrow c_k\, \vec{\ell}, \mathcal{H}' \\ \mathcal{H}', \mathcal{S}, \mathcal{L} \vdash e_k[\vec{\ell}/\overrightarrow{x_k}] \Downarrow w, \mathcal{H}'' \end{array}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash \text{case } e \text{ of } \{c_i\ \overrightarrow{x_i} \text{ -> } e_i\}_{i=1}^{n} \Downarrow w, \mathcal{H}''} \qquad (\textsc{Case}_{\Downarrow})$$

Figure A.1: Eager operational semantics

$$\frac{w \text{ is in } \textit{whnf}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{0} w \Downarrow w, \mathcal{H}} \tag{$\text{WHNF}_{\Downarrow\text{C}}$}$$

$$\frac{\ell \notin \mathcal{L} \qquad \mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash^{m} \mathcal{H}(\ell) \Downarrow w, \mathcal{H}'}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m} \ell \Downarrow w, \mathcal{H}'[\ell \mapsto w]} \tag{$\text{VAR}_{\Downarrow\text{C}}$}$$

$$\frac{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m} e \Downarrow \lambda x.\, e', \mathcal{H}' \qquad \mathcal{H}', \mathcal{S}, \mathcal{L} \vdash^{m'} e'[\ell/x] \Downarrow w, \mathcal{H}''}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m + m'} e\,\ell \Downarrow w, \mathcal{H}''} \tag{$\text{APP}_{\Downarrow\text{C}}$}$$

$$\frac{\begin{array}{c}\ell \text{ is fresh} \qquad \mathcal{H}\big[\ell \mapsto \widehat{e}[\ell/x]\big], \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash^{m'} \widehat{e}[\ell/x] \Downarrow w', \mathcal{H}' \\ \mathcal{H}'[\ell \mapsto w'], \mathcal{S}, \mathcal{L} \vdash^{m} e[\ell/x] \Downarrow w, \mathcal{H}''\end{array}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{1 + m' + m} \text{let } x = \widehat{e} \text{ in } e \Downarrow w, \mathcal{H}''} \tag{$\text{EAGERLET}_{\Downarrow\text{C}}$}$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{S} \cup \bigcup_{i=1}^{n} \left(\{\overrightarrow{x_i}\} \cup \mathrm{BV}(e_i)\right), \mathcal{L} \vdash^{m} e \Downarrow c_k\, \vec{\ell}, \mathcal{H}' \\ \mathcal{H}', \mathcal{S}, \mathcal{L} \vdash^{m'} e_k[\vec{\ell}/\overrightarrow{x_k}] \Downarrow w, \mathcal{H}''\end{array}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m + m'} \text{case } e \text{ of } \{c_i\ \overrightarrow{x_i} \text{ -> } e_i\}_{i=1}^{n} \Downarrow w, \mathcal{H}''} \tag{$\text{CASE}_{\Downarrow\text{C}}$}$$

Figure A.2: Cost-instrumented eager operational semantics

$$\begin{array}{llll} A, B, C & ::= & X & \text{– type variable} \\ & \mid & A \xrightarrow{q} B & \text{– function type} \\ & \mid & \mu X.\{c_1 : (p_1, \vec{B}_1) \mid \cdots \mid c_n : (p_n, \vec{B}_n)\} & \text{– data type} \end{array}$$

with $q, p_1, \ldots, p_n \in \mathbb{Q}_0^+$

Figure A.3: Annotated types

$$\frac{}{\curlyvee(A \mid \emptyset)} \tag{SHAREEMPTY}$$

$$\frac{}{\curlyvee(X \mid X, \ldots, X)} \tag{SHAREVAR}$$

$$\frac{\begin{array}{c} B_i = \mu X.\big\{c_1 : (p'_{i1}, \vec{B}_{i1}) \mid \cdots \mid c_m : (p'_{im}, \vec{B}_{im})\big\} \\ \curlyvee\Big(\vec{A}_j \,\Big|\, \vec{B}_{1j}, \ldots, \vec{B}_{nj}\Big) \qquad p_j \geq \sum_{i=1}^{n} p'_{ij} \qquad (1 \leq i \leq n,\ 1 \leq j \leq m) \end{array}}{\curlyvee\Big(\mu X.\big\{c_1 : (p_1, \vec{A}_1) \mid \cdots \mid c_m : (p_m, \vec{A}_m)\big\} \mid B_1, \ldots, B_n\Big)} \tag{SHAREDAT}$$

$$\frac{\curlyvee(A_i \mid A) \qquad \curlyvee(B \mid B_i) \qquad q_i \geq q \qquad (1 \leq i \leq n)}{\curlyvee\big(A \xrightarrow{q} B \,\big|\, A_1 \xrightarrow{q_1} B_1, \ldots, A_n \xrightarrow{q_n} B_n\big)} \tag{SHAREFUN}$$

$$\frac{\curlyvee(A_j \mid B_{1j}, \ldots, B_{nj}) \qquad m = \left|\vec{A}\right| = \left|\vec{B}_i\right| \qquad (1 \leq i \leq n,\ 1 \leq j \leq m)}{\curlyvee\Big(\vec{A} \,\Big|\, \vec{B}_1, \ldots, \vec{B}_n\Big)} \tag{SHAREVEC}$$

Figure A.4: Sharing relation

$$\frac{\begin{array}{c}\Gamma, x{:}A' \vdash^{q'} \widehat{e} : A \qquad \Delta, x{:}A \vdash^{q} e : C \\ x \notin \mathrm{dom}(\Gamma, \Delta) \qquad \curlyvee(A \,|\, A, A') \qquad q' = 0 \text{ if } \widehat{e} \equiv c\,\vec{y} \text{ or } \widehat{e} \equiv \lambda y.e' \\ p = \begin{cases} p', \text{ if } \widehat{e} \equiv c\,\vec{y} \text{ and } A = \mu X.\{\cdots \mid c : (p', \vec{B}) \mid \cdots\} \\ 0, \text{ otherwise} \end{cases}\end{array}}{\Gamma, \Delta \vdash^{1 + q' + q + p} \text{let } x = \widehat{e} \text{ in } e : C} \quad (\textsc{EagerLet})$$

$$\frac{}{x{:}A \vdash^{0} x : A} \quad (\textsc{EagerVar})$$

$$\frac{\Gamma, x{:}A \vdash^{q} e : C \qquad x \notin \mathrm{dom}(\Gamma) \qquad \curlyvee(\Gamma \,|\, \Gamma, \Gamma)}{\Gamma \vdash^{0} \lambda x.e : A \xrightarrow{q} C} \quad (\textsc{Abs})$$

$$\frac{\Gamma \vdash^{q} e : A \xrightarrow{q'} C}{\Gamma, y{:}A \vdash^{q + q'} e\,y : C} \quad (\textsc{App})$$

$$\frac{B = \mu X.\{\cdots \mid c : (p, \vec{A}) \mid \cdots\}}{y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X] \vdash^{0} c\,\vec{y} : B} \quad (\textsc{Cons})$$

$$\frac{\begin{array}{c}\Gamma \vdash^{q} e : B \qquad B = \mu X.\{c_1 : (p_1, \overrightarrow{A_1}) \mid \cdots \mid c_n : (p_n, \overrightarrow{A_n})\} \\ (\bigcup_{i=1}^{n}\{\overrightarrow{x_i}\}) \cap \mathrm{dom}(\Delta) = \emptyset \\ i = 1, \ldots, n \begin{cases} |\overrightarrow{A_i}| = |\overrightarrow{x_i}| = k_i \\ \Delta, x_{i_1}{:}A_{i_1}[B/X], \ldots, x_{i_{k_i}}{:}A_{i_{k_i}}[B/X] \vdash^{q' + p_i} e_i : C \end{cases}\end{array}}{\Gamma, \Delta \vdash^{q + q'} \text{case } e \text{ of } \{c_i\ \overrightarrow{x_i} \text{ -> } e_i\}_{i=1}^{n} : C} \quad (\textsc{Case})$$

Figure A.5: Syntax directed type rules

$$\frac{\Gamma \vdash^{q} e : C \qquad \curlyvee(A' \,|\, (\Gamma, x{:}A){\restriction_x})}{\Gamma, x{:}A \vdash^{q} e : C} \quad (\textsc{Weak})$$

$$\frac{\Gamma \vdash^{q'} e : A \qquad q \geq q'}{\Gamma \vdash^{q} e : A} \quad (\textsc{Relax})$$

$$\frac{\Gamma, x{:}B \vdash^{q} e : C \qquad A <: B}{\Gamma, x{:}A \vdash^{q} e : C} \quad (\textsc{Supertype})$$

$$\frac{\Gamma \vdash^{q} e : B \qquad B <: C}{\Gamma \vdash^{q} e : C} \quad (\textsc{Subtype})$$

$$\frac{\Gamma, x{:}A_1, x{:}A_2 \vdash^{q} e : C \qquad \curlyvee(A \,|\, A_1, A_2)}{\Gamma, x{:}A \vdash^{q} e : C} \quad (\textsc{Share})$$

Figure A.6: Structural type rules

$$\phi_{\mathcal{H}}(\widehat{e}{:}A) \overset{\text{def}}{=} \begin{cases} p + \sum_i \phi_{\mathcal{H}}(\mathcal{H}(\ell_i){:}B_i[A/X]) & \text{if } A = \mu X.\{\cdots \mid c{:}(p, \vec{B}) \mid \cdots\} \text{ and } \widehat{e} = c\,\vec{\ell} \\ 0 & \text{otherwise} \end{cases} \tag{A.1}$$

$$\phi_{\mathcal{H}}(\Gamma) \overset{\text{def}}{=} \sum \{\phi_{\mathcal{H}}(\mathcal{H}(x){:}A) \mid x{:}A \in \Gamma\} \tag{A.2}$$

Figure A.7: Potential

Definition A.1 (Type Consistency of Locations)**.** We say that location ℓ admits type A under context Γ and heap configuration $(\mathcal{H}, \mathcal{L})$, and write $\Gamma; \mathcal{H}, \mathcal{L} \vdash_{\textsc{Loc}} \ell : A$, if one of the following cases holds:

(Loc1) $\mathcal{H}(\ell)$ is in *whnf* and $\Gamma \overset{0}{\vdash} \mathcal{H}(\ell) : A$

(Loc3) $\mathcal{H}(\ell)$ not in *whnf* and $\ell \in \mathcal{L}$ and $\Gamma = \emptyset$

Definition A.2 (Type Consistency of Heaps)**.** We say that a heap state $(\mathcal{H}, \mathcal{L})$ is consistent with global contexts and global types, and write $\mathcal{C} \vdash_{\textsc{Mem}} (\mathcal{H}, \mathcal{L}) : \mathcal{M}$, if and only if for all $\ell \in \mathrm{dom}(\mathcal{H})$: $\mathcal{C}(\ell); \mathcal{H}, \mathcal{L} \vdash_{\textsc{Loc}} \ell : \mathcal{M}(\ell)$ holds.

A.2 Proof of the Soundness Theorem for the Eager System

The proof of Theorem 5.18 follows by induction on the lengths of the derivations of (5.98) and (5.97) ordered lexicographically, with the derivation of the evaluation taking priority over the typing derivation. We proceed by case analysis of the typing rule used in premise (5.97), considering just the case EagerLet since the remaining cases are similar to (or simpler than) the ones presented in the soundness proof of the lazy system (in Section 5.5.6.7).

Case EagerLet**:** We start with hypothesis

$$\Gamma, \Delta \overset{1+q'+q+p}{\vdash\!\!\!-\!\!\!-\!\!\!-\!\!\!-} \text{let } x = \widehat{e} \text{ in } e : C \text{ (5.97)}$$

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \longmapsto \text{let } x = \widehat{e} \text{ in } e \Downarrow w, \mathcal{H}'' \text{ (5.98)}$$

$$\mathcal{C} \vdash_{\textsc{Mem}} (\mathcal{H}, \mathcal{L}) : \mathcal{M} \text{ (5.99)}$$

$$\curlyvee(\mathcal{M} \mid (\Gamma, \Delta, \Theta), \mathcal{C}) \text{ (5.100)}$$

Applying Lemma 5.2 (Substitution) to the premises of rule EAGERLET (5.97) we obtain

$$\Gamma, \ell{:}A' \overset{q'}{\vdash} \widehat{e}[\ell/x] : A \tag{A.3}$$

$$\Delta, \ell{:}A \overset{q}{\vdash} e[\ell/x] : C \tag{A.4}$$

The premises of rule EAGERLET$_{\Downarrow}$ (5.98) instantiate as

$$\mathcal{H}_1, \mathcal{S}, \mathcal{L}_1 \vdash \widehat{e}[\ell/x] \Downarrow w', \mathcal{H}' \tag{A.5}$$

$$\mathcal{H}_2, \mathcal{S}, \mathcal{L} \vdash e[\ell/x] \Downarrow w, \mathcal{H}'' \tag{A.6}$$

where $\mathcal{L}_1 = \mathcal{L} \cup \{\ell\}$, $\mathcal{H}_1 = \mathcal{H}\big[\ell \mapsto \widehat{e}[\ell/x]\big]$, $\mathcal{H}_2 = \mathcal{H}'[\ell \mapsto w']$ and ℓ is a suitably fresh location (without loss of generality we can assume not only that ℓ is fresh by Definition 4.2 w.r.t. (5.98) but also that ℓ does not occur in Γ, Δ, Θ, $\mathcal{C}$ nor $\mathcal{M}$).

Expression $\widehat{e}[\ell/x]$ is either not in *whnf* or is in *whnf*. Also, if in *whnf*, expression $\widehat{e}[\ell/x]$ is either a constructor application or a λ-abstraction. We proceed by considering each of these mutually exclusive cases separately.

If $\widehat{e}[\ell/x]$ is not in *whnf*: We intend to apply the induction hypothesis twice, so we must establish the required premises first.

To apply the induction hypothesis over the term $\widehat{e}[\ell/x]$, let $\mathcal{C}_1 = \mathcal{C}[\ell \mapsto \emptyset]$ and $\mathcal{M}_1 = \mathcal{M}[\ell \mapsto A']$, for some idempotent type A' with $\curlyvee(A \,|\, A, A')$ provided by the premises of (5.97).

Type consistency is extended to $\mathcal{C}_1 \vdash_{\text{MEM}} (\mathcal{H}_1, \mathcal{L}_1) : \mathcal{M}_1$ by case (LOC3) of Definition A.2.

Compatibility $\curlyvee(\mathcal{M}_1 \,|\, (\Gamma, \ell{:}A', \Delta, \Theta), \mathcal{C}_1)$ follows from (5.100), ℓ being suitably fresh and Definition 5.12 (Global Compatibility).

From premise $m \geq t + 1 + q' + q + p + \phi_{\mathcal{H}}(\Gamma, \Delta) + \phi_{\mathcal{H}}(\Theta)$ (5.101) we derive $m_1 \geq (t + 1 + q + p) + q' + \phi_{\mathcal{H}_1}(\Gamma, \ell{:}A') + \phi_{\mathcal{H}_1}(\Delta, \Theta)$, observing that $\phi_{\mathcal{H}}(\Gamma, \Delta, \Theta) = \phi_{\mathcal{H}_1}(\Gamma, \Delta, \Theta)$ (since ℓ is suitably fresh) and $p = 0 = \phi_{\mathcal{H}_1}(\ell{:}A')$ (since $\widehat{e}[\ell/x]$ is not in *whnf*).

We can now apply the first induction hypothesis, obtaining $m_1', \Gamma_1', \mathcal{C}_1', \mathcal{M}_1'$ and m_1'' such that:

$$\Gamma_1' \overset{0}{\vdash} w' : A \tag{A.7}$$

$$\mathcal{H}_1, \mathcal{S}, \mathcal{L}_1 \overset{m_1''}{\vdash} \widehat{e}[\ell/x] \Downarrow w', \mathcal{H}' \tag{A.8}$$

$$\mathcal{M}_1 <: \mathcal{M}_1' \tag{A.9}$$

$$\mathcal{C}'_1 \vdash_{\textsf{MEM}} (\mathcal{H}', \mathcal{L}_1) : \mathcal{M}'_1 \tag{A.10}$$

$$\curlyvee(\mathcal{M}'_1 \,|\, (\Gamma'_1, \Delta, \Theta), \mathcal{C}'_1\,) \tag{A.11}$$

$$m'_1 \geq (t + 1 + q + p) + \phi_{\mathcal{H}'}\big(w'{:}A\big) + \phi_{\mathcal{H}'}(\Delta, \Theta) \tag{A.12}$$

$$m_1 - m'_1 \geq m''_1 \tag{A.13}$$

In order to extend type consistency (A.10) to $\mathcal{H}_2$, let $\mathcal{C}_2 = \mathcal{C}'_1[\ell \mapsto \Gamma'_1]$ and $\mathcal{M}_2 = \mathcal{M}'_1[\ell \mapsto A]$. Note that $\mathcal{C}'_1(\ell) = \emptyset$ from case (Loc3) of (A.10) and that $\mathcal{M}'_1(\ell) = A'$ from (A.9) and the definition of $\mathcal{M}_1$. From (A.7) and case (Loc1) of Definition A.2 (Type Consistency of Heaps) $\mathcal{C}_2 \vdash_{\textsf{MEM}} (\mathcal{H}_2, \mathcal{L}) : \mathcal{M}_2$ holds.

From (A.11) we extend global compatibility to $\curlyvee(\mathcal{M}_2 \,|\, (\Delta, \ell{:}A, \Theta), \mathcal{C}_2\,)$ which holds by Definition 5.12 since eventual types $(\Delta, \Theta)\restriction_\ell \cup \mathcal{C}_2\restriction_\ell$ are idempotent (A.9).

From (A.12) we derive $m_2 \geq (t{+}1{+}p){+}q{+}\phi_{\mathcal{H}_2}(\Delta, \ell{:}A){+}\phi_{\mathcal{H}_2}(\Theta)$, observing that $\phi_{\mathcal{H}'}(w'{:}A){+}\phi_{\mathcal{H}'}(\Delta, \Theta) = \phi_{\mathcal{H}_2}(w'{:}A){+}\phi_{\mathcal{H}_2}(\Delta, \Theta)$ (if the update $\mathcal{H}_2$ introduced new cycles we would apply Lemma 5.17 (Idempotent Cycles) and, since any new cycles must include the updated location ℓ, this would imply type A is idempotent and the potential of idempotent types $\phi_{\mathcal{H}_2}((\Delta, \Theta)\restriction_\ell\,)$ is zero) and that $\phi_{\mathcal{H}_2}(w'{:}A) = \phi_{\mathcal{H}_2}(\ell{:}A)$ (by Figure A.7 (Potential)).

We have all the premises required to apply the second induction hypothesis, obtaining $m'_2, \Gamma'_2, \mathcal{C}'_2, \mathcal{M}'_2$ and m''_2 such that:

$$\Gamma'_2 \overset{0}{\vdash} w : C \tag{A.14}$$

$$\mathcal{H}_2, \mathcal{S}, \mathcal{L} \overset{m''_2}{\vdash} e[\ell/x] \Downarrow w, \mathcal{H}'' \tag{A.15}$$

$$\mathcal{M}_2 <: \mathcal{M}'_2 \tag{A.16}$$

$$\mathcal{C}'_2 \vdash_{\textsf{MEM}} (\mathcal{H}'', \mathcal{L}) : \mathcal{M}'_2 \tag{A.17}$$

$$\curlyvee(\mathcal{M}'_2 \,|\, (\Gamma'_2, \Theta), \mathcal{C}'_2\,) \tag{A.18}$$

$$m'_2 \geq (t + 1 + p) + \phi_{\mathcal{H}''}(w{:}C) + \phi_{\mathcal{H}''}(\Theta) \tag{A.19}$$

$$m_2 - m'_2 \geq m''_2 \tag{A.20}$$

Let $\Gamma' = \Gamma'_2$, $\mathcal{M}' = \mathcal{M}'_2$ and $\mathcal{C}' = \mathcal{C}'_2$. Equations (A.14) (A.17) (A.18) directly establish the proof obligations (5.102) (5.105) (5.106) respectively.

Conclusion (5.104) follows by (A.16) and the transitivity of subtyping.

By applying rule $\textsc{EagerLet}_{\Downarrow\mathsf{C}}$ with premises (A.8), (A.15) and l being fresh, we establish proof obligation (5.103), yielding

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \xmapsto{1 + m_1'' + m_2''} \text{let } x = \widehat{e} \text{ in } e \Downarrow w, \mathcal{H}''$$

If we choose $m' = t + \phi_{\mathcal{H}''}(w{:}C) + \phi_{\mathcal{H}''}(\Theta)$ all we need to complete the proof of case EagerLet is to show that $m - m' \geq 1 + (m_1 - m_1') + (m_2 - m_2')$ ($\geq 1 + m_1'' + m_2'' = m''$).

$$\begin{aligned}
& m - m' \geq 1 + (m_1 - m_1') + (m_2 - m_2') \\
\iff & m - t - \phi_{\mathcal{H}''}(w{:}C) - \phi_{\mathcal{H}''}(\Theta) \geq 1 + m_1 - m_1' + m_2 - t - 1 - p - \phi_{\mathcal{H}''}(w{:}C) - \phi_{\mathcal{H}''}(\Theta) \\
\iff & \qquad \{\, p = 0 \text{ by premise of EagerLet (5.97), since } \widehat{e}[\ell/x] \text{ is not in } \textit{whnf} \,\} \\
& m \geq m_1 - m_1' + m_2 \\
\iff & t + 1 + q' + q + p + \phi_{\mathcal{H}}(\Gamma, \Delta) + \phi_{\mathcal{H}}(\Theta) \\
& \geq t + 1 + q + p + q' + \phi_{\mathcal{H}_1}\big(\Gamma, \ell{:}A'\big) + \phi_{\mathcal{H}_1}(\Delta, \Theta) - m_1' + m_2 \\
\iff & \phi_{\mathcal{H}}(\Gamma, \Delta) + \phi_{\mathcal{H}}(\Theta) \geq \phi_{\mathcal{H}_1}\big(\Gamma, \ell{:}A'\big) + \phi_{\mathcal{H}_1}(\Delta, \Theta) - m_1' + m_2 \\
\iff & \qquad \{\, \phi_{\mathcal{H}_1}(\ell{:}A') = 0 \text{ since type } A' \text{ is idempotent by premise of EagerLet (5.97)} \,\} \\
& \phi_{\mathcal{H}}(\Gamma, \Delta) + \phi_{\mathcal{H}}(\Theta) \geq \phi_{\mathcal{H}_1}(\Gamma) + \phi_{\mathcal{H}_1}(\Delta, \Theta) - m_1' + m_2 \\
\iff & \qquad \{\, \phi_{\mathcal{H}}(\Gamma, \Delta, \Theta) = \phi_{\mathcal{H}_1}(\Gamma, \Delta, \Theta) \text{ since } \ell \text{ is suitably fresh} \,\} \\
& m_1' \geq m_2 \\
\iff & t + 1 + q + p + \phi_{\mathcal{H}'}\big(w'{:}A\big) + \phi_{\mathcal{H}'}(\Delta, \Theta) \geq t + 1 + p + q + \phi_{\mathcal{H}_2}(\Delta, \ell{:}A) + \phi_{\mathcal{H}_2}(\Theta) \\
\iff & \phi_{\mathcal{H}'}\big(w'{:}A\big) + \phi_{\mathcal{H}'}(\Delta, \Theta) \geq \phi_{\mathcal{H}_2}(\Delta, \ell{:}A) + \phi_{\mathcal{H}_2}(\Theta) \\
\iff & \qquad \{\, \phi_{\mathcal{H}'}(w'{:}A) + \phi_{\mathcal{H}'}(\Delta, \Theta) = \phi_{\mathcal{H}_2}(w'{:}A) + \phi_{\mathcal{H}_2}(\Delta, \Theta) \text{ by Lemma 5.17 (Id. Cycles)} \,\} \\
& \phi_{\mathcal{H}_2}\big(w'{:}A\big) + \phi_{\mathcal{H}_2}(\Delta, \Theta) \geq \phi_{\mathcal{H}_2}(\Delta, \ell{:}A) + \phi_{\mathcal{H}_2}(\Theta) \\
\iff & \phi_{\mathcal{H}_2}\big(w'{:}A\big) \geq \phi_{\mathcal{H}_2}(\ell{:}A)
\end{aligned}$$

This last inequality is in fact an equality since $\phi_{\mathcal{H}_2}(\ell{:}A) = \phi_{\mathcal{H}_2}(w'{:}A)$ by the definition of potential (Figure A.7).

This concludes the proof of case EagerLet when $\widehat{e}[\ell/x]$ is not in *whnf*.

If $\widehat{e}[\ell/x]$ is in *whnf*: Evaluation (A.5) terminates immediately by $\text{W}{\small\text{HNF}}_{\Downarrow}$ and we have $w' = \widehat{e}[\ell/x]$ and $\mathcal{H}_1 = \mathcal{H}' = \mathcal{H}_2$. We use rule $\text{W}{\small\text{EAK}}_{\Downarrow\text{C}}$ to obtain

$$\mathcal{H}_1, \mathcal{S}, \mathcal{L}_1 \overset{0}{\vdash} \widehat{e}[\ell/x] \Downarrow w', \mathcal{H}' \tag{A.21}$$

We intend to apply the induction hypothesis over the term $e[\ell/x]$, so we must establish the required premises first.

Let $\mathcal{C}_2 = \mathcal{C}[\ell \mapsto \Gamma, \ell{:}A']$ and $\mathcal{M}_2 = \mathcal{M}[\ell \mapsto A]$.

Type consistency (5.99) is extended to $\mathcal{C}_2 \vdash_{\text{MEM}} (\mathcal{H}_2, \mathcal{L}) : \mathcal{M}_2$ by case (Loc1) of Definition A.2, using (A.3) and the fact that $q' = 0$ from premise of rule EagerLet (5.97).

Compatibility $\curlyvee(\mathcal{M}_2 \,|\, (\Delta, \ell{:}A, \Theta), \mathcal{C}_2\,)$ follows from (5.100), ℓ being suitably fresh, $\curlyvee(A \,|\, A, A'\,)$ from premise of rule EagerLet (5.97), and Definition 5.12 (Global Compatibility).

Since expression $\widehat{e}[\ell/x]$ is in *whnf*, it can either be a constructor application or a λ-abstraction. We now consider each case separately.

If $\widehat{e}[\ell/x]$ is in *whnf* and $\widehat{e}[\ell/x] \equiv c\ \vec{y}$: From premise $m \geq t + 1 + q' + q + p + \phi_{\mathcal{H}}(\Gamma, \Delta) + \phi_{\mathcal{H}}(\Theta)$ (5.101) we want to derive $m_2 \geq (t + 1 + q') + q + \phi_{\mathcal{H}_2}(\Delta, \ell{:}A) + \phi_{\mathcal{H}_2}(\Theta)$ and for that purpose we have to show $t + 1 + q' + q + p + \phi_{\mathcal{H}}(\Gamma, \Delta) + \phi_{\mathcal{H}}(\Theta) \geq (t + 1 + q') + q + \phi_{\mathcal{H}_2}(\Delta, \ell{:}A) + \phi_{\mathcal{H}_2}(\Theta)$, or equivalently $p + \phi_{\mathcal{H}}(\Gamma, \Delta) + \phi_{\mathcal{H}}(\Theta) \geq \phi_{\mathcal{H}_2}(\Delta, \ell{:}A) + \phi_{\mathcal{H}_2}(\Theta)$. First note that $\phi_{\mathcal{H}}(\Gamma, \Delta, \Theta) = \phi_{\mathcal{H}_2}(\Gamma, \Delta, \Theta)$ since ℓ is suitably fresh, and thus we just have to show $p + \phi_{\mathcal{H}_2}(\Gamma) \geq \phi_{\mathcal{H}_2}(\ell{:}A)$. Since type A' is idempotent by premise of EagerLet (5.97), we have $\phi_{\mathcal{H}_2}(\ell{:}A') = 0$ and thus $p + \phi_{\mathcal{H}_2}(\Gamma) = p + \phi_{\mathcal{H}_2}(\Gamma, \ell{:}A')$. From Lemma 5.3 (Cons Inversion) applied to (A.3) we obtain $\curlyvee(\Gamma, \ell{:}A' \,|\, y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X]\,)$. By Lemma 5.7 generalised to contexts, we then have $\phi_{\mathcal{H}_2}(\Gamma, \ell{:}A') \geq \phi_{\mathcal{H}_2}(y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X])$ and thus $p + \phi_{\mathcal{H}_2}(\Gamma, \ell{:}A') \geq p + \phi_{\mathcal{H}_2}(y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X])$. Finally, by the definition of potential (Figure A.7) we have $\phi_{\mathcal{H}_2}(\ell{:}A) = p + \phi_{\mathcal{H}_2}(y_1{:}A_1[B/X], \ldots, y_k{:}A_k[B/X])$ and thus obtain what was needed to prove $p + \phi_{\mathcal{H}_2}(\Gamma) \geq \phi_{\mathcal{H}_2}(\ell{:}A)$.

We have all the premises required to apply the induction hypothesis over $e[\ell/x]$, obtaining $m_2', \Gamma_2', \mathcal{C}_2', \mathcal{M}_2'$ and m_2'' such that:

$$\Gamma_2' \overset{0}{\vdash} w : C \tag{A.22}$$

$$\mathcal{H}_2, \mathcal{S}, \mathcal{L} \overset{m_2''}{\vdash} e[\ell/x] \Downarrow w, \mathcal{H}'' \tag{A.23}$$

$$\mathcal{M}_2 <: \mathcal{M}'_2 \tag{A.24}$$

$$\mathcal{C}'_2 \vdash_{\text{MEM}} (\mathcal{H}'', \mathcal{L}) : \mathcal{M}'_2 \tag{A.25}$$

$$\curlyvee(\mathcal{M}'_2 \,|\, (\Gamma'_2, \Theta), \mathcal{C}'_2\,) \tag{A.26}$$

$$m'_2 \geq (t + 1 + q') + \phi_{\mathcal{H}''}(w{:}C) + \phi_{\mathcal{H}''}(\Theta) \tag{A.27}$$

$$m_2 - m'_2 \geq m''_2 \tag{A.28}$$

Let $\Gamma' = \Gamma'_2$, $\mathcal{M}' = \mathcal{M}'_2$ and $\mathcal{C}' = \mathcal{C}'_2$. Equations (A.22) (A.25) (A.26) directly establish the proof obligations (5.102) (5.105) (5.106) respectively.

Conclusion (5.104) follows by (A.24) and the transitivity of subtyping.

By applying rule $\textsc{EagerLet}_{\Downarrow\text{C}}$ with premises (A.21), (A.23) and l being fresh, we establish proof obligation (5.103), yielding

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \;\vdash^{1 + m''_2}\; \text{let } x = \widehat{e} \text{ in } e \Downarrow w, \mathcal{H}''$$

If we choose $m' = t + \phi_{\mathcal{H}''}(w{:}C) + \phi_{\mathcal{H}''}(\Theta)$ all we need to complete the proof of case $\textsc{EagerLet}$ is to show that $m - m' \geq 1 + (m_2 - m'_2)$ $(\geq 1 + m''_2 = m'')$.

$$\begin{aligned}
& m - m' \geq 1 + (m_2 - m'_2) \\
\Longleftrightarrow\; & m - t - \phi_{\mathcal{H}''}(w{:}C) - \phi_{\mathcal{H}''}(\Theta) \geq 1 + m_2 - t - 1 - q' - \phi_{\mathcal{H}''}(w{:}C) - \phi_{\mathcal{H}''}(\Theta) \\
\Longleftrightarrow\; & \qquad\qquad \{\, q' = 0 \text{ by premise of } \textsc{EagerLet} \text{ (5.97), since } \widehat{e}[\ell/x] \text{ is in } \textit{whnf} \,\} \\
& m \geq m_2 \\
\Longleftrightarrow\; & t + 1 + q' + q + p + \phi_{\mathcal{H}}(\Gamma, \Delta) + \phi_{\mathcal{H}}(\Theta) \geq t + 1 + q' + q + \phi_{\mathcal{H}_2}(\Delta, \ell{:}A) + \phi_{\mathcal{H}_2}(\Theta)
\end{aligned}$$

Note though that we already showed this last inequality is true, when we established the induction premise (5.101).

This concludes the proof of case $\textsc{EagerLet}$ when $\widehat{e}[\ell/x]$ is in *whnf* and $\widehat{e}[\ell/x] \equiv c\,\vec{y}$.

If $\widehat{e}[\ell/x]$ is in *whnf* and $\widehat{e}[\ell/x] \equiv \lambda y.e'$: From premise $m \geq t + 1 + q' + q + p + \phi_{\mathcal{H}}(\Gamma, \Delta) + \phi_{\mathcal{H}}(\Theta)$ (5.101) we want to derive $m_2 \geq (t + 1 + q' + p) + q + \phi_{\mathcal{H}_2}(\Delta, \ell{:}A) + \phi_{\mathcal{H}_2}(\Theta)$ and for that purpose we have to show $t + 1 + q' + q + p + \phi_{\mathcal{H}}(\Gamma, \Delta) + \phi_{\mathcal{H}}(\Theta) \geq (t + 1 + q' + p) + q + \phi_{\mathcal{H}_2}(\Delta, \ell{:}A) + \phi_{\mathcal{H}_2}(\Theta)$, or equivalently $\phi_{\mathcal{H}}(\Gamma, \Delta) + \phi_{\mathcal{H}}(\Theta) \geq \phi_{\mathcal{H}_2}(\Delta, \ell{:}A) + \phi_{\mathcal{H}_2}(\Theta)$. Note that $\phi_{\mathcal{H}}(\Gamma, \Delta, \Theta) = \phi_{\mathcal{H}_2}(\Gamma, \Delta, \Theta)$ since ℓ is suitably fresh, and thus we just have to show $\phi_{\mathcal{H}_2}(\Gamma) \geq \phi_{\mathcal{H}_2}(\ell{:}A)$. This inequality holds, since by the definition of potential (Figure A.7) we have $\phi_{\mathcal{H}_2}(\ell{:}A) = 0$, given that $\mathcal{H}_2(\ell)$ is a λ-abstraction.

We have all the premises required to apply the induction hypothesis over $e[\ell/x]$, obtaining $m_2', \Gamma_2', \mathcal{C}_2', \mathcal{M}_2'$ and m_2'' such that:

$$\Gamma_2' \overset{0}{\vdash} w : C \tag{A.29}$$

$$\mathcal{H}_2, \mathcal{S}, \mathcal{L} \overset{m_2''}{\vdash} e[\ell/x] \Downarrow w, \mathcal{H}'' \tag{A.30}$$

$$\mathcal{M}_2 <: \mathcal{M}_2' \tag{A.31}$$

$$\mathcal{C}_2' \vdash_{\mathsf{MEM}} (\mathcal{H}'', \mathcal{L}) : \mathcal{M}_2' \tag{A.32}$$

$$\curlyvee(\mathcal{M}_2' \mid (\Gamma_2', \Theta), \mathcal{C}_2') \tag{A.33}$$

$$m_2' \geq (t + 1 + q' + p) + \phi_{\mathcal{H}''}(w{:}C) + \phi_{\mathcal{H}''}(\Theta) \tag{A.34}$$

$$m_2 - m_2' \geq m_2'' \tag{A.35}$$

Let $\Gamma' = \Gamma_2'$, $\mathcal{M}' = \mathcal{M}_2'$ and $\mathcal{C}' = \mathcal{C}_2'$. Equations (A.29) (A.32) (A.33) directly establish the proof obligations (5.102) (5.105) (5.106) respectively.

Conclusion (5.104) follows by (A.31) and the transitivity of subtyping.

By applying rule EagerLet$_{\Downarrow\mathsf{C}}$ with premises (A.21), (A.30) and l being fresh, we establish proof obligation (5.103), yielding

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{1 + m_2''}{\vdash} \text{let } x = \widehat{e} \text{ in } e \Downarrow w, \mathcal{H}''$$

If we choose $m' = t + \phi_{\mathcal{H}''}(w{:}C) + \phi_{\mathcal{H}''}(\Theta)$ all we need to complete the proof of case EagerLet is to show that $m - m' \geq 1 + (m_2 - m_2')$ $(\geq 1 + m_2'' = m'')$.

$$\begin{aligned}
& m - m' \geq 1 + (m_2 - m_2') \\
\Longleftrightarrow\; & m - t - \phi_{\mathcal{H}''}(w{:}C) - \phi_{\mathcal{H}''}(\Theta) \geq 1 + m_2 - t - 1 - q' - \phi_{\mathcal{H}''}(w{:}C) - \phi_{\mathcal{H}''}(\Theta) \\
\Longleftrightarrow\; & \qquad\qquad \{\, q' = 0 \text{ by premise of EagerLet (5.97), since } \widehat{e}[\ell/x] \text{ is in } \textit{whnf} \,\} \\
& m \geq m_2 \\
\Longleftrightarrow\; & t + 1 + q' + q + p + \phi_{\mathcal{H}}(\Gamma, \Delta) + \phi_{\mathcal{H}}(\Theta) \geq t + 1 + q' + p + q + \phi_{\mathcal{H}_2}(\Delta, \ell{:}A) + \phi_{\mathcal{H}_2}(\Theta)
\end{aligned}$$

Note though that we already showed this last inequality is true, when we established the induction premise (5.101).

This last sub-case concludes the proof of case EAGERLET and since the remaining cases are similar to (or simpler than) the ones presented in the soundness proof of the lazy system (in Section 5.5.6.7) this also concludes the proof of the soundness theorem for the eager system.

B. Complete Derivations

B.1 Simple Example: Analysing Call-By-Need

$$\dfrac{\dfrac{}{\mathsf{z}{:}\mathsf{T}^{q''}\big(A'\big)\ \overset{q''}{\vdash}\ \mathsf{z} : A'}\,\textsc{Var} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{y}{:}\mathsf{T}^{q}(B)\ \overset{q}{\vdash}\ \mathrm{y} : B}\,\textsc{Var}}{\emptyset\ \overset{0}{\vdash}\ \lambda\mathrm{y}.\mathrm{y} : \mathsf{T}^{q}(B) \xrightarrow{q} B}\,\textsc{Abs}}{\mathrm{x}{:}\mathsf{T}^{q''}\big(A'\big)\ \overset{0}{\vdash}\ \lambda\mathrm{y}.\mathrm{y} : \mathsf{T}^{q}(B) \xrightarrow{q} B}\,\textsc{Weak}}{\emptyset\ \overset{0}{\vdash}\ \lambda\mathrm{x}.\lambda\mathrm{y}.\mathrm{y} : \mathsf{T}^{q''}\big(A'\big) \xrightarrow{0} \mathsf{T}^{q}(B) \xrightarrow{q} B}\,\textsc{Abs}}{\mathsf{z}{:}\mathsf{T}^{q''}\big(A'\big)\ \overset{0}{\vdash}\ (\lambda\mathrm{x}.\lambda\mathrm{y}.\mathrm{y})\,\mathsf{z} : \mathsf{T}^{q}(B) \xrightarrow{q} B}\,\textsc{App}}{\emptyset\ \overset{1}{\vdash}\ \mathsf{let}\ \mathsf{z} = \mathsf{z}\ \mathsf{in}\ (\lambda\mathrm{x}.\lambda\mathrm{y}.\mathrm{y})\,\mathsf{z} : \mathsf{T}^{q}(B) \xrightarrow{q} B}\,\textsc{Let}$$

$$\text{where } \curlyvee(A' \mid A', A')$$

Figure B.1: Type derivation for a non-strict evaluation example

$$
\dfrac{\left\{\begin{array}{l}
\dfrac{(\text{Figure } B.1,\ \text{where q=0})}{\mathtt{f}{:}\mathsf{T}^1(\mathsf{T}^0(B)\xrightarrow{0}B)\overset{1}{\vdash}\text{let }\mathtt{z}=\mathtt{z}\text{ in }(\lambda\mathtt{x}.\lambda\mathtt{y}.\,\mathtt{y})\,\mathtt{z}:\mathsf{T}^0(B)\xrightarrow{0}B}\,\textsc{Weak}
\\[2ex]
\dfrac{\left\{\begin{array}{l}
\dfrac{\dfrac{\dfrac{}{\mathtt{x}{:}\mathsf{T}^{q'}(C)\overset{q'}{\vdash}\mathtt{x}:C}\,\textsc{Var}}{\emptyset\overset{0}{\vdash}\lambda\mathtt{x}.\,\mathtt{x}:B}\,\textsc{Abs}}{\mathtt{i}{:}\mathsf{T}^0(B)\overset{0}{\vdash}\lambda\mathtt{x}.\,\mathtt{x}:B}\,\textsc{Weak}
\\[2ex]
\dfrac{\dfrac{\dfrac{\left\{\begin{array}{l}
\dfrac{\dfrac{\dfrac{}{\mathtt{f}{:}\mathsf{T}^0(\mathsf{T}^0(B)\xrightarrow{0}B)\overset{0}{\vdash}\mathtt{f}:\mathsf{T}^0(B)\xrightarrow{0}B}\,\textsc{Var}}{\mathtt{f}{:}\mathsf{T}^0(\mathsf{T}^0(B)\xrightarrow{0}B),\,\mathtt{i}{:}\mathsf{T}^0(B)\overset{0}{\vdash}\mathtt{f}\,\mathtt{i}:B}\,\textsc{App}}{\mathtt{f}{:}\mathsf{T}^0(\mathsf{T}^0(B)\xrightarrow{0}B),\,\mathtt{i}{:}\mathsf{T}^0(B),\,\mathtt{v}{:}\mathsf{T}^0(B)\overset{0}{\vdash}\mathtt{f}\,\mathtt{i}:B}\,\textsc{Weak}
\\[2ex]
\dfrac{\dfrac{}{\mathtt{f}{:}\mathsf{T}^0(\mathsf{T}^0(B)\xrightarrow{0}B)\overset{0}{\vdash}\mathtt{f}:\mathsf{T}^0(B)\xrightarrow{0}B}\,\textsc{Var}}{\mathtt{f}{:}\mathsf{T}^0(\mathsf{T}^0(B)\xrightarrow{0}B),\,\mathtt{v}{:}\mathsf{T}^0(B)\overset{0}{\vdash}\mathtt{f}\,\mathtt{v}:B}\,\textsc{App}
\end{array}\right.}{\begin{array}{l}\mathtt{f}{:}\mathsf{T}^0(\mathsf{T}^0(B)\xrightarrow{0}B),\\ \mathtt{f}{:}\mathsf{T}^0(\mathsf{T}^0(B)\xrightarrow{0}B),\,\mathtt{i}{:}\mathsf{T}^0(B)\overset{1}{\vdash}\text{let }\mathtt{v}=\mathtt{f}\,\mathtt{i}\text{ in }\mathtt{f}\,\mathtt{v}:B\end{array}}\,\textsc{Let}}{\mathtt{f}{:}\mathsf{T}^0(\mathsf{T}^0(B)\xrightarrow{0}B),\,\mathtt{i}{:}\mathsf{T}^0(B)\overset{1}{\vdash}\text{let }\mathtt{v}=\mathtt{f}\,\mathtt{i}\text{ in }\mathtt{f}\,\mathtt{v}:B}\,\textsc{Share}}{\mathtt{f}{:}\mathsf{T}^1(\mathsf{T}^0(B)\xrightarrow{0}B),\,\mathtt{i}{:}\mathsf{T}^0(B)\overset{2}{\vdash}\text{let }\mathtt{v}=\mathtt{f}\,\mathtt{i}\text{ in }\mathtt{f}\,\mathtt{v}:B}\,\textsc{Prepay}
\end{array}\right.}{\mathtt{f}{:}\mathsf{T}^1(\mathsf{T}^0(B)\xrightarrow{0}B)\overset{3}{\vdash}\text{let }\mathtt{i}=\lambda\mathtt{x}.\,\mathtt{x}\text{ in }\ldots:B}\,\textsc{Let}
\end{array}\right.}{\emptyset\overset{4}{\vdash}\text{let }\mathtt{f}=(\text{let }\mathtt{z}=\mathtt{z}\text{ in }(\lambda\mathtt{x}.\lambda\mathtt{y}.\,\mathtt{y})\,\mathtt{z})\text{ in let }\mathtt{i}=\lambda\mathtt{x}.\,\mathtt{x}\text{ in let }\mathtt{v}=\mathtt{f}\,\mathtt{i}\text{ in }\mathtt{f}\,\mathtt{v}:B}\,\textsc{Let}
$$

where $\mathsf{B} = \mathsf{T}^{q'}(C)\xrightarrow{q'}C$

Figure B.2: Type derivation for a lazy-evaluation example

B.2 Higher-Order Functions: `map`

$$\dfrac{\left\{\begin{array}{l}
\dfrac{\dfrac{(\text{Figure } B.4)}{\begin{array}{l}
\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L_{in}}) \xrightarrow{q_0} \mathtt{L_{out}}\Big),\ \mathtt{f}{:}\mathsf{T}^0(A \xrightarrow{q_f} B) \\
\overset{0}{\vdash}\ \lambda\mathtt{xs}.\,\text{case } \mathtt{xs} \text{ of } \mathtt{Nil} \text{ -> let } \mathtt{nil} = \mathtt{Nil} \text{ in } \mathtt{nil}, \\
\qquad\qquad\quad \mathtt{Cons\ x\ xs'} \text{ -> let } \mathtt{y} = \mathtt{f\ x} \text{ in} \\
\qquad\qquad\qquad\qquad\quad \text{let } \mathtt{ys'} = \mathtt{map\ f\ xs'} \text{ in} \\
\qquad\qquad\qquad\qquad\quad \text{let } \mathtt{ys} = \mathtt{Cons\ y\ ys'} \text{ in } \mathtt{ys} \\
\ :\ \mathsf{T}^{q_0}(\mathtt{L_{in}}) \xrightarrow{q_0} \mathtt{L_{out}}
\end{array}}\,\textsc{Abs}}{\begin{array}{l}
\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L_{in}}) \xrightarrow{q_0} \mathtt{L_{out}}\Big) \\
\overset{0}{\vdash}\ \lambda\mathtt{f}.\lambda\mathtt{xs}.\,\text{case } \mathtt{xs} \text{ of } \mathtt{Nil} \text{ -> let } \mathtt{nil} = \mathtt{Nil} \text{ in } \mathtt{nil}, \\
\qquad\qquad\quad \mathtt{Cons\ x\ xs'} \text{ -> let } \mathtt{y} = \mathtt{f\ x} \text{ in} \\
\qquad\qquad\qquad\qquad\quad \text{let } \mathtt{ys'} = \mathtt{map\ f\ xs'} \text{ in} \\
\qquad\qquad\qquad\qquad\quad \text{let } \mathtt{ys} = \mathtt{Cons\ y\ ys'} \text{ in } \mathtt{ys} \\
\ :\ \mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L_{in}}) \xrightarrow{q_0} \mathtt{L_{out}}
\end{array}}\,\textsc{Abs} \\[2em]
\dfrac{}{\begin{array}{l}
\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L_{in}}) \xrightarrow{q_0} \mathtt{L_{out}}\Big) \\
\overset{0}{\vdash}\ \mathtt{map}\ :\ \mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L_{in}}) \xrightarrow{q_0} \mathtt{L_{out}}
\end{array}}
\end{array}\right.}{\begin{array}{l}
\emptyset\ \overset{1}{\vdash}\ \text{let } \mathtt{map} = \lambda\mathtt{f}.\lambda\mathtt{xs}.\,\text{case } \mathtt{xs} \text{ of } \mathtt{Nil} \text{ -> let } \mathtt{nil} = \mathtt{Nil} \text{ in } \mathtt{nil}, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathtt{Cons\ x\ xs'} \text{ -> let } \mathtt{y} = \mathtt{f\ x} \text{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{let } \mathtt{ys'} = \mathtt{map\ f\ xs'} \text{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{let } \mathtt{ys} = \mathtt{Cons\ y\ ys'} \text{ in } \mathtt{ys} \\
\qquad \text{in } \mathtt{map}\ :\ \mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L_{in}}) \xrightarrow{q_0} \mathtt{L_{out}}
\end{array}}\,\textsc{Let}$$

where

$$\begin{array}{l}
\mathtt{L_{in}} = \mathtt{L}^{q_t}(3{+}q_f{+}q_l{+}p_c', 1{+}p_n', A) \\
\mathtt{L_{out}} = \mathtt{L}^0(p_c', p_n', \mathsf{T}^0(B)) \\
\mathtt{L'_{in}} = \mathtt{L}^{q_t}(0, 0, A'),\ \text{with } \curlyvee(A \mid A, A') \\
\mathtt{L'_{out}} = \mathtt{L}^0(0, 0, \mathsf{T}^0(B')),\ \text{with } \curlyvee(B \mid B, B') \\
q_l = \mathtt{max}(q_0, q_t)
\end{array}$$

Figure B.3: Type derivation for `map` applied to a list with potential

$$
\dfrac{\left\{\begin{array}{l}
\dfrac{}{\mathtt{xs}{:}\mathsf{T}^{q_0}(\mathtt{L_{in}}) \vdash^{q_0} \mathtt{xs} \ : \ \mathtt{L_{in}}}\textsc{Var}
\\[4ex]
\dfrac{\dfrac{\dfrac{\left\{\begin{array}{l}
\dfrac{\dfrac{}{\emptyset \vdash^{0} \mathtt{Nil} \ : \ \mathtt{L_{out}}}\textsc{Cons}}{\mathtt{nil}{:}\mathsf{T}^0(\mathtt{L'_{out}}) \vdash^{0} \mathtt{Nil} \ : \ \mathtt{L_{out}}}\textsc{Weak}
\\[4ex]
\dfrac{}{\mathtt{nil}{:}\mathsf{T}^0(\mathtt{L_{out}}) \vdash^{0} \mathtt{nil} \ : \ \mathtt{L_{out}}}\textsc{Var}
\end{array}\right.}{\emptyset \vdash^{1+p'_n} \text{let } \mathtt{nil} = \mathtt{Nil} \text{ in } \mathtt{nil} \ : \ \mathtt{L_{out}}}\textsc{Let}}{\begin{array}{l}\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L_{in}}) \xrightarrow{q_0} \mathtt{L_{out}}\Big)\\ \vdash^{1+p'_n} \text{let } \mathtt{nil} = \mathtt{Nil} \text{ in } \mathtt{nil} \ : \ \mathtt{L_{out}}\end{array}}\textsc{Weak}}{\begin{array}{l}\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L_{in}}) \xrightarrow{q_0} \mathtt{L_{out}}\Big), \ \mathtt{f}{:}\mathsf{T}^0(A \xrightarrow{q_f} B)\\ \vdash^{1+p'_n} \text{let } \mathtt{nil} = \mathtt{Nil} \text{ in } \mathtt{nil} \ : \ \mathtt{L_{out}}\end{array}}\textsc{Weak}
\\[4ex]
\dfrac{\dfrac{\left\{\begin{array}{l}
\dfrac{\dfrac{\dfrac{}{\mathtt{f}{:}\mathsf{T}^0(A \xrightarrow{q_f} B) \vdash^{0} \mathtt{f} \ : \ A \xrightarrow{q_f} B}\textsc{Var}}{\mathtt{f}{:}\mathsf{T}^0(A \xrightarrow{q_f} B), \ \mathtt{x}{:}A \vdash^{q_f} \mathtt{f\ x} \ : \ B}\textsc{App}}{\mathtt{f}{:}\mathsf{T}^0(A \xrightarrow{q_f} B), \ \mathtt{x}{:}A, \ \mathtt{y}{:}\mathsf{T}^{q_f}(B') \vdash^{q_f} \mathtt{f\ x} \ : \ B}\textsc{Weak}
\\[4ex]
\text{(Figure } B.5\text{)}
\end{array}\right.}{\begin{array}{l}\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L_{in}}) \xrightarrow{q_0} \mathtt{L_{out}}\Big),\\ \mathtt{f}{:}\mathsf{T}^0(A \xrightarrow{q_f} B), \ \mathtt{f}{:}\mathsf{T}^0(A \xrightarrow{q_f} B), \ \mathtt{x}{:}A, \ \mathtt{xs'}{:}\mathsf{T}^{q_t}(\mathtt{L_{in}})\\ \vdash^{3+q_f+q_l+p'_c} \text{let } \mathtt{y} = \mathtt{f\ x} \text{ in}\\ \qquad\qquad \text{let } \mathtt{ys'} = \mathtt{map\ f\ xs'} \text{ in}\\ \qquad\qquad \text{let } \mathtt{ys} = \mathtt{Cons\ y\ ys'} \text{ in } \mathtt{ys} \ : \ \mathtt{L_{out}}\end{array}}\textsc{Let}}{\begin{array}{l}\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L_{in}}) \xrightarrow{q_0} \mathtt{L_{out}}\Big),\\ \mathtt{f}{:}\mathsf{T}^0(A \xrightarrow{q_f} B), \ \mathtt{x}{:}A, \ \mathtt{xs'}{:}\mathsf{T}^{q_t}(\mathtt{L_{in}})\\ \vdash^{3+q_f+q_l+p'_c} \text{let } \mathtt{y} = \mathtt{f\ x} \text{ in}\\ \qquad\qquad \text{let } \mathtt{ys'} = \mathtt{map\ f\ xs'} \text{ in}\\ \qquad\qquad \text{let } \mathtt{ys} = \mathtt{Cons\ y\ ys'} \text{ in } \mathtt{ys} \ : \ \mathtt{L_{out}}\end{array}}\textsc{Share}
\end{array}\right.}{\begin{array}{l}\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L_{in}}) \xrightarrow{q_0} \mathtt{L_{out}}\Big), \ \mathtt{f}{:}\mathsf{T}^0(A \xrightarrow{q_f} B), \ \mathtt{xs}{:}\mathsf{T}^{q_0}(\mathtt{L_{in}})\\ \vdash^{q_0} \text{case } \mathtt{xs} \text{ of } \mathtt{Nil} \text{ -> let } \mathtt{nil} = \mathtt{Nil} \text{ in } \mathtt{nil},\\ \qquad\quad \mathtt{Cons\ x\ xs'} \text{ -> let } \mathtt{y} = \mathtt{f\ x} \text{ in}\\ \qquad\qquad\qquad\qquad\quad \text{let } \mathtt{ys'} = \mathtt{map\ f\ xs'} \text{ in}\\ \qquad\qquad\qquad\qquad\quad \text{let } \mathtt{ys} = \mathtt{Cons\ y\ ys'} \text{ in } \mathtt{ys} \ : \ \mathtt{L_{out}}\end{array}}\textsc{Case}
$$

Figure B.4: Auxiliary type derivation for `map` applied to a list with potential

$$\left\{\begin{array}{ll}
\hline
\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L}_{\mathtt{in}}) \xrightarrow{q_0} \mathtt{L}_{\mathtt{out}}\Big) & \textsc{Var} \\
\quad \overset{0}{\vdash} \mathtt{map} \;:\; \mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L}_{\mathtt{in}}) \xrightarrow{q_0} \mathtt{L}_{\mathtt{out}} & \\
\hline
\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L}_{\mathtt{in}}) \xrightarrow{q_0} \mathtt{L}_{\mathtt{out}}\Big),\ \mathtt{f}{:}\mathsf{T}^0(A \xrightarrow{q_f} B) & \textsc{App} \\
\overset{0}{\vdash} \mathtt{map\ f} \;:\; \mathsf{T}^{q_0}(\mathtt{L}_{\mathtt{in}}) \xrightarrow{q_0} \mathtt{L}_{\mathtt{out}} & \\
\hline
\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L}_{\mathtt{in}}) \xrightarrow{q_0} \mathtt{L}_{\mathtt{out}}\Big),\ \mathtt{f}{:}\mathsf{T}^0(A \xrightarrow{q_f} B) & \textsc{Subtype} \\
\overset{0}{\vdash} \mathtt{map\ f} \;:\; \mathsf{T}^{\mathtt{min}(q_0,q_t)}(\mathtt{L}_{\mathtt{in}}) \xrightarrow{q_0} \mathtt{L}_{\mathtt{out}} & \\
\hline
\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L}_{\mathtt{in}}) \xrightarrow{q_0} \mathtt{L}_{\mathtt{out}}\Big),\ \mathtt{f}{:}\mathsf{T}^0(A \xrightarrow{q_f} B), & \textsc{App} \\
\mathtt{xs}'{:}\mathsf{T}^{\mathtt{min}(q_0,q_t)}(\mathtt{L}_{\mathtt{in}}) \overset{q_0}{\vdash} \mathtt{map\ f\ xs}' \;:\; \mathtt{L}_{\mathtt{out}} & \\
\hline
\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L}_{\mathtt{in}}) \xrightarrow{q_0} \mathtt{L}_{\mathtt{out}}\Big),\ \mathtt{f}{:}\mathsf{T}^0(A \xrightarrow{q_f} B), & \textsc{Weak} \\
\mathtt{xs}'{:}\mathsf{T}^{\mathtt{min}(q_0,q_t)}(\mathtt{L}_{\mathtt{in}}),\ \mathtt{ys}'{:}\mathsf{T}^{q_0}(\mathtt{L}'_{\mathtt{out}}) \overset{q_0}{\vdash} \mathtt{map\ f\ xs}' \;:\; \mathtt{L}_{\mathtt{out}} & \\
 & \\
\left\{\begin{array}{ll}
\hline
\mathtt{y}{:}\mathsf{T}^0(B),\ \mathtt{ys}'{:}\mathsf{T}^0(\mathtt{L}_{\mathtt{out}}) \overset{0}{\vdash} \mathtt{Cons\ y\ ys}' \;:\; \mathtt{L}_{\mathtt{out}} & \textsc{Cons} \\
\hline
\mathtt{y}{:}\mathsf{T}^0(B),\ \mathtt{ys}'{:}\mathsf{T}^0(\mathtt{L}_{\mathtt{out}}),\ \mathtt{ys}{:}\mathsf{T}^0(\mathtt{L}'_{\mathtt{out}}) \overset{0}{\vdash} \mathtt{Cons\ y\ ys}' \;:\; \mathtt{L}_{\mathtt{out}} & \textsc{Weak} \\
 & \\
\hline
\mathtt{ys}{:}\mathsf{T}^0(\mathtt{L}_{\mathtt{out}}) \overset{0}{\vdash} \mathtt{ys} \;:\; \mathtt{L}_{\mathtt{out}} & \textsc{Var}
\end{array}\right. & \\
\hline
\mathtt{y}{:}\mathsf{T}^0(B),\ \mathtt{ys}'{:}\mathsf{T}^0(\mathtt{L}_{\mathtt{out}}) \overset{1+p'_c}{\vdash} \text{let } \mathtt{ys} = \mathtt{Cons\ y\ ys}' \text{ in } \mathtt{ys} \;:\; \mathtt{L}_{\mathtt{out}} & \textsc{Let} \\
\hline
\mathtt{y}{:}\mathsf{T}^0(B),\ \mathtt{ys}'{:}\mathsf{T}^{q_0}(\mathtt{L}_{\mathtt{out}}) \overset{1+q_0+p'_c}{\vdash} \text{let } \mathtt{ys} = \mathtt{Cons\ y\ ys}' \text{ in } \mathtt{ys} \;:\; \mathtt{L}_{\mathtt{out}} & \textsc{Prepay}
\end{array}\right.$$

$$\begin{array}{ll}
\hline
\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L}_{\mathtt{in}}) \xrightarrow{q_0} \mathtt{L}_{\mathtt{out}}\Big),\ \mathtt{f}{:}\mathsf{T}^0(A \xrightarrow{q_f} B), & \textsc{Let} \\
\mathtt{xs}'{:}\mathsf{T}^{\mathtt{min}(q_0,q_t)}(\mathtt{L}_{\mathtt{in}}),\ \mathtt{y}{:}\mathsf{T}^0(B) & \\
\overset{2+q_0+p'_c}{\vdash} \text{let } \mathtt{ys}' = \mathtt{map\ f\ xs}' \text{ in let } \mathtt{ys} = \mathtt{Cons\ y\ ys}' \text{ in } \mathtt{ys} \;:\; \mathtt{L}_{\mathtt{out}} & \\
\hline
\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L}_{\mathtt{in}}) \xrightarrow{q_0} \mathtt{L}_{\mathtt{out}}\Big),\ \mathtt{f}{:}\mathsf{T}^0(A \xrightarrow{q_f} B), & \textsc{Prepay}^{*} \\
\mathtt{xs}'{:}\mathsf{T}^{q_t}(\mathtt{L}_{\mathtt{in}}),\ \mathtt{y}{:}\mathsf{T}^0(B) & \\
\overset{2+q_l+p'_c}{\vdash} \text{let } \mathtt{ys}' = \mathtt{map\ f\ xs}' \text{ in let } \mathtt{ys} = \mathtt{Cons\ y\ ys}' \text{ in } \mathtt{ys} \;:\; \mathtt{L}_{\mathtt{out}} & \\
\hline
\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_0}(\mathtt{L}_{\mathtt{in}}) \xrightarrow{q_0} \mathtt{L}_{\mathtt{out}}\Big),\ \mathtt{f}{:}\mathsf{T}^0(A \xrightarrow{q_f} B), & \textsc{Prepay} \\
\mathtt{xs}'{:}\mathsf{T}^{q_t}(\mathtt{L}_{\mathtt{in}}),\ \mathtt{y}{:}\mathsf{T}^{q_f}(B) & \\
\overset{2+q_f+q_l+p'_c}{\vdash} \text{let } \mathtt{ys}' = \mathtt{map\ f\ xs}' \text{ in let } \mathtt{ys} = \mathtt{Cons\ y\ ys}' \text{ in } \mathtt{ys} \;:\; \mathtt{L}_{\mathtt{out}} &
\end{array}$$

$*$ Note that rule PREPAY justifies the typing $\mathtt{xs}'{:}\mathsf{T}^{q_t}(\mathtt{L}_{\mathtt{in}})$ from $\mathtt{xs}'{:}\mathsf{T}^{\mathtt{min}(q_0,q_t)}(\mathtt{L}_{\mathtt{in}})$ by prepaying the amount $\mathtt{max}(q_t - q_0, 0)$.

Figure B.5: Auxiliary type derivation for map applied to a list with potential (cont.)

$$
\dfrac{
\left\{
\begin{array}{l}
\dfrac{
\dfrac{\text{(Figure } B.7)}{
\begin{array}{l}
\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\, p'_n-2)} \mathtt{L}_{\mathtt{out}}\Big),\ \mathtt{f}{:}\mathsf{T}^0(A' \xrightarrow{q_f} B) \\
\overset{0}{\vdash}\ \lambda\mathtt{xs}.\,\text{case } \mathtt{xs} \text{ of } \mathtt{Nil} \text{ -> let } \mathtt{nil} = \mathtt{Nil} \text{ in } \mathtt{nil}, \\
\qquad\qquad\qquad \mathtt{Cons\ x\ xs'} \text{ -> let } \mathtt{y} = \mathtt{f\ x} \text{ in} \\
\qquad\qquad\qquad\qquad\qquad\quad \text{let } \mathtt{ys'} = \mathtt{map\ f\ xs'} \text{ in} \\
\qquad\qquad\qquad\qquad\qquad\quad \text{let } \mathtt{ys} = \mathtt{Cons\ y\ ys'} \text{ in } \mathtt{ys} \\
\quad :\ \mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\, p'_n-2)} \mathtt{L}_{\mathtt{out}}
\end{array}
}\ \textsc{Abs}
}{
\begin{array}{l}
\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\, p'_n-2)} \mathtt{L}_{\mathtt{out}}\Big) \\
\overset{0}{\vdash}\ \lambda\mathtt{f}.\lambda\mathtt{xs}.\,\text{case } \mathtt{xs} \text{ of } \mathtt{Nil} \text{ -> let } \mathtt{nil} = \mathtt{Nil} \text{ in } \mathtt{nil}, \\
\qquad\qquad\qquad \mathtt{Cons\ x\ xs'} \text{ -> let } \mathtt{y} = \mathtt{f\ x} \text{ in} \\
\qquad\qquad\qquad\qquad\qquad\quad \text{let } \mathtt{ys'} = \mathtt{map\ f\ xs'} \text{ in} \\
\qquad\qquad\qquad\qquad\qquad\quad \text{let } \mathtt{ys} = \mathtt{Cons\ y\ ys'} \text{ in } \mathtt{ys} \\
\quad :\ \mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\, p'_n-2)} \mathtt{L}_{\mathtt{out}}
\end{array}
}\ \textsc{Abs}
\\[4ex]
\dfrac{}{
\begin{array}{l}
\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\, p'_n-2)} \mathtt{L}_{\mathtt{out}}\Big) \\
\overset{0}{\vdash}\ \mathtt{map}\ :\ \mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\, p'_n-2)} \mathtt{L}_{\mathtt{out}}
\end{array}
}\ \textsc{Var}
\end{array}
\right.
}{
\begin{array}{l}
\emptyset \overset{1}{\vdash}\ \text{let } \mathtt{map} = \lambda\mathtt{f}.\lambda\mathtt{xs}.\,\text{case } \mathtt{xs} \text{ of } \mathtt{Nil} \text{ -> let } \mathtt{nil} = \mathtt{Nil} \text{ in } \mathtt{nil}, \\
\qquad\qquad\qquad\qquad\qquad \mathtt{Cons\ x\ xs'} \text{ -> let } \mathtt{y} = \mathtt{f\ x} \text{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{let } \mathtt{ys'} = \mathtt{map\ f\ xs'} \text{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{let } \mathtt{ys} = \mathtt{Cons\ y\ ys'} \text{ in } \mathtt{ys} \\
\qquad \text{in } \mathtt{map}\ :\ \mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\, p'_n-2)} \mathtt{L}_{\mathtt{out}}
\end{array}
}\ \textsc{Let}
$$

where

$\mathtt{L}'_{\mathtt{in}} = \mathtt{L}^{q_l}(0, 0, A'),$ with $\curlyvee(A' \mid A', A')$
$\mathtt{L}_{\mathtt{out}} = \mathtt{L}^{3+q_f+q_l+p'_c}(p'_c, p'_n, \mathsf{T}^0(B))$
$\mathtt{L}'_{\mathtt{out}} = \mathtt{L}^{3+q_f+q_l+p'_c}(0, 0, \mathsf{T}^0(B')),$ with $\curlyvee(B \mid B, B')$
$q_l = \mathtt{max}(q_0, q_t)$

Figure B.6: Type derivation for `map` applied to a list with no potential

$$
\dfrac{
\begin{cases}
\dfrac{}{\mathtt{xs}{:}\mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \overset{q_l}{\vdash} \mathtt{xs} \ : \ \mathtt{L}'_{\mathtt{in}}}\ \textsc{Var}
\\[2em]
\dfrac{
\dfrac{
\dfrac{
\begin{cases}
\dfrac{\dfrac{}{\emptyset \overset{0}{\vdash} \mathtt{Nil} \ : \ \mathtt{L}_{\mathtt{out}}}\ \textsc{Cons}}{\mathtt{nil}{:}\mathsf{T}^0(\mathtt{L}'_{\mathtt{out}}) \overset{0}{\vdash} \mathtt{Nil} \ : \ \mathtt{L}_{\mathtt{out}}}\ \textsc{Weak}
\\[2em]
\dfrac{}{\mathtt{nil}{:}\mathsf{T}^0(\mathtt{L}_{\mathtt{out}}) \overset{0}{\vdash} \mathtt{nil} \ : \ \mathtt{L}_{\mathtt{out}}}\ \textsc{Var}
\end{cases}
}{\emptyset \overset{1+p'_n}{\vdash} \text{let } \mathtt{nil} = \mathtt{Nil} \text{ in } \mathtt{nil} \ : \ \mathtt{L}_{\mathtt{out}}}\ \textsc{Let}
}{\begin{array}{l}\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\,p'_n-2)} \mathtt{L}_{\mathtt{out}}\Big) \\ \overset{1+p'_n}{\vdash} \text{let } \mathtt{nil} = \mathtt{Nil} \text{ in } \mathtt{nil} \ : \ \mathtt{L}_{\mathtt{out}}\end{array}}\ \textsc{Weak}
}{\begin{array}{l}\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\,p'_n-2)} \mathtt{L}_{\mathtt{out}}\Big),\ \mathtt{f}{:}\mathsf{T}^0(A' \xrightarrow{q_f} B) \\ \overset{1+p'_n}{\vdash} \text{let } \mathtt{nil} = \mathtt{Nil} \text{ in } \mathtt{nil} \ : \ \mathtt{L}_{\mathtt{out}}\end{array}}\ \textsc{Weak}
}{\begin{array}{l}\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\,p'_n-2)} \mathtt{L}_{\mathtt{out}}\Big),\ \mathtt{f}{:}\mathsf{T}^0(A' \xrightarrow{q_f} B) \\ \overset{3+q_f+\mathtt{max}(p'_c,\,p'_n-2)}{\vdash} \text{let } \mathtt{nil} = \mathtt{Nil} \text{ in } \mathtt{nil} \ : \ \mathtt{L}_{\mathtt{out}}\end{array}}\ \textsc{Relax}
\\[2em]
\dfrac{
\dfrac{
\begin{cases}
\dfrac{\dfrac{\dfrac{}{\mathtt{f}{:}\mathsf{T}^0(A' \xrightarrow{q_f} B) \overset{0}{\vdash} \mathtt{f} \ : \ A' \xrightarrow{q_f} B}\ \textsc{Var}}{\mathtt{f}{:}\mathsf{T}^0(A' \xrightarrow{q_f} B),\ \mathtt{x}{:}A' \overset{q_f}{\vdash} \mathtt{f\ x} \ : \ B}\ \textsc{App}}{\mathtt{f}{:}\mathsf{T}^0(A' \xrightarrow{q_f} B),\ \mathtt{x}{:}A',\ \mathtt{y}{:}\mathsf{T}^{q_f}(B') \overset{q_f}{\vdash} \mathtt{f\ x} \ : \ B}\ \textsc{Weak}
\\[2em]
\text{(Figure } B.8\text{)}
\end{cases}
}{\begin{array}{l}\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\,p'_n-2)} \mathtt{L}_{\mathtt{out}}\Big), \\ \mathtt{f}{:}\mathsf{T}^0(A' \xrightarrow{q_f} B),\ \mathtt{f}{:}\mathsf{T}^0(A' \xrightarrow{q_f} B),\ \mathtt{x}{:}A',\ \mathtt{xs}'{:}\mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \\ \overset{3+q_f+\mathtt{max}(p'_c,\,p'_n-2)}{\vdash} \text{let } \mathtt{y} = \mathtt{f\ x} \text{ in} \\ \qquad\qquad \text{let } \mathtt{ys}' = \mathtt{map\ f\ xs}' \text{ in} \\ \qquad\qquad \text{let } \mathtt{ys} = \mathtt{Cons\ y\ ys}' \text{ in } \mathtt{ys} \ : \ \mathtt{L}_{\mathtt{out}}\end{array}}\ \textsc{Let}
}{\begin{array}{l}\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\,p'_n-2)} \mathtt{L}_{\mathtt{out}}\Big), \\ \mathtt{f}{:}\mathsf{T}^0(A' \xrightarrow{q_f} B),\ \mathtt{x}{:}A',\ \mathtt{xs}'{:}\mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \\ \overset{3+q_f+\mathtt{max}(p'_c,\,p'_n-2)}{\vdash} \text{let } \mathtt{y} = \mathtt{f\ x} \text{ in} \\ \qquad\qquad \text{let } \mathtt{ys}' = \mathtt{map\ f\ xs}' \text{ in} \\ \qquad\qquad \text{let } \mathtt{ys} = \mathtt{Cons\ y\ ys}' \text{ in } \mathtt{ys} \ : \ \mathtt{L}_{\mathtt{out}}\end{array}}\ \textsc{Share}
\end{cases}
}{\begin{array}{l}\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\,p'_n-2)} \mathtt{L}_{\mathtt{out}}\Big), \\ \mathtt{f}{:}\mathsf{T}^0(A' \xrightarrow{q_f} B),\ \mathtt{xs}{:}\mathsf{T}^{q_l}(\mathtt{L}'_{\mathtt{in}}) \\ \overset{3+q_f+q_l+\mathtt{max}(p'_c,\,p'_n-2)}{\vdash} \text{case } \mathtt{xs} \text{ of } \mathtt{Nil} \text{ -> let } \mathtt{nil} = \mathtt{Nil} \text{ in } \mathtt{nil}, \\ \qquad\qquad\qquad \mathtt{Cons\ x\ xs}' \text{ -> let } \mathtt{y} = \mathtt{f\ x} \text{ in} \\ \qquad\qquad\qquad\qquad \text{let } \mathtt{ys}' = \mathtt{map\ f\ xs}' \text{ in} \\ \qquad\qquad\qquad\qquad \text{let } \mathtt{ys} = \mathtt{Cons\ y\ ys}' \text{ in } \mathtt{ys} \ : \ \mathtt{L}_{\mathtt{out}}\end{array}}\ \textsc{Case}
$$

Figure B.7: Auxiliary type derivation for map applied to a list with no potential

$$
\left\{\begin{array}{l}
\dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{l}\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathsf{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\,p'_n-2)} \mathsf{L}_{\mathsf{out}}\Big)\\ \quad\vdash^{0} \mathtt{map} \;:\; \mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathsf{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\,p'_n-2)} \mathsf{L}_{\mathsf{out}}\end{array}}\textsc{Var}}{\begin{array}{l}\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathsf{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\,p'_n-2)} \mathsf{L}_{\mathsf{out}}\Big),\\ \quad\mathtt{f}{:}\mathsf{T}^0(A' \xrightarrow{q_f} B) \vdash^{0} \mathtt{map\ f} \;:\; \mathsf{T}^{q_l}(\mathsf{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\,p'_n-2)} \mathsf{L}_{\mathsf{out}}\end{array}}\textsc{App}}{\begin{array}{l}\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathsf{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\,p'_n-2)} \mathsf{L}_{\mathsf{out}}\Big),\\ \mathtt{f}{:}\mathsf{T}^0(A' \xrightarrow{q_f} B),\ \mathtt{xs}'{:}\mathsf{T}^{q_l}(\mathsf{L}'_{\mathtt{in}}) \vdash^{3+q_f+q_l+\mathtt{max}(p'_c,\,p'_n-2)} \mathtt{map\ f\ xs}' \;:\; \mathsf{L}_{\mathsf{out}}\end{array}}\textsc{App}}{\begin{array}{l}\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathsf{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\,p'_n-2)} \mathsf{L}_{\mathsf{out}}\Big),\\ \mathtt{f}{:}\mathsf{T}^0(A' \xrightarrow{q_f} B),\ \mathtt{xs}'{:}\mathsf{T}^{q_l}(\mathsf{L}'_{\mathtt{in}}),\\ \mathtt{ys}'{:}\mathsf{T}^{3+q_f+q_l+\mathtt{max}(p'_c,p'_n-2)}(\mathsf{L}'_{\mathsf{out}}) \vdash^{3+q_f+q_l+\mathtt{max}(p'_c,\,p'_n-2)} \mathtt{map\ f\ xs}' \;:\; \mathsf{L}_{\mathsf{out}}\end{array}}\textsc{Weak}
\\[2em]
\dfrac{\dfrac{\left\{\begin{array}{l}\dfrac{\dfrac{}{\mathtt{y}{:}\mathsf{T}^0(B),\ \mathtt{ys}'{:}\mathsf{T}^{3+q_f+q_l+p'_c}(\mathsf{L}_{\mathsf{out}}) \vdash^{0} \mathtt{Cons\ y\ ys}' \;:\; \mathsf{L}_{\mathsf{out}}}\textsc{Cons}}{\mathtt{y}{:}\mathsf{T}^0(B),\ \mathtt{ys}'{:}\mathsf{T}^{3+q_f+q_l+p'_c}(\mathsf{L}_{\mathsf{out}}),\ \mathtt{ys}{:}\mathsf{T}^0(\mathsf{L}'_{\mathsf{out}}) \vdash^{0} \mathtt{Cons\ y\ ys}' \;:\; \mathsf{L}_{\mathsf{out}}}\textsc{Weak}\\[1.5em] \dfrac{}{\mathtt{ys}{:}\mathsf{T}^0(\mathsf{L}_{\mathsf{out}}) \vdash^{0} \mathtt{ys} \;:\; \mathsf{L}_{\mathsf{out}}}\textsc{Var}\end{array}\right.}{\mathtt{y}{:}\mathsf{T}^0(B),\ \mathtt{ys}'{:}\mathsf{T}^{3+q_f+q_l+p'_c}(\mathsf{L}_{\mathsf{out}}) \vdash^{1+p'_c} \text{let } \mathtt{ys} = \mathtt{Cons\ y\ ys}' \text{ in } \mathtt{ys} \;:\; \mathsf{L}_{\mathsf{out}}}\textsc{Let}}{\begin{array}{l}\mathtt{y}{:}\mathsf{T}^0(B),\ \mathtt{ys}'{:}\mathsf{T}^{3+q_f+q_l+\mathtt{max}(p'_c,p'_n-2)}(\mathsf{L}_{\mathsf{out}})\\ \vdash^{1+\mathtt{max}(p'_c,\,p'_n-2)} \text{let } \mathtt{ys} = \mathtt{Cons\ y\ ys}' \text{ in } \mathtt{ys} \;:\; \mathsf{L}_{\mathsf{out}}\end{array}}\textsc{Prepay*}
\end{array}\right.
$$

$$
\dfrac{\dfrac{\cdots}{\begin{array}{l}\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathsf{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\,p'_n-2)} \mathsf{L}_{\mathsf{out}}\Big),\ \mathtt{f}{:}\mathsf{T}^0(A' \xrightarrow{q_f} B),\\ \mathtt{xs}'{:}\mathsf{T}^{q_l}(\mathsf{L}'_{\mathtt{in}}),\ \mathtt{y}{:}\mathsf{T}^0(B)\\ \vdash^{2+\mathtt{max}(p'_c,\,p'_n-2)} \text{let } \mathtt{ys}' = \mathtt{map\ f\ xs}' \text{ in let } \mathtt{ys} = \mathtt{Cons\ y\ ys}' \text{ in } \mathtt{ys} \;:\; \mathsf{L}_{\mathsf{out}}\end{array}}\textsc{Let}}{\begin{array}{l}\mathtt{map}{:}\mathsf{T}^0\Big(\mathsf{T}^0(A' \xrightarrow{q_f} B) \xrightarrow{0} \mathsf{T}^{q_l}(\mathsf{L}'_{\mathtt{in}}) \xrightarrow{3+q_f+q_l+\mathtt{max}(p'_c,\,p'_n-2)} \mathsf{L}_{\mathsf{out}}\Big),\ \mathtt{f}{:}\mathsf{T}^0(A' \xrightarrow{q_f} B),\\ \mathtt{xs}'{:}\mathsf{T}^{q_l}(\mathsf{L}'_{\mathtt{in}}),\ \mathtt{y}{:}\mathsf{T}^{q_f}(B)\\ \vdash^{2+q_f+\mathtt{max}(p'_c,\,p'_n-2)} \text{let } \mathtt{ys}' = \mathtt{map\ f\ xs}' \text{ in let } \mathtt{ys} = \mathtt{Cons\ y\ ys}' \text{ in } \mathtt{ys} \;:\; \mathsf{L}_{\mathsf{out}}\end{array}}\textsc{Prepay}
$$

$*$ Note that rule Prepay justifies the typing $\mathtt{ys}'{:}\mathsf{T}^{3+q_f+q_l+\mathtt{max}(p'_c,p'_n-2)}(\mathsf{L}_{\mathsf{out}})$ from $\mathtt{ys}'{:}\mathsf{T}^{3+q_f+q_l+p'_c}(\mathsf{L}_{\mathsf{out}})$ by prepaying the amount $\mathtt{max}(p'_n-2-p'_c, 0)$.

Figure B.8: Auxiliary type derivation for `map` applied to a list with no potential (cont.)