Sara Barros Moreira

Resource Analysis for Lazy Evaluation with Polynomial Potential

U.PORTO

FC FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
September 2020

Sara Barros Moreira

Resource Analysis for Lazy Evaluation with Polynomial Potential

Master thesis

Supervisor: Pedro Vasconcelos
Co-supervisor: Mário Florido

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
September 2020

Acknowledgements

First and foremost I would like to thank my supervisors, professors Pedro Vasconcelos and Mário Florido, for their guidance during this year. Their enthusiasm about all the topics discussed, availability, and friendliness, were crucial to move forward this thesis.

I also want to thank my friends and family for all their help and support.

Abstract

The amount of resources a program needs to run is a crucial aspect to its quality, so it is important to have reliable methods that efficiently predict resource usage before execution time. Predicting resource bounds is particularly hard for lazy functional languages, and this can be an obstacle to a broad adoption of non-strict programming languages..

In 2003, Hoffman and Jost [14] presented a system able to automatically obtain upper-bounds on heap cell usage for strict first order programs. Their approach was based on the combination of type-based analysis with Tarjan's [31] description of amortised analysis. This technique can also be called automatic amortised resource analysis (AARA).

In 2012 AARA was extended to lazily evaluated programs, with a system that was able to predict upper-bounds on memory allocation costs [30]. This analysis was successfully applied to several programs, but limited to bounds that are linear in the size of the input.

The main purpose of this dissertation is to extend this previous system for lazy evaluation to polynomial resource bounds, by combining the two previous approaches.

Contents

Abstract 4

List of Figures 7

1 Introduction 8

1.1 Motivation . . . 8

1.2 Objectives and Contribution . . . 10

1.3 Thesis Outline . . . 11

2 Background 12

2.1 The lambda-calculus . . . 12

2.2 Type-based Analysis . . . 17

2.3 Amortisation . . . 20

2.4 Automatic Amortised Resource Analysis . . . 24

2.4.1 First-Order Languages . . . 25

2.4.2 Higher-Order Languages . . . 28

2.4.3 Polynomial Potential . . . 29

2.4.4 Lazy Evaluation . . . 31

2.4.5 Other Related Work . . . 32

3 Polynomial Analysis for Lazy Evaluation 34

3.1 A Motivating Example 34
3.2 A Lazy Functional Language 35
3.2.1 Operational Semantics 36
3.3 Annotated Type System 39
3.3.1 Annotated Types 39
3.3.2 The Sharing Relation 40
3.3.3 Subtyping 40
3.3.4 Type System 42
3.4 Worked Examples 44
3.5 Further Discussion 50

4 Exploratory Implementation **52**
4.1 Overview 52
4.2 Examples 55

5 Conclusion **58**

References **59**

A Analysis for a Simple Lazy Functional Language **63**
A.1 Abstract Syntax for Terms 63
A.2 Abstract Syntax for Types 65
A.3 Damas-Milner Type Inference 67
A.4 Resource Analysis 72

List of Figures

2.1 Type system for SL 19

2.2 Derivation tree for Ex. 2.2.1 19

2.3 Insert and resize operations 21

2.4 Rules T:CONS and T:MATL 30

3.1 Syntax for SLFL expressions and normal forms 35

3.2 Operational semantics for SLFL 37

3.3 Syntax directed type rules 41

3.4 Structural type rules 42

3.5 Sharing rules 43

3.6 Translation of the *pairs* function and auxiliary definitions into SLFL. . 45

4.1 Overview of our system flow 53

4.2 Abstract syntax for terms and types 54

Chapter 1

Introduction

1.1 Motivation

The amount of resources a program needs to run is a crucial aspect to its quality. Critical software can greatly benefit from guaranteed predictions of resource usage, so it is important to study methods that can more accurately estimate resource consumption arising from the execution of programs.

Manually analysing a program, especially to obtain guaranteed approximations, can be very tedious and can very easily lead to error. As a result, automatic analysis of resources has become subject of extensive research [14, 15, 21, 3, 12, 11, 20, 8, 9, 16, 13, 17, 10, 22].

In 2003, Hoffman and Jost [14] proposed a system that was able to automatically and efficiently obtain linear upper bounds on heap cell usage for first order programs. Their approach was later recognised as an instance of amortised analysis (Tarjan, 1985 [31]). This prompted a line of research and development on the field of automatic amortised resource analysis (AARA).

The AARA approach relies on the combination of amortisation with type-based analysis [27], where the type system provides an infrastructure to express the resource analysis.

An important aspect of this technique is that it can be reduced to an linear programming (LP) optimisation problem, which is a well-known area with solutions can be efficiently computed [6].

We should understand that, for Turing complete languages, resource analysis is not decidable, and because of that, there will exist many programs for which these systems will not derive bounds. That being said, this does not take away the fact that resource bound analysis is very beneficial and even crucial for the programs to which it can be applied.

Following Hoffman and Jost's article [14], a lot of development has been done to improve the technique and to surpass some of the obstacles that prevent resource analysis to be applicable to more programming constructs. AARA has since then been extended to polynomial bounds for first-order programs [12, 9], linear bounds for higher-order programs [20], linear bounds for lazily evaluated programs [30, 22] and even had a system that inferred polynomial bounds for higher-order programs [10]. There has been work towards integrating this last system on OCaml's compiler.

As mentioned, part of this extensive research on AARA falls upon its usage to analyse resource usage of lazily-evaluated programs [30, 22]. The importance of this research is clear: although lazy evaluation offers known advantages in terms of modularity and higher abstraction [18], its operational properties (such as time and space behaviour) are more difficult to predict than for strict languages, which can sometimes be an obstacle to a more widespread use of non-strict programming languages, such as Haskell. In addition to being important in this context, this research can also be useful for languages like OCaml, where lazy evaluation can be used if explicitly declared [2].

Previous work on type-based amortised analysis for lazy languages has enabled the automatic prediction of resource bounds for lazy higher-order functional programs with linear costs on the number of (co)data constructors [30, 22]. While this system is an important contribution, it is limited to linear bounds, which means that functions with polynomial costs can not be typed. Because many functions fall under this category, it is important to overcome this limitation.

As a motivating example, consider the two functions *attach* and *pairs* (adapted to Haskell from [11]):

```
pairs :: [a] -> [(a, a)]
pairs [] = []
pairs (x:xs) = attach x xs ++ pairs xs

attach :: a -> [a] -> [(a, a)]
attach _ [] = []
attach y (x:xs) = (x,y):attach x ys
```

The function *pairs* takes a list and computes a list of pairs that are two-element sublists of the given list; this uses an auxiliary definition *attach* that pairs a single element to every element of the argument list.

Considering an execution model where pattern match and constructors have unitary costs, it is straightforward that *attach* requires worst-case time and space that is linear on the length n of the input list. Moreover, a precise bound can be derived by the type system in [22] through a type annotated with a constant *potential* associated with each input list node. Function *pairs*, however, exhibits quadratic time and space on the length its input. Hence, it does not admit a type derivation in the system of [22].

1.2 Objectives and Contribution

The main objective of this thesis is to define a resource analysis system for lazy evaluation that allows polynomial bounds. Our approach is the following:

1. Study amortised analysis. The purpose of this step is to provide familiarisation with the concepts involved in amortisation, emphasising the potential method, which is crucial to the understanding of AARA systems;
2. Study type and effect systems and type-based analysis [27]. This is important to understand how type systems can provide an infrastructure for the analysis;
3. Study prior work on AARA. This step is very relevant to understand how amortisation and type-based analysis were combined in previous works to reach the automatic inference of cost bounds. We focus on the systems for strict evaluation with polynomial bounds [11] and for lazy evaluation with linear bounds [22].The idea is to better understand the key contributions that reflect the properties of lazy evaluation and allow the expression of polynomial bounds;
4. Propose a combined type system for lazy evaluation with polynomial bounds based of those two previous systems;
5. Experiment with some examples that reflect the main properties of this new system;
6. Study the implementation of some of these concepts.

The main contribution is the successful extension of the previous analysis system for lazy evaluation to polynomial bounds. Our type system is formulated for a simple lazy functional language with higher-order functions, pairs, lists and recursion. For simplicity, we focus our analysis on the number of allocations needed for a program to execute, however, this could be easily extended to a parametric cost analysis, as seen in previous works. We apply our analysis to a few simple examples, which we will see detailed later in this dissertation. A secondary contribution is a prototype implementation of this system and a demonstration of the implemented system on two relevant examples.

This work has been present at the 2020 Workshop on Implementation of Functional Languages (IFL)[26].

1.3 Thesis Outline

The rest of this thesis is organised as follows: Chapter 2 surveys important background, more precisely, it introduces relevant basic concepts of the λ-calculus, it explains type-based analysis and its advantages, and it presents a brief description of amortisation. There's also a brief overview of related work about AARA, where we emphasise the previous approaches to lazy evaluation [22] and polynomial bounds [12]. Chapter 3 presents a small lazy functional language and its annotated operational semantics. It also presents the main contribution of this thesis: a type system for resource analysis with polynomial bounds. In the same chapter we demonstrate the analysis on two simple examples, and finish with some final remarks. In Chapter 4 an overview of the prototype implementation of our system is shown and demonstrated on two examples. Finally, Chapter 5 concludes the thesis and offers some directions for future work.

Chapter 2

Background

In this chapter we provide an overview of the theoretical basis that supports our work. We start by introducing the lambda-calculus and some concepts that are present throughout this thesis. We then explain the theory behind type-based analysis and amortised analysis. Finally, we present a summary of other works related to AARA and how they were relevant for this thesis.

2.1 The lambda-calculus

In 1936, Alonzo Church defined a formal system called *lambda calculus* (or λ-calculus). Through this system, Church defined the notion of a computable function that later would be used as a basis for all functional languages (such as Haskell, ML, and others) [4].

When we talk about *the* lambda-calculus, we are generally referring to pure lambda calculus, or lambda-calculus in its most simple shape. The terms for pure λ-calculus may take one of these three forms:

$$\begin{array}{rl} x & \text{variable} \\ (\lambda x.M) & \text{abstraction} \\ (MN) & \text{application} \end{array}$$

Where M and N are λ-terms. x is a simple variable, a character or string. An abstraction $(\lambda x.M)$ essentially represents a nameless function that receives one argument, x, and has body M. (MN) is the application of M to N.

As we can see, there are no numbers, arithmetic operations, conditional statements, and other elements common to programming languages. However, it is easy to extend this system to include these elements, as we will see further in this dissertation, or even encode them as λ-expressions.

Free and bound variables Considering a λ-term $(\lambda x.M)$, we say that the variable x is a *bound variable* in that abstraction, and every occurrence of x in the body M is bound by this abstraction. In contrast, an occurrence is considered to be *free* if it is not bound by any abstraction, for example, in $(\lambda x.xy)$, x occurs bound and $y$occurs free.

Definition 1. We call $\mathrm{BV}(M)$ the set of all bound variable occurrences in M and define it as follows:

$$\begin{aligned} \mathrm{BV}(x) &= \emptyset \\ \mathrm{BV}(\lambda x.M) &= \{x\} \cup \mathrm{BV}(M) \\ \mathrm{BV}(MN) &= \mathrm{BV}(M) \cup \mathrm{BV}(N) \end{aligned}$$

Definition 2. We call $\mathrm{FV}(M)$ the set of all free variable occurrences in M and define it as follows:

$$\begin{aligned} \mathrm{FV}(x) &= \{x\} \\ \mathrm{FV}(\lambda x.M) &= \mathrm{FV}(M) \setminus \{x\} \\ \mathrm{FV}(MN) &= \mathrm{FV}(M) \cup \mathrm{FV}(N) \end{aligned}$$

Substitution Consider an abstraction $(\lambda x.M)$. If we name the function it represents as f, we can say $f(x) = M$ This means that, applying f to some argument N requires the substitution by N of all free occurrences of x in M.

Definition 3. The substitution of x by N in term M can be represented like $M[N/x]$ and defined as follows:

$$\begin{aligned} x[N/y] &= \begin{cases} N & \text{if x = y} \\ x & \text{otherwise} \end{cases} \\ (\lambda x.M)[N/y] &= \begin{cases} M & \text{if x = y} \\ (\lambda x.M[N/y]) & \text{otherwise} \end{cases} \\ (ML)[N/y] &= (M[N/y]L[N/y]) \end{aligned}$$

Variable capture Note that, if we apply some substitution $(\lambda y.xy)[y/x]$, this would originate a term $(\lambda y.yy)$. However, we know that the y introduced by the substitution is not the same y introduced by the abstraction, so the term $(\lambda y.yy)$ is confusing two different variables that can represent different things, because they have the same name. This must be avoided by, before applying a substitution $\lambda x.M[N/y]$, requiring that $x \notin FV(N)$ (we can see that in the example mentioned this does not happen, because $y \in FV(y)$). In situations where this condition is not initially met, the bound variable must be renamed in order to avoid variable capture, for example, considering again the term $(\lambda y.xy)[y/x]$, we could rename y to z in the term $(\lambda y.xy)$ and this would originate $(\lambda y.xy)[y/x] = (\lambda z.xz)[y/x] = (\lambda z.yz)$. We can see that the original confusion does not happen anymore.

In this dissertation we will consider a *variable convention* [24] that states that, if M is a λ-term in any context, then all free variables in M are chosen to be different from all bound variables of M. This way, the problem of variable capture will not happen, because it would be violating this convention.

Currying As we can observe in the syntax of λ-terms, one abstraction bounds only one variable, meaning that it can only represent a function with one argument. *Currying* is used to express functions with multiple arguments by taking advantage of functions whose result is another function. For example, if we want to formalise the function $f(x, y) = M$, we can represent like $(\lambda x.(\lambda y.M))$, for any term M and when we apply it to arguments L and N the result is obtained by replacing x with L and y by N. We can see how this works for any number of arguments.

Reduction A reduction relation can actually be defined as two relations, one-step reduction ($\rightarrow$) and multi-step reduction ($\twoheadrightarrow$).

Definition 4. A one-step reduction can be represented as $M \rightarrow L$, which essentially means "M reduces to L by one step". We define $\rightarrow$ with the following rules:

$$\frac{M \rightarrow M'}{MN \rightarrow M'N} \quad \frac{N \rightarrow N'}{MN \rightarrow MN'} \quad \frac{N \rightarrow N'}{(\lambda x.N) \rightarrow (\lambda x.N')}$$

$$(\lambda x.M)N \rightarrow_\beta M[N/x] \quad (\beta)$$

Multi-step reduction is a reflexive transitive closure of one-step reduction and, as the name indicates, allows many steps, including 0 (because of reflexivity).

Definition 5. We represent multi-step reduction like $M \twoheadrightarrow M'$, which can be read as "M reduces to M'", and define it with the following rules:

$$\frac{M \rightarrow M'}{M \twoheadrightarrow M'} \quad \frac{M \twoheadrightarrow M'' \quad M'' \twoheadrightarrow M'}{M \twoheadrightarrow M'} \quad \frac{}{M \twoheadrightarrow M'}$$

When a term can admit no reduction, we say that it is in *normal form*.

Definition 6. If M is a λ-term, then M is in normal form if it does not have any sub-terms of the form $(\lambda x.R)S$.

Considering again the term mentioned above, we can write:

$$(\lambda x.x)((\lambda y.y)a) \rightarrow_\beta (\lambda y.y)a \rightarrow_\beta a$$

and

$$(\lambda x.x)((\lambda y.y)a) \twoheadrightarrow a$$

We say that a is the normal form of the term $((\lambda x.x)((\lambda y.y)a)$.

Note how, in the first reduction, we are faced with the choice of what term to reduce first. We decided to start by reducing the outermost application, but we could have chosen to start by reducing $(\lambda y.y)a$ and we would have reached the same value. This raises the question of whether we can actually reach different normal forms depending on the choices we make during reduction. The answer to this question is no, and this is proven by a corollary of the Church-Rosser Theorem (See section 3.2 of Hankin's guide for a detailed explanation [24]).

Reduction strategies Another question that can be raised is whether there is a better choice to be made depending on the term to be reduced. This is a more complicated discussion.

We will briefly introduce the concept behind two reduction orders: *normal order reduction* and *applicative order reduction*. Normal order reduction is an reduction strategy that chooses the leftmost reducible expression (redex) to be reduced first, a beta-redex is substituted as is in the body of the expression and therefore evaluated as often as it is used. It compares to a call-by-name evaluation strategy in programming languages. Contrasting with a normal order reduction, there is a reduction strategy called applicative order reduction. This strategy chooses to evaluate the leftmost innermost argument, meaning that all arguments are evaluated once; this strategy corresponds to a call-by-value evaluation.

Example 2.1.1.

$$(\lambda x.xx)((\lambda y.y)a) \to_\beta (\lambda x.xx)a \to_\beta aa \qquad \text{(Applicative order)}$$
$$(\lambda x.xx)((\lambda y.y)a) \to_\beta ((\lambda y.y)a)((\lambda y.y)a) \to_\beta a((\lambda y.y)a) \to_\beta aa \qquad \text{(Normal order)}$$

Since applicative order reduces all arguments only once, it has the advantage of preventing excessive computations. However, consider the following expression:

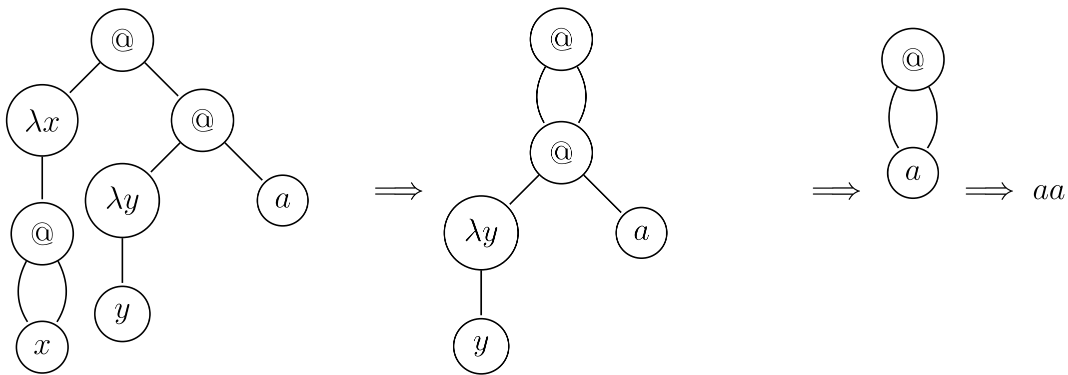
$$(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))z$$

Applicative order would try to evaluate the argument $(\lambda x.xx)(\lambda x.xx)$ over and over and to no avail. Normal order reduction, on the other hand, would immediately evaluate the expression to z. This means that, under a normal order evaluation, if an argument is not needed, it will not be evaluated. Moreover, applicative order may fail to reach a normal form whereas normal order will always reach a normal form if it exists.

Lazy evaluation, or call-by-need, is an technique used to implement normal order evaluation efficiently, in a way that avoids duplication of computations. It does this by delaying the evaluation of expressions until their value is needed.

To avoid duplication, this technique introduces mechanisms to "share" sub-expressions by representing a term as graph. We will not dive deep into graph theory, but essentially, graph reduction is the main implementation technique used for lazy evaluation. As mentioned, a term is represented as a graph that can have multiple references to the same node, which avoids repeated evaluations. A graph can have @-nodes to represent applications, λ-nodes to represent an abstraction, and leafs to represent variables or constants. It is important to understand this concept because it heavily influences the mechanics of our analysis, to better understand it let us take a look at the following example.

Example 2.1.2. In this example we can see how, when evaluating the same expression, the evaluation of the argument is delayed until needed. We start by reducing the outermost @-node (corresponds to the reduction $(\lambda x.xx)((\lambda y.y)a) \to_\beta ((\lambda y.y)a)((\lambda y.y)a)$). The two pointers to the node x, now point to the application $((\lambda y.y)a)$, and because of that, we avoid reducing the term $(\lambda y.y)a$ twice, but still reach the correct value.



This is an informal representation of this strategy. Later in this thesis, when we introduce our operational semantics, it will be more clear how it can be formally defined.

2.2 Type-based Analysis

Static analyses [25] are program analysis techniques which aim to predict, before execution, program behaviour arising during execution. This is usually useful for compiler optimizations (e.g. to check if the program contains dead code, to avoid redundant computations, etc), to ensure program correctness (e.g. statically checked exception handlers in Java), and program development (aid for modern IDEs, to support debugging, refactoring, and program understanding).

Type-based analysis [27] is an approach to static analysis that attaches static analysis information to types.

One main advantage of this approach is the fact that it facilitates modular analysis, as types allow for the expression of interfaces between components. It also helps the comunication with the programmer by extending an already-known notation, namely, types.

Other advantages revolve around efficiency and completeness. Types provide an infrastructure from which the analysis can be done. For example, in a type and effect system, each typing rule provides a localised setting for the analysis, as we will show in example 1. Furthermore, the correctness of the analysis is subsumed by he correctness of the type system, which means that the correctness of the analysis can be formulated and proven using the well studied methods in type systems.

Type and Effect Systems Type and effect systems are a particular case of type-based analysis. Overall, these systems improve the information given by types by decorating them with annotations so that they express more about the program being analyzed (effects).

There are many classes of analysis in which one can take advantage of these systems. One specific class that can benefit from this technique is exception analysis. We will exemplify how it works with the following example:

Consider a small functional language SL [27] with types $\tau := \mathsf{int} \mid \tau \to \tau$ and expressions

$$
\begin{aligned}
e ::= {} & c \\
\mid {} & x \\
\mid {} & \lambda x.e \\
\mid {} & e\ e \\
\mid {} & \mathsf{ifzero}\ e\ e\ e \\
\mid {} & \mathsf{raise}\ T \\
\mid {} & \mathsf{try}\ e\ \mathsf{handle}\ T\ e
\end{aligned}
$$

Let us annotate the types with effects ϵ, which represent possible exceptions raised, $\tau := \mathsf{int} \mid \tau \xrightarrow{\epsilon} \tau$.

The analysis is formulated using annotated type inference rules as presented in figure 3.2. There are three rules deserving of explanation. RAISE raises the exception that is associated to it, and TRY handles the exception associated to it, removing it from the set of possible exceptions raised. The abstraction rule ABS is the one that captures the exception resulting from the body of the function and annotates it to the type of the function. We omit the operational semantics for this language (see [27]).

The purpose of this analysis is to determine an approximation to the expeptions that could be raised by evaluating a program.

Example 2.2.1. For the program:

$$(\lambda x.\mathsf{ifzero}\ x\ (\mathsf{raise}\ \mathsf{Z})\ x)\ 3$$

we can analyse $(\lambda x.\mathsf{ifzero}\ x\ (\mathsf{raise}\ Z)\ x)$ so as to obtain $\mathsf{int} \xrightarrow{Z} \mathsf{int}; \emptyset$, as shown in Fig. 2.2. Hence, the overall type and effect of the entire program is $\mathsf{int}; Z$, meaning that the evaluation of the program might raise exception Z.

$$\frac{}{\Gamma \vdash c{:}\mathsf{int};\emptyset}\ (\textsc{Const})$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x{:}\tau;\emptyset}\ (\textsc{Var})$$

$$\frac{\Gamma, x:\tau_1 \vdash e:\tau_2;\epsilon}{\Gamma \vdash \lambda x.e:\tau_1 \xrightarrow{\epsilon} \tau_2;\emptyset}\ (\textsc{Abs})$$

$$\frac{\Gamma \vdash e:\tau_2 \xrightarrow{\epsilon_3} \tau_1;\epsilon_1 \quad \Gamma \vdash e_2:\tau_2;\epsilon_2}{\Gamma \vdash e_1\ e_2:\tau_1;\epsilon_1 \cup \epsilon_2 \cup \epsilon_3}\ (\textsc{App})$$

$$\frac{\Gamma \vdash e_0:\mathsf{int};\emptyset \quad \Gamma \vdash e1:\tau;\epsilon_1 \quad \Gamma \vdash e_2:\tau;\epsilon_2}{\Gamma \vdash \mathsf{ifzero}\ e_0\ e_1\ e_2:\tau;\epsilon_1 \cup \epsilon_2}\ (\textsc{Ifzero})$$

$$\frac{}{\Gamma \vdash \mathsf{raise}\ T:\tau;T}\ (\textsc{Raise})$$

$$\frac{\Gamma \vdash e_1:\tau;\epsilon_1 \quad \Gamma \vdash e_2:\tau;\epsilon_2}{\Gamma \vdash \mathsf{try}\ e_1\ \mathsf{handle}\ T\ e_2:\tau;(\epsilon_1/T) \cup \epsilon_2}\ (\textsc{Try})$$

Figure 2.1: Type system for SL

$$\frac{\dfrac{}{\vdash 3{:}\mathsf{int};\emptyset}\ \text{Const} \qquad \dfrac{\dfrac{\dfrac{\Gamma(x) = \mathsf{int}}{x{:}\mathsf{int} \vdash x{:}\mathsf{int};\emptyset}\ \textsc{Var} \quad \dfrac{}{x{:}\mathsf{int} \vdash (\mathsf{raise}\ Z):\tau;Z}\ \text{Raise} \quad \dfrac{\Gamma(x) = int}{x{:}\mathsf{int} \vdash x:int;\emptyset}\ \text{Var}}{x:int \vdash \mathsf{ifzero}\ (\mathsf{raise}\ Z)\ x{:}\mathsf{int};Z}\ \text{Ifzero}}{\vdash \lambda x.\mathsf{ifzero}\ x\ (\mathsf{raise}\ Z)\ x{:}\mathsf{int} \xrightarrow{Z} \mathsf{int}}\ \text{Abs}}{\vdash (\lambda x.\mathsf{ifzero}\ x\ (\mathsf{raise}\ \mathsf{Z})\ x)\ 3{:}\mathsf{int};Z}\ \text{App}$$

Figure 2.2: Derivation tree for Ex. 2.2.1

2.3 Amortisation

Amortized analysis [28] is a method for analysing the complexity of a sequence of operations. While a worst case analysis considers the worst case for each operation, and an average case analysis considers the average cost over all possible inputs, an amortised analysis is concerned with the overall worst-case cost over a sequence of operations. The motivation for this type of analysis arises from the fact that some operations can be costly, while others can be faster or "cheaper", and in the end they can even each other out. In some cases, analysing the worst-case per operation may be too pessimistic.

In an amortised analysis we define a notation of "amortised cost" for each operation that satisfies the following equation:

$$\sum_{n=1}^{m} a_n \geq \sum_{n=1}^{m} t_n$$

With a as the amortised cost and t as the actual cost,this means that, for each sequence of operations, the total amortised cost is an upper bound of the total actual cost. As a consequence, in each intermediate step of the sequence, the accumulated amortised cost is an upper bound of the accumulated actual cost. This allows for the existence of operations with an actual cost that exceeds their amortized cost, these are called *expensive operations*. *Cheap operations* are operations with a cost lower than their amortised cost. Expensive operations can only occur when the difference between the accumulated amortized cost and the acumulated actual cost (*accumulated savings*) is enough to cover the "extra" cost.

There are three different methods for amortised analysis: the *aggregate method (total cost)*, the *accounting method (banker's view)* and the *potential method* (physicist's view). The choice of which to use depends on how convenient each is to the situation.

We will look at the classic example [7] of a dynamic array A that needs to be resized (doubled, in this case) every time we want to insert an element el but the size of the array has reached its full capacity. The functions are defined in pseudo code in figure 3.1, with $\mathsf{size}(A)$ being a function that returns the size of array A and $\mathsf{capacity}(A)$ being a function that returns the full capacity of array A (size + free space).

Let us use amortized analysis to prove that the cost of a sequence of n insert operations is $O(n)$, that is, each insert has an amortized cost of $O(1)$.

```
1: procedure INSERT (EL, A)
2:     if size(A) = capacity(A) then
3:         A ← Resize(A)
4:     A.append(el)
5: procedure RESIZE (A)
6:     B ← new Array(size(A)*2)
7:     for let x of A do B.append(x)
8:     return B
```

Figure 2.3: Insert and resize operations

It's easy to see how an amortized analysis can be a more appropriate approach to this case, as the resize operation, which is more costly, only happens every so often.

Aggregate Method In the aggregate method the total running time of the sequence of operations is analyzed. In a sequence that takes $T(n)$ time in the worst case, the amortized cost of each operation is $T(n)/n$.

Lets assume we want to insert n elements to our dynamic array A and each operation i has a cost of c_i. The we can see that c_i is as follows:

$$c_i = 1 + \begin{cases} i-1 & \mathsf{size}(A_i) = \mathsf{capacity}(A_i) \\ 0 & \mathsf{otherwise} \end{cases}$$

The total amortized cost will be:

$$\frac{\sum_{i=1}^{n} c_i}{n}$$

and $\frac{\sum_{i=1}^{n} c_i}{n} = \frac{n+\sum_{j=1}^{|log_2(n-1)|} 2^j}{n} = \frac{\mathcal{O}(n)}{n} = \mathcal{O}(1)$.

Note that when $\mathsf{size}(A_i) = \mathsf{capacity}(A_i)$, $i-1$ is a power of 2 (since we're always doubling the size of the arry), $\sum_{j=1}^{|log_2(n-1)|} 2^j$ is simply the sumations of the powers of 2 up to $n-1$.

Accounting Method This method handles the accumulated savings like credits associated to a location in a data structure. These credits are used to "pay" for future accesses to those locations.

Much like a savings bank account, low-cost operations are charged a little bit more than their true cost, and the surplus is deposited into the bank account for later use. The amount of extra charge should be chosen such that the balance in the bank account always remains positive.

It's important to understand that the extra cost does not mean that the operations actually need those extra resources, this serves simply for analysis purposes.

The amortized cost of an operation is the total cost of that operation, plus the credits alocated by the operation minus the credits spent by the operation.

Let us assume again we want to insert n elements to our dynamic array A and each operation i has a cost of c_i.

We define that each operation has an actual cost c_i of 1, but we charge 3 credits for each insertion. These are the credits that will pay for the extra cost of the resizing operations.

When we want to insert an element and there's enough space, we insert it into one of the free spaces of the array with a cost of 1, and associate 1 credit to that element, and another to the element $\frac{\text{capacity}(A)}{2}$ positions prior to the current position (in the first insertion we "waste" one of the credits).

When we want to resize the array, each element already in the array will have 1 credit associated to it, that pays for its insertion in the new array.

This way, the extra cost of the resize operations is spread over the entire sequence, making the amortized cost of each operation O(1).

Potential Method This method defines a function Φ that maps each state of the data structure d_i to a real number (*potential of* d_i). This function should be chosen such that the potential of the initial state is 0 and never becomes negative, that is, $\Phi(d_0) = 0$ and $\Phi(d_i) \geq 0$, for all i. This potential represents a lower bound to the accumulated savings.

The amortized cost of an operation is defined as its actual cost (t_i), plus the change in potential between d_{i-1} and d_i, where d_i is the state of data structure before operation i:

$$a_i = t_i + \Phi(d_i) - \Phi(d_{i-1})$$

This means that:

$$\begin{aligned}\sum_{i=1}^{j} t_i &= \sum_{i=1}^{j}(a_i + \Phi(d_{i-1}) - \Phi(d_i)) \\ &= \sum_{i=1}^{j} a_i + \sum_{i=1}^{j}(\Phi(d_{i-1}) - \Phi(d_i)) \\ &= \sum_{i=1}^{j} a_i + \Phi(d_0) - \Phi(d_j)\end{aligned}$$

Note that the sequence of potential function values forms a *telescoping series* and thus all terms except the initial and final values cancel in pairs. And because $\Phi(d_j)$ is always equal or greater than $\Phi(d_0)$, then $\Sigma(a_i) \geq \Sigma(t_i)$.

Let us assume again we want to insert n elements to our dynamic array A.

We define the potential function as $\Phi(A_i) = 2\mathsf{size}(A_i) - \mathsf{capacity}(A_i)$. A_i is the state of the array before operation i.

Given that when we resize the array, it always becomes at least half full, the potential function will never be negative.

When we do a resize operation, the potential value becomes zero (because $\mathsf{capacity}(A_i) = 2 * \mathsf{capacity}(A_{i-1})$ and $\mathsf{size}(A_i) = \mathsf{capacity}(A_{i-1})$). The allocation of a new array and copy of the values from the old array to the new one has an atual cost of O(n), but the decrease in potential eavens out that value leaving a total amortized cost of O(1).[1]

As we can see, with the right choice of potential function, amortized analysis gives a tigher bound for the sequence of operations than simply analysing each operation individually - that would give a pessimistic bound of $O(n^2)$ for n operations.

This last method (potential method) is very relevant in the context of this thesis because the AARA approach is based on it. More precisely, this approach takes on the concept of potential and associates it to the data structures involved in the analysis through a type system. This will be further discussed in following sections of this dissertation.

[1] Note that, immediatly after a resizing operation the array becomes half full, so the potential is never negative.

2.4 Automatic Amortised Resource Analysis

In 2003, Hofmann and Jost [14] proposed a system for static automatic analysis of heap space usage for a strict first-order language. This system was able to obtain linear bounds on the heap space consumption of a program by using a type system refined with resource annotations. This annotated type system allowed the analyser to predict the amount of heap space needed to evaluate the program by keeping track of the memory resources available. This form of analysis would later be recognised as an instance of Tarjan's amortisation [31].

Further work has been done using this approach, which is more specifically based on the potential method of amortised analysis. The idea behind this approach is the association of potential to data structures through type annotations, where the annotations serve as coefficients for the potential function. The key to a successful analysis is the choice of a "good" potential function, "good" being a potential function that simplifies the amortised costs. To reach good values for the potential function, the type rules stipulate restrictions over the type annotations. During type derivation, those restrictions generate constraints that are collected and then sent to an off-the-shelf LP solver, to be solved automatically.

In 2010 [12], the same authors address the biggest limitation on previous article [14]: restriction to linear bounds. Their new system infers polynomial upper bounds on resource usage for first-order programs as a function of their input, and is generic in terms of resources. This extension is done without losing expressiveness. The infered polynomial bounds result in linear constraints, meaning that the inference of polynomial bounds can still be reduced to an linear optimisation problem.

Later, Jost et al. [20] present the first automatic amortised analysis able to determine linear upper-bounds on the use of quantitative resources for strict, higher-order recursive programs.

In [22] AARA is extended to compute linear bounds for lazily evaluated functional languages. This is an important extension because it tries to remove an obstacle to the broader use of lazy languages: the fact that resource usage for their execution very hard to predict. This system improves the precision of the analysis for co-recursive data by combining two previous analyses that considered the allocation costs of recursive and co-recursive programs. The system is generalised to a parametric cost model and has the key aspect of tracking self-references, which is essential to model the graph reduction techniques that are typically used in lazy functional language

implementations.

These four contributions to AARA are the main support for our analysis, and we will explain how in the next sections.

2.4.1 First-Order Languages

A starting point for the study of AARA for this dissertation was Hoffmann and Jost's article [14], which we have summarised above and will now explain in more detail. It served as a support to better understand how certain concepts, such as resource usage and collection of resource constraints, are introduced in a language, and how they appear in practice.

As mentioned, the article presents a system that can efficiently obtain *linear bounds* on the heap space consumption of *first-order functional programs.* They present the problem as follows:

"Given a functional program containing function f of a certain type (...) *find a function v such that the computation $f(w)$ requires no more than $v(w)$ additional cells".*

In short, the article addresses this problem by instrumenting the code of the program being analysed by a counter that is augmented each time there is the need to allocate a heap cell. The function v is the function computed by the instrumented code, followed by a projection that only keeps the value of the counter.

A first-order typed language LF is defined and decorated with a freelist that represents the number of available heap cells. Resource annotations are introduced to the type system, which allow the analyser to predict the amount of heap space needed to evaluate the program. A simplified version of their language is given by the following grammar (where * represent the empty tuple, also known as unit):

$$
\begin{aligned}
e ::= & * \quad | \quad x \\
| \ & \mathsf{nil} \\
| \ & f(x_1 ... x_n) \\
| \ & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
| \ & \mathsf{cons}(x_1, x_2) \\
| \ & \mathsf{match}\ x\ \mathsf{with}\ \mathsf{cons}(x_1, x_2)\ \texttt{->}\ e_1\ |\ \mathsf{nil}\ \texttt{->}\ e_2
\end{aligned}
$$

To define the operational semantics of the language they consider a set of locations **Loc**, which represents a set of heap addresses; a stack S, which is a mapping from variables to values; a heap σ, which is a mapping from locations to values, and a freelist m.

Lastly, they define their values as:

$$v ::= c \mid l \mid \mathsf{NULL} \mid (v, v)$$

A function **SIZE** is defined, which returns the size for a given value. The idea is that value v occupies $\mathbf{SIZE}(v)$ words in the heap.

The evaluation of an expression is done according to a relation of the form: $m, \sigma, S \vdash e \rightsquigarrow v, \sigma', m'$ (where $\rightsquigarrow$ represents the evaluation operation). As a side effect, the heap can be modified and the freelist can increase or decrease.

We will omit in this report most rules for the operational semantics, except two particular ones. There are two versions of pattern matching, `Match` immediately deallocates the node matched against, meaning that the freelist grows, while the `Match'` construct preserves it, and the freelist stays the same.

$$\frac{\begin{array}{c} S(x) = l \quad \sigma(l) = (v_h, v_t) \quad m_0 = m + \mathbf{SIZE}(\sigma(l)) \\ m_0, S[x_h \to v_h][x_t \to v_t], \sigma \setminus l \vdash e_1 \rightsquigarrow v, \sigma', m' \end{array}}{m, S, \sigma \vdash \mathsf{match}\ x\ \mathsf{with}\ \mathsf{cons}(x_1, x_2)\ \texttt{->}\ e_1\ |\ \mathsf{nil}\ \texttt{->}\ e_2 \rightsquigarrow v, \sigma', m'} \ (\textsc{Match-Cons})$$

$$\frac{\begin{array}{c} S(x) = l \quad \sigma(l) = (v_h, v_t) \\ m, S[x_h \to v_h][x_t \to v_t], \sigma \vdash e_1 \rightsquigarrow v, \sigma', m' \end{array}}{m, S, \sigma \vdash \mathsf{match}\ x\ \mathsf{with}\ \mathsf{cons}(x_1, x_2)\ \texttt{->}\ e_1\ |\ \mathsf{nil}\ \texttt{->}\ e_2 \rightsquigarrow v, \sigma', m'} \ (\textsc{Match-Cons'})$$

The resource annotations are introduced in the type system, the following grammars

defines the simple types and the annotated types.

Simple types

$A := 1 \mid B \mid A \otimes A \mid A \oplus A \mid L(A)$

$F := (A, ..., A) \rightarrow A$

Annotated types

$P := 1 \mid B \mid P \otimes P \mid R \oplus R \mid L(R)$

$R := (P, k)$

$F := (P, ...P, k) \rightarrow R$

The typing judgment for the annotated type system is $\Gamma, n \vdash_\Sigma e : A, n'$, which means that, given a signature Σ that maps a set of function names to their first-order types, a typing environment Γ, and with n heap cells available, e has type A with n' free heap cells. The typing judgement is done according to a set of rules defined in the article, which we will also omit in this report, except for the pattern matching rules and the list constructor rule. We decided to show this last one because, while it may not be obvious in the pattern matching rules, most of the typing rules result in side inequalities, which we call *constraints*.

$$\frac{n \geq \mathbf{SIZE}(A \otimes L(A,k)) + k + n'}{\Gamma, x_h : A, x_t : L(A,k), n \vdash_\Sigma \mathsf{cons}(xh, xt) : L(A,k), n'} \text{ (Cons)}$$

$$\frac{\begin{array}{c}\Gamma, n \vdash_\Sigma e_1 : C, n' \\ \Gamma, x_h : Ax_t : L(A,k), n + \mathbf{SIZE}(A \otimes L(A,k)) + k \vdash_\Sigma e_1 : C, n'\end{array}}{\Gamma, x_h : A, x_t : L(A,k), n \vdash_\Sigma \mathsf{match}\ x\ \mathsf{with}\ \mathsf{cons}(x_1, x_2)\ \texttt{->}\ e_1\ \texttt{|}\ \mathsf{nil}\ \texttt{->}\ e_2 : C, n'} \text{ (LIST-ELIM)}$$

$$\frac{\Gamma, n \vdash_\Sigma e_1 : C, n' \quad \Gamma, x_h : Ax_t : L(A,k), n \vdash_\Sigma e_1 : C, n'}{\Gamma, x_h : A, x_t : L(A,k), n \vdash_\Sigma \mathsf{match}\ x\ \mathsf{with}\ \mathsf{cons}(x_1, x_2)\ \texttt{->}\ e_1\ \texttt{|}\ \mathsf{nil}\ \texttt{->}\ e_2 : C, n'} \text{ (LIST-ELIM')}$$

It is important to notice that the annotated type derivation T' of some program P is determined by its underlying non-annotated type derivation T. It is important because it means that, if we want to derive T', all that needs to be done it find the numerical annotations such that the constraints resulting of the derivation are satisfied. These constraints are linear, which means that the inference of annotations can be reduced to a linear optimization problem, and can be handled by an LP solver.

Let us consider a function f of type $L(A), L(A) \rightarrow L(A)$ and assume we have infered the annotations such that the annotated type of f is $L(A,4), L(A,0), 1 \rightarrow L(A,1), 2$. This means that the function call $f(l_1, l_2)$ evaluates to l_3 provided that there are at least $1 + 4 * |l_1|$ available heap cells, and that after the evaluation there are at least $2 + 1 * |l_3|$ free heap cells available.

As a support to the study of this article, we implemented a prototype based on the system presented. The development and implementation are discussed in chapter 4.

While the article presents an efficient solution in the context of first-order functions, it lacks to give a solution that extends to higher-order functions. This issue is approached in following articles that we have mentioned above, one of them [20] which we will discuss briefly in the next section.

2.4.2 Higher-Order Languages

The previous section described an approach to first-order languages. This work was eventually extended to higher-order functions in [20] and further developed in Jost's PhD thesis [19]. In 2016 it was extended for polynomial bounds [10].

When dealing with higher-order programs, the control flow is more difficult to determine. It becomes necessary to deal with closures, which are records that store a function together with an environment.

To address this new paradigm, the function call $f(x_1, \ldots, x_n)$ we saw in the previous section becomes obsolete, and to replace it, they consider a general application $(x1\ x2\ )$ and introduce a general lambda abstraction $(\lambda x.e)$.

The type rule pertaining to abstraction is worth explaining. Here we present a simplified adaptation of the abstraction rule in [20], that captures an important behaviour.

$$\frac{x{:}A \left|\frac{q}{q'}\right. e{:}B \qquad dom(\Gamma) = \mathrm{FV}(e) \setminus \{x\} \qquad \Gamma \curlyvee \{\Gamma, \Gamma\}}{\Gamma \left|\frac{\text{cost of abstraction}}{0}\right. \lambda x.e{:}A \xrightarrow[q']{q} B} \ \textsc{Abs}$$

In this judgement, Γ represents a typing environment, $A \xrightarrow[q']{q} B$ represents the type of a function with $q - q'$ constant cost, and $\mathrm{FV}(e)$ is the set of free variables of the expression e.

Because the potential stored in the function closure becomes available for each application of that function, to ensure that potential is not used more than once, the potential stored in a closure is restricted to be 0. This allows the unlimited repeated application of functions. This can be achieved by stating that the context Γ shares to itself and this restriction is represented by the rule $\Gamma \curlyvee \{\Gamma, \Gamma\}$. The relation $\curlyvee$ will be better defined later in this thesis, but in this context it essentially just generates the constraint $x = x + x$ for each resource variable in Γ, forcing them all to be zero. All the potential required for the function body must then be provided by its arguments (except for a constant amount, given by the arrow annotations).

The body of a function is only analysed once. A set of constraints is associated with the function when analysed, and these constraints are copied from the type at each application. If some resource variable only occurs in a function's type and constraints, it is given a fresh name for each application, and because of this, the LP-solver may choose a different solution for each individual application of the same function.

2.4.3 Polynomial Potential

In this section, we briefly explain Hoffman's approach to polynomial potential [12]. We go over the main contributions of this system and what influenced our approach.

This article presents a technique for inferring polynomial bounds, that still relies only on linear constraints. This is a very important feature because, until then, it was considered that the dependence on linear programming imposed a limitation to linear bounds.

One key aspect of this work is the use of binomial coefficients as a basis for polynomials, rather than the more common monomial basis x^n for $n >= 0$ (i.e. representing potential like this $\sum_{i=1}^{k} q_i \binom{n}{i}$ rather than this $\sum_{i=1}^{k} q_i.n^i$).

First, let us consider a list of type $L^{\vec{p}}(A)$. This is a simple list type, refined with a resource annotation $\vec{p} = (p_1, \ldots, p_k)$, where $(p_1, \ldots, p_k)$ represents a vector of coefficients that will be used to calculate the potential of the list. We can translate this annotated type to: the number q_1 is the potential assigned to every element of the list, q_2 is the potential assigned to every element of every suffix of the list, q_3 is the potential assigned to every element of every suffix of the suffixes of the list, and so on.

The main advantage of using binomial coefficients is the fact that it simplifies the definition of the *additive shift*. The additive shift is an operation on the coefficients

$$\frac{\vec{p} = (p_1, \ldots, p_k)}{\Sigma; x_h{:}A, x_t{:}L^{\lhd(\vec{p})} \left|\frac{p_1+K^{cons}}{0}\right. \mathsf{cons}(x_h, x_t){:}L^{(\vec{p})}(A)} \text{ T:Cons}$$

$$\frac{\begin{array}{c}\vec{p} = (p_1, \ldots, p_k) \\ \Sigma; \Gamma; x_h{:}A, x_t{:}L^{\lhd(\vec{p})}(A) \left|\frac{q+p_1-K_1^{matC}}{q'+K_2^{matC}}\right. e_1{:}B \\ \Sigma; \Gamma \left|\frac{q-K^{nil}}{q'-K^{nil}}\right. e_2{:}B\end{array}}{\Sigma; x{:}L^{(\vec{p})} \left|\frac{q}{q'}\right. \mathsf{match}\ x\ \mathsf{with}\ \mathsf{cons}(x_h, x_t)\ \texttt{->}\ e_1\ \texttt{|}\ \mathsf{nil}\ \texttt{->}\ e_2{:}B} \text{ T:MatL}$$

Figure 2.4: Rules T:Cons and T:MatL

represented by a resource annotation, that corresponds to the change in potential for typing branches of a pattern match. Let us consider a vector of coefficients $\vec{p} = (p_1, p_2, \ldots, p_k)$, the additive shift of vector $\vec{p}$ is

$$\lhd\vec{p} = (p_1 + p_2, p_2 + p_3, \ldots, p_{k-1} + p_k, p_k)$$

The idea is that the potential assigned to the tail $xs{:}L^{\lhd\vec{p}}$ of a list $x :: xs{:}L^{\vec{p}}$ is used to pay for recursive calls, calls to auxiliary functions and constant costs before and after recursive calls.

Similarly to the other works on AARA, the inference of constraints on the resource annotations is done during type inference, so it is also important to explain how these concepts were introduced in the type rules and why. As mentioned, the additive shift allows the typing of the branches of a pattern match, so naturally, we see these concepts arise in match rules and constructor rules. In his analysis, Hoffman works with list and tree data structures, but because we only consider lists in our analysis, we are only interested in the rules written for lists. We can see them in Fig. 2.4.

Some things to mention before explaining the particularities of these rules, note how the turnstile is annotated with values, one above and another below. Those are the values that keep track of resource usage during type inference. To be more specific, a judgement of the form $\Gamma \left|\frac{z'}{z}\right. e{:}C$ can be read as: considering a typing environment Γ and with z resource units available, we can infer the type C for the expression e and infer that the evaluation of e consumes $z - z'$ resource units.

T:Cons infers the type of a list constructor and illustrates the fact that one has to pay for the potential that is assigned to the new list. To do so, they require that the tail of the list x_t is typed with the additive shift of the potential of the new list and that there are p_1 resource units available. The parameter K^{cons} is a parametric constant, it is there to formalise the fact that we need to pay for the cost of allocating space

for the new list. The rule T:MatL complements T:Cons, and shows how to use the potential of a list to pay for resource usage, particularly in the "cons" branch. The tail of the list is annotated with the additive shift of the potential of the list, allowing recursive calls (with annotation $\vec{p}$) and calls to auxiliary functions (with annotation $(p_2, p_3, \ldots)$), furthermore, p_1 resource units become directly available.

To summarise, we have explained the idea behind the additive shift and described how Hoffmann introduced it in the type inference rules. The way it is inserted into the type system through a vector of coefficients, and the way the type rules use these values during inference is used in our system in a mostly identical manner.

2.4.4 Lazy Evaluation

In [22], Jost et al. approach the problem of inferring strict cost bounds for lazy functional languages by taking advantage of an AARA system to keep track of resource usage. In this section, much like in the previous one, we briefly explain this approach, focusing mainly on the key points that we took advantage of for our system.

The main contributions of this system deal with the particularities of the mechanics that define lazy evaluation, namely, how it delays the evaluation of arguments and uses references to prevent multiple evaluations of the same terms.

One very important contribution is the introduction of an annotated *thunk* structure to the type system. This structure essentially denotes a delayed evaluation of a term and maintains the cost of evaluating the delayed term. $\mathsf{T}^p(A)$ means: to evaluate the delayed expression of type A, we need p resource units available.

The use of resource annotations is also crucial, much like in other AARA systems. They are used during type inference to keep track of the resource usage of an expression, and attached to the types of functions to denote the overall cost evaluating the function.

$$\Gamma \,\Big|^{z'}_{z}\, e{:}C$$

This judgement means, under the environment Γ and with z resource unit available, the evaluation of e has type C and leaves z' resource units available.

Finally and possibly the most important contribution, the type rule Prepay. This is a structural rule that allows the cost of a thunk to be paid in advance, preventing that same cost to be accounted in further uses of the same thunk, "simulating" this way the memoization of a call-by-need evaluation.

$$\frac{\Gamma, x{:}\mathsf{T}^{q_0}(A) \,\Big|\!\frac{p}{p'}\; e{:}\; C}{\Gamma, x{:}\mathsf{T}^{q_0+q_1}(A) \,\Big|\!\frac{p+q_1}{p'}\; e{:}\; C} \qquad (\text{PREPAY})$$

These are the main points that we considered to understand how we could handle lazy evaluation in our analysis. Supplementary to these elements, we also took advantage of most syntactic and semantic choices of this article to write our system and the language that supports it. We will come back to these choices next when we explain our language and operational semantics.

2.4.5 Other Related Work

Here we provide a brief overview of some other works related to AARA that were not directly relevant for our development, but add to the knowledge of how AARA has developed along the years and how it can be adopted in different contexts.

Following work [15] used the AARA approach to obtain heap space requirements for Java-like programs with explicit deallocations. The data is assigned a potential related to its input and layout, and the allocations are then payed with this potential. This way, the potential provides an upper bound on the heap space usage for the given input. Whereas in the previous work a refined type consisted of a simple type together with a number, object-oriented languages require a more complex approach due to aliasing and inheritance, and so a refined type in this context consists of a number together with refined types for the attributes and methods.

Atkey [3] presented a system that extends AARA to pointer-manipulation languages by embedding a logic of resources based on intuitonistic logic of bunched implications within separation logic.

In [9] it is studied how AARA can be used to derive worst case resource usage for procedures with several arguments, and the previous inference of bounds is generalised for arbitrary multivariate polynomials (with limits like $m * n$). The drawbacks of a univariate analysis are the fact that many functions have multivariate characteristics, and the fact that, if data from different sources is interconnected in a program, multivariate bounds like $(m + n)^2$ will appear.

In 2016, Hoffmann et. al [10] presented a resource analysis system based on AARA that derives worst case bounds for higher-order programs with user defined inductive types. The derived bounds are multivariate resource polynomials. The system was

integrated with Inria's OCaml compiler and reuses the parser and type inference from this compiler.

This chapter provided a revision of relevant background concepts for this thesis. Concepts related to the lambda-calculus were addressed, as well as type-based analysis and amortisation, two crucial techniques in the context of this thesis. A review of previous works on AARA was also presented, with focus on the analysis systems for lazy evaluation with linear bounds and strict evaluation with polynomial bounds, the two main focal points of this thesis.

Chapter 3

Polynomial Analysis for Lazy Evaluation

In this chapter, we explain the approach we took to reach our main contribution: extend the current lazy evaluation analysis system to consider polynomial potential. In summary, we present a lazy functional language and its operational semantics (against which the analysis is done), then we explain the type inference rules needed to analyse the resource usage, and finally we provide a detailed demonstration of how the analysis can be done with some examples.

3.1 A Motivating Example

We have explained the motivation behind the need to extend the current resource analysis systems for lazy evaluation to consider polynomial potential, but to understand what that means at a practical level, let us take a look back at the function *pairs* that we presented in the introduction.

```
pairs ::[ a]  −> [(a, a)]
pairs  []  = []
pairs  (x:xs)  = (attach x  xs)++(pairs xs)
```

```
attach :: a  −> [a]  −> [(a, a)]
attach  _  []  = []
attach  y  (x:xs)  = (x,y ):( attach  x  ys)
```

With the aid of two auxiliary functions, attach (illustrated above) and ++ (Haskell's append), pairs takes a list as an argument and computes a list of pairs that are two-element sub-lists of the given list. This function iterates over every sub-list of the input list, meaning that this is a function with *polynomial potential*. This also means

that the system we studied for *lazy evaluation* is not able to derive bounds for this function, which is very undesirable since it imposes a great limitation to that system.

To understand how we should approach this problem, we studied how we could use J. Hoffman's [12] approach for strict languages (the approach that we have explained in Section 2.3.4), and combine it with Jost's system [22] (explained in section 2.3.3), to obtain a resource analysis system for lazy languages with polynomial potential.

3.2 A Lazy Functional Language

In this section, we present the language and operational semantics against which our analysis is done.

We start by introducing a *simple lazy functional language* (SLFL) composed by the syntactical terms e and w, presented in Fig. 3.1. Our expressions e include variables, lambda expressions, list constructors, let-expressions, and pattern matching. The values w are in weak head normal form and include constant values, pairs, list constructors and lambda expressions. To simplify the presentation of our expressions, sometimes we will be using a semicolon instead of in in let-expressions.

$$
\begin{array}{rcl}
e & ::= & c \;\mid\; \lambda x.\, e \;\mid\; e\; y \;\mid\; \mathsf{let}\; x = e_1 \;\mathsf{in}\; e_2 \\
 & \mid & (x_1, x_2) \;\mid\; \mathsf{cons}(x_h, x_t) \;\mid\; \mathsf{nil} \\
 & \mid & \mathsf{match}\; e_0 \;\mathsf{with}\; (x_1, x_2)\; \texttt{->}\; e_1 \\
 & \mid & \mathsf{match}\; e_0 \;\mathsf{with}\; \mathsf{cons}(x_h, x_t)\; \texttt{->}\; e_1 \;\texttt{|}\; \mathsf{nil}\; \texttt{->}\; e_2 \\
 & & \\
w & ::= & c \;\mid\; \lambda x.\, e \;\mid\; (x_1, x_2) \;\mid\; \mathsf{cons}(x_h, x_t) \;\mid\; \mathsf{nil}
\end{array}
$$

Figure 3.1: Syntax for SLFL expressions and normal forms

As mentioned, our syntax and cost model are largely based on Jost et al.'s semantics [22], which in its turn, is based on Sestof's revision [29] of Launchbury's operational semantics for lazy evaluation [23]. The main difference is the restriction to list and pairs constructors rather than more general recursive types. This was done to simplify the presentation, and we believe it would be a straightforward task to extend this system to more general data structures.

3.2.1 Operational Semantics

Here we present the rules that define the operational semantics for SLFL.

Before we explore the rules in more detail, it is important to explain the structure of our judgements and its meaning:

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \,\Big|\frac{m}{m'}\, e \Downarrow w, \mathcal{H}'$$

The relation can be read as follows: under a heap $\mathcal{H}$, a set of bound variables $\mathcal{S}$ and a set of locations $\mathcal{L}$, an expression e is evaluated to value w, in weak head normal form, consuming $m - m'$ resource units and producing a new heap $\mathcal{H}$'. The semantic rules in Fig. 3.2 illustrate how an expression is evaluated.

A heap $\mathcal{H}$ is a mapping from variables to *thunks*. As was mentioned in Section 2, a *thunk* is a delayed evaluation of an expression, meaning that our heap saves expressions that are possibly not yet evaluated. A set of locations $\mathcal{L}$ is used to keep track of the locations of the expressions that are being evaluated (See rule $\text{VAR}_{\Downarrow}$), this is done to prevent cyclic evaluation. We also use a set of variables $\mathcal{S}$ to keep track of bound variables.

The operational semantics is instrumented by a counting mechanism that keeps track of resource usage for each expression. The resource usage tracked in these rules is the target of our cost analysis. For simplicity, we decided that our analysis would only be interested in calculating cost bounds on the number of allocations used in an expression. Note that, however, the system could easily be extended to consider multiple cost parameters, such as the number of steps, number of applications, and others. This could be done by assigning different constants to each reduction rule to specify how many resource units should be available when considering a specific cost parameter. We can see this parametrization be used in Hoffman's [12] and Jost et al.'s [22] analyses. In our system we consider only one constant, 1, in the reduction rules LET and LETCONS.

Discussing the evaluation rules As mentioned above, these rules are largely based on the semantics from [22], their construction and meaning are mostly identical. The main differences can be seen in the definition for rules $\text{MATCH-L}_{\Downarrow}$, $\text{MATCH-P}_{\Downarrow}$ and $\text{LETCONS}_{\Downarrow}$.

Rule $\text{WHNF}_{\Downarrow}$: A lambda expression, a constructor and a constant are already final values so they evaluate to themselves and leave the heap unmodified. This incurs no

$$\frac{}{\mathcal{H}, \mathcal{S}, \mathcal{L} \mid\!\frac{m}{m}\; w \Downarrow w, \mathcal{H}} \quad (\text{WHNF}_{\Downarrow})$$

$$\frac{\mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{l\} \mid\!\frac{m}{m}\; e \Downarrow w, \mathcal{H}'}{\mathcal{H}[l \to e], \mathcal{S}, \mathcal{L} \mid\!\frac{m}{m'}\; l \Downarrow w, \mathcal{H}'[l \to w]} \quad (\text{Var}_{\Downarrow})$$

$$\frac{l \text{ is fresh} \qquad \mathcal{H}[l \to e_1[l/x]], \mathcal{S}, \mathcal{L} \mid\!\frac{m}{m'}\; e_2[l/x] \Downarrow w, \mathcal{H}'}{\mathcal{H}, \mathcal{S}, \mathcal{L} \mid\!\frac{m+1}{m'}\; \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow w, \mathcal{H}'} \quad (\text{Let}_{\Downarrow})$$

$$\frac{\mathcal{H}, \mathcal{S}, \mathcal{L} \mid\!\frac{m}{m'}\; e \Downarrow \lambda x.\ e', \mathcal{H}' \qquad \mathcal{H}', \mathcal{S}, \mathcal{L} \mid\!\frac{m'}{m''}\; e'[y/x] \Downarrow w, \mathcal{H}''}{\mathcal{H}, \mathcal{S}, \mathcal{L} \mid\!\frac{m}{m''}\; e\ y \Downarrow w, \mathcal{H}''} \quad (\text{App}_{\Downarrow})$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{S} \cup (\{x_1, x_2\} \cup \mathsf{BV}(e_1) \cup \mathsf{BV}(e_2)), \mathcal{L} \mid\!\frac{m}{m'}\; e_0 \Downarrow \mathsf{cons}(l_1, l_2), \mathcal{H}' \\ \mathcal{H}', \mathcal{S}, \mathcal{L} \mid\!\frac{m'}{m''}\; e_1[l_1/x_1, l_2/x_2] \Downarrow w, \mathcal{H}''\end{array}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \mid\!\frac{m}{m''}\; \mathsf{match}\ e_0\ \mathsf{with}\ \mathsf{cons}(x_1, x_2)\ \texttt{->}\ e_1\ \texttt{|}\ \mathsf{nil}\ \texttt{->}\ e_2 \Downarrow w, \mathcal{H}''} \quad (\text{Match-L}_{\Downarrow})$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{S} \cup (\{x_1, x_2\} \cup \mathsf{BV}(e_1) \cup \mathsf{BV}(e_2)), \mathcal{L} \mid\!\frac{m}{m'}\; e_0 \Downarrow \mathsf{nil}, \mathcal{H}' \\ \mathcal{H}', \mathcal{S}, \mathcal{L} \mid\!\frac{m'}{m''}\; e_2 \Downarrow w, \mathcal{H}''\end{array}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \mid\!\frac{m}{m''}\; \mathsf{match}\ e_0\ \mathsf{with}\ \mathsf{cons}(x_1, x_2)\ \texttt{->}\ e_1\ \texttt{|}\ \mathsf{nil}\ \texttt{->}\ e_2 \Downarrow w, \mathcal{H}''} \quad (\text{Match-N}_{\Downarrow})$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{S} \cup (\{x_1, x_2\} \cup \mathsf{BV}(e_1) \cup \mathsf{BV}(e_2)), \mathcal{L} \mid\!\frac{m}{m'}\; e_0 \Downarrow (l_1, l_2), \mathcal{H}' \\ \mathcal{H}', \mathcal{S}, \mathcal{L} \mid\!\frac{m'}{m''}\; e_1[l_1/x_1, l_2/x_2] \Downarrow w, \mathcal{H}''\end{array}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \mid\!\frac{m}{m''}\; \mathsf{match}\ e_0\ \mathsf{with}\ (x_1, x_2)\ \texttt{->}\ e_1 \Downarrow w, \mathcal{H}''} \quad (\text{Match-P}_{\Downarrow})$$

Figure 3.2: Operational semantics for SLFL

cost.

Rule $\text{VAR}_{\Downarrow}$: A variable l that is linked to an expression e in the initial heap, evaluates to a value w if the evaluation of e reaches that same value. The final heap will have the expression e that is linked to l, replaced by the value w, this way we avoid re-evaluations of e, obtaining *lazy evaluation*. This means that the cost of evaluating a variable is the cost of evaluating the expression that is associated with it.

Rule $\text{LET}_{\Downarrow}$: the expression e_1 bound to x is not evaluated, instead a thunk is allocated and associated with a fresh location l in the heap. The rules proceed to evaluate the expressions e_2. Because the purpose of our analysis is to infer cost bounds on the number of allocations, the evaluation of these rules needs to cost at least 1 resource unit, plus the cost of evaluating e_2.

Rule $\text{MATCH-P}_{\Downarrow}$ and $\text{MATCH-L}_{\Downarrow}$: In both these rules, the variables bound by the pattern matching are replaced in each branch by the respective locations that result from the evaluation of e_0 and are stored in the heap. The final value and heap are the result of evaluating the branch taken.

Example 3.2.1. Consider the term:

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

.

We can see how this term evaluates to $\lambda x.\, x$ under the rules of Fig. 3.2, leaving a heap $\Theta = [l_1 \to \lambda y.y, l_2 \to \lambda x.x, l_3 \to \lambda x.x]$.

$$\begin{array}{lrr}
[l_1 \to \lambda y.y, l_2 \to \lambda x.x, l_3 \to l_1\ l_2\ ] \left|\frac{0}{0}\right. \lambda x.x_{\Downarrow}\lambda x.x, [l_1 \to \lambda y.y] & \text{WHNF}_{\Downarrow} & (1)\\
[l_1 \to \lambda y.y, l_2 \to \lambda x.x, l_3 \to l_1\ l_2\ ] \left|\frac{0}{0}\right. l_{2\Downarrow}\lambda x.x, [l_1 \to \lambda y.y, l_2 \to \lambda x.x] & \text{VAR}_{\Downarrow}\ (1) & (2)\\
[l_1 \to \lambda y.y, l_2 \to \lambda x.x, l_3 \to l_1\ l_2\ ] \left|\frac{0}{0}\right. \lambda y.y_{\Downarrow}\lambda y.y, [l_1 \to \lambda y.y] & \text{WHNF}_{\Downarrow} & (3)\\
[l_1 \to \lambda y.y, l_2 \to \lambda x.x, l_3 \to l_1\ l_2\ ] \left|\frac{0}{0}\right. l_1 \Downarrow \lambda y.y, [l_1 \to \lambda y.y] & \text{VAR}_{\Downarrow}\ (3) & (4)\\
[l_1 \to \lambda y.y, l_2 \to \lambda x.x, l_3 \to l_1\ l_2\ ] \left|\frac{0}{0}\right. l_1\ l_2\ \Downarrow \lambda x.x, [l_1 \to \lambda y.y, l_2 \to \lambda x.x] & & \\
 & \text{APP}_{\Downarrow}\ (4{,}2) & (5)\\
[l_1 \to \lambda y.y, l_2 \to \lambda x.x, l_3 \to l_1\ l_2\ ] \left|\frac{0}{0}\right. l_{3\Downarrow}\lambda x.x, [\ldots, l_3 \to \lambda x.x] & \text{VAR}_{\Downarrow}\ (5) & (6)\\
[l_1 \to \mathsf{let}\ z = z;\ (\lambda x.\lambda y.y)\ z\ , \ldots, l_4 \to z] \left|\frac{0}{0}\right. \lambda y.y_{\Downarrow}\lambda y.y & \text{WHNF}_{\Downarrow} & (7)\\
[l_1 \to \mathsf{let}\ z = z;\ (\lambda x.\lambda y.y)\ z\ , \ldots, l_4 \to z] \left|\frac{0}{0}\right. (\lambda x.\lambda y.y)_{\Downarrow}\lambda x.\lambda y.y & \text{WHNF}_{\Downarrow} & (8)\\
[l_1 \to \mathsf{let}\ z = z;\ (\lambda x.\lambda y.y)\ z\ , \ldots, l_4 \to z] \left|\frac{0}{0}\right. (\lambda x.\lambda y.y)\ l_4\ {}_{\Downarrow}\lambda y.y & \text{APP}_{\Downarrow}\ (8{,}7) & (9)\\
[l_1 \to \mathsf{let}\ z = z;\ (\lambda x.\lambda y.y)\ z\ , \ldots] \left|\frac{1}{0}\right. \mathsf{let}\ z = z;\ (\lambda x.\lambda y.y)\ z\ {}_{\Downarrow}\lambda y.y & \text{LET}_{\Downarrow}\ (9) & (10)\\
[l_1 \to \mathsf{let}\ z = z;\ (\lambda x.\lambda y.y)\ z\ , \ldots] \left|\frac{1}{0}\right. l_{1\Downarrow}\lambda y.y, [l_1 \to \lambda y.y] & \text{VAR}_{\Downarrow}\ (10) & (11)
\end{array}$$

$$[l_1 \to \mathsf{let}\ z = z;\ (\lambda x.\lambda y.y)\ z\ , l_2 \to \lambda x.x, l_3 \to l_1\ l_2\ ] \Big|\!\frac{1}{0}\ l_1\ l_3 \Downarrow \lambda x.x, \Theta \qquad \textsc{App}_{\Downarrow}\ (11{,}6) \quad (12)$$

$$[l_1 \to \mathsf{let}\ z = z;\ (\lambda x.\lambda y.y)\ z\ , l_2 \to \lambda x.x] \Big|\!\frac{2}{0}\ \mathsf{let}\ v = l_1\ l_2\ ;\ l_1\ v \Downarrow \lambda x.x, \Theta \qquad \textsc{Let}_{\Downarrow}\ (12) \quad (13)$$

$$[l_1 \to \mathsf{let}\ z = z;\ (\lambda x.\lambda y.y)\ z\ ] \Big|\!\frac{3}{0}\ \mathsf{let}\ i = \lambda x.x;\ \mathsf{let}\ v = l_1\ i\ ;\ l_1\ v \Downarrow \lambda x.x, \Theta \qquad \textsc{Let}_{\Downarrow}\ (13) \quad (14)$$

$$\Big|\!\frac{4}{0}\ \mathsf{let}\ f = \mathsf{let}\ z = z;\ (\lambda x.\lambda y.y)\ z\ ;\ \mathsf{let}\ i = \lambda x.x;\ \mathsf{let}\ v = f\ i\ ;\ f\ v \Downarrow \lambda x.x, \Theta \qquad \textsc{Let}_{\Downarrow}\ (14) \quad (15)$$

3.3 Annotated Type System

In this section, we present our type system to analyse resource usage and provide a detailed description of how the analysis works using some illustrating examples.

3.3.1 Annotated Types

Here, we present the syntax for the annotated types of our language and the type rules used to perform the cost analysis. Types include primitives, function types, thunks, pairs and lists.

$$A,\ B \quad ::= \quad \mathsf{int}\ \mid\ A \xrightarrow{q} B\ \mid\ \mathsf{T}^q(A)\ \mid\ A \times B \mid \mathsf{L}^q(\vec{p}, A)$$

The variables q and $\vec{p}$ stand for *cost annotations*. More precisely, $\vec{p}$ stands for *list potential* and actually represents a vector of cost annotations, $\vec{p} = (p_1,\ \ldots,\ p_n)$.

The annotation q on function types is an upper bound on the cost of applying that function. Thunk types represent a delayed evaluation of an expression of type A and are also annotated with an upper bound on the cost of evaluating the delayed expression. List types are annotated with a simple annotation q, representing the cost of evaluating one constructor of the list, and a vector annotation $\vec{p}$, which represents the potential associated with that list. The primitive type int is free of cost annotations and type pairs is a pair of any type.

We define the additive shift of a vector of coefficients $\vec{p}$ as Hoffmann (see section 2.4.3):

$$\triangleleft(p_1, p_2, \ldots, p_n) = (p_1 + p_2,\ p_2 + p_3,\ \ldots,\ p_{n-1} + p_n,\ p_n)$$

We also define an addition operation on vectors of coefficients of equal length:

$$(p_1,\ \ldots,\ p_n) + (q_1,\ \ldots,\ q_n) = (p_1 + q_1,\ p_2 + q_2,\ \ldots,\ p_n + q_n)$$

In Fig. 3.3 and Fig. 3.4 we present the type rules used to derive these types and their cost annotations.

3.3.2 The Sharing Relation

Before we go on to explain how the type system works, it is important to explain the concept of *sharing*: $A \curlyvee \{B_1, \ldots, B_n\}$. In short, sharing allows the potential of a type A to be distributed amongst other types $\{B_1, \ldots, B_n\}$. The rules presented in Fig. 3.5 illustrate how the sharing relation applies depending on the types it is used on, and they follow very strictly the construction and explanation of the sharing rules presented in [22]. The main difference is present in the rule regarding list types (because Jost et al. system deals with possibly recursive algebraic data types, and not only lists). In our sharing relation, the SHARELIST rule allows for the potential of a certain list A, to be shared amongst types B_i.

3.3.3 Subtyping

The subtyping relation is a particular case of sharing. It allows us to relax the annotations associated to a type by requiring them to be greater or equal than those of that type. We say a type A_1 is a subtype of a type A when $A_1 <: A$, this relation could also be represented as $A_1 \curlyvee \{A, A'\}$, where A' is a type with annotations greater than or equal to zero. We can say that subtyping has the following properties:

$$\mathsf{int} <: \mathsf{int}$$

$$\mathsf{T}^{q_1}(A_1) <: \mathsf{T}^{q_2}(A_2) \qquad \text{if } q_1 \geq q_2 \text{ and } A_1 <: A_2$$

$$A_1 \times A_2 <: B_1 \times B_2 \qquad \text{if } A_1 <: B_1 \text{ and } B_1 <: B_2$$

$$A_1 \xrightarrow{q_1} B_1 <: A_2 \xrightarrow{q_2} B_2 \qquad \text{if } q_1 \geq q_2 \text{ and } A_1 <: A_2 \text{ and } B_2 <: B_1$$

$$\mathsf{L}^{q_1}(\vec{p_1}, A_1) <: \mathsf{L}^{q_2}(\vec{p_2}, A_2) \qquad \text{if } q_1 \geq q_2 \text{ and } \vec{p_1} \geq \vec{p_2} \text{ and } A_1 <: A_2$$

We say $\vec{q} \geq \vec{p}$ if, $|\vec{q}| = |\vec{p}| = n$ and $\forall_{1 \leq i \leq n}, q_i \geq p_1$.

$$\frac{}{\left|\frac{0}{0}\right. n : \mathsf{int}} \quad \text{(Const)}$$

$$\frac{}{x{:}\mathsf{T}^p(A) \left|\frac{p}{0}\right. x : A} \quad \text{(Var)}$$

$$\frac{\Gamma \left|\frac{z}{z'}\right. e : A \xrightarrow{p} C}{\Gamma, y : A \left|\frac{z+p}{z'}\right. e\ y : C} \quad \text{(App)}$$

$$\frac{\Gamma, x{:}A \left|\frac{p}{0}\right. e : C \quad x \notin \Gamma \quad \Gamma \curlyvee \{\Gamma, \Gamma\}}{\Gamma \left|\frac{0}{0}\right. \lambda x.e : A \xrightarrow{p} C} \quad \text{(Abs)}$$

$$\frac{\begin{array}{c} A \curlyvee \{A, A'\} \qquad x \notin \{\Gamma, \Delta\} \qquad e_1 \text{ is not a constructor} \\ \Gamma, x : \mathsf{T}^0(A') \left|\frac{p}{0}\right. e_1 : A \qquad \Delta, x : \mathsf{T}^p(A) \left|\frac{z}{z'}\right. e_2 : C \end{array}}{\Gamma, \Delta \left|\frac{z+1}{z'}\right. \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : C} \quad \text{(Let)}$$

$$\frac{\begin{array}{c} \vec{q} = (q_1, \ldots, q_k) \quad A = \mathsf{L}^p(\vec{q}, B) \quad A \curlyvee \{A,\ A'\} \\ \Gamma, x : \mathsf{T}^0(A') \left|\frac{0}{0}\right. \mathsf{cons}(x_h, x_t) : A \qquad \Delta, x : \mathsf{T}^0(A) \left|\frac{z}{z'}\right. e : C \end{array}}{\Gamma, \Delta \left|\frac{z+1+q_1}{z'}\right. \mathsf{let}\ x = \mathsf{cons}(x_h, x_t)\ \mathsf{in}\ e : C} \quad \text{(Letcons)}$$

$$\frac{}{x_1{:}A_1, x_2{:}A_2 \left|\frac{0}{0}\right. (x_1, x_2) : A_1 \times A_2} \quad \text{(Pair)}$$

$$\frac{}{\left|\frac{0}{0}\right. \mathsf{nil} : \mathsf{L}^q(\vec{p}, A)} \quad \text{(Nil)}$$

$$\frac{}{x_h{:}B, x_t{:}\mathsf{T}^p(\mathsf{L}^p(\lhd\vec{q}, B)) \left|\frac{0}{0}\right. \mathsf{cons}(x_h, x_t) : \mathsf{L}^p(\vec{q}, B)} \quad \text{(Cons)}$$

$$\frac{\Gamma \left|\frac{z}{z'}\right. e_0 : A_1 \times A_2 \qquad \Delta,\ x_1 : A_1,\ x_2 : A_2 \left|\frac{z'}{z''}\right. e_1 : C}{\Gamma, \Delta \left|\frac{z}{z''}\right. \mathsf{match}\ e_0\ \mathsf{with}\ (x_1, x_2)\ \texttt{->}\ e_1 : C} \quad \text{(Match-P)}$$

$$\frac{\begin{array}{c} \vec{q} = (q_1, \ldots, q_k) \\ \Gamma \left|\frac{z}{z'}\right. e_0 : \mathsf{L}^p(\vec{q}, A) \\ \Delta, x_h : A, x_t : \mathsf{T}^p(\mathsf{L}^p(\lhd\vec{q}, A)) \left|\frac{z'+q_1}{z''}\right. e_1 : C \\ \Delta \left|\frac{z'}{z''}\right. e_2 : C \end{array}}{\Gamma, \Delta \left|\frac{z}{z''}\right. \mathsf{match}\ e_0\ \mathsf{with}\ \mathsf{cons}(x_h, x_t)\ \texttt{->}\ e_1\ \texttt{|}\ \mathsf{nil}\ \texttt{->}\ e_2 : C} \quad \text{(Match-L)}$$

Figure 3.3: Syntax directed type rules

$$\frac{\Gamma, x{:}\mathsf{T}^{q_0}(A) \left|\frac{p}{p'}\right. e{:}\ C}{\Gamma, x{:}\mathsf{T}^{q_0+q_1}(A) \left|\frac{p+q_1}{p'}\right. e{:}\ C} \quad (\textsc{Prepay})$$

$$\frac{\Gamma \left|\frac{p}{p'}\right. e{:}C}{\Gamma, x{:}A \left|\frac{p}{p'}\right. e{:}\ C} \quad (\textsc{Weak})$$

$$\frac{\Gamma, x{:}A_1, x{:}A_2 \left|\frac{p}{p'}\right. e{:}\ C \quad A \curlyvee \{A_1, A_2\}}{\Gamma, x{:}A \left|\frac{p}{p'}\right. e{:}\ C} \quad (\textsc{Share})$$

$$\frac{\Gamma \left|\frac{p}{p'}\right. e{:}\ A \quad q \geq p \quad q - p \geq q' - p'}{\Gamma \left|\frac{q}{q'}\right. e{:}\ A} \quad (\textsc{Relax})$$

$$\frac{\Gamma \left|\frac{p}{p'}\right. e{:}\ A \quad A <: B}{\Gamma \left|\frac{p}{p'}\right. e{:}\ B} \quad (\textsc{Subtype})$$

$$\frac{\Gamma, x{:}B \left|\frac{p}{p'}\right. e{:}\ C \quad A <: B}{\Gamma, x{:}A \left|\frac{p}{p'}\right. e{:}\ C} \quad (\textsc{Supertype})$$

Figure 3.4: Structural type rules

3.3.4 Type System

The type rules required for our analysis are presented in Fig. 3.3. These rules are complemented with the structural rules in Fig. 3.4, which introduce some flexibility to our analysis in ways that we will explain next. Our judgements have the form $\Gamma \left|\frac{p'}{p}\right. e : A$ and can be read as follows: considering a typing context Γ, and with p resource units available, we can derive the annotated type A for expression e, leaving p' resource units available. These rules result from combining the ones presented in the two previous systems [22, 12] While many rules are identical to previous work, there are important differences in rules that concern the use of potential, namely, Letcons, Cons and Match.

We now describe each rule informally, focusing on on how type annotations express resource usage. Recall that we consider cost bounds for the number of allocations, i.e. the number of let-expressions evaluated.

Rule Const does not consume any resources as evaluating a primitive value incurs no additional allocations.

Rule Var deals with the elimination of a thunk type, so it is necessary to pay for the

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

$$\frac{A \curlyvee \{A_1, \ldots, A_n\} \qquad B \curlyvee \{B_1, \ldots, B_n\}}{A \times B \curlyvee \{A_1 \times B_1, \ldots, A_n \times B_n\}} \quad (\textsc{SharePair})$$

$$\frac{B_i = \mathsf{L}^{p_i}(\vec{q_i}, A_i) \qquad A \curlyvee \{A_1, \ldots, A_n\} \qquad \vec{q} \geq \sum_{i=1}^{n} \vec{q_i} \qquad p_i \geq p}{\mathsf{L}^{p}(\vec{q}, A) \curlyvee \{B_1, \ldots, B_n\}} \quad (\textsc{ShareList})$$

$$\frac{A_i \curlyvee \{A\} \quad C \curlyvee \{C_i\} \quad q_i \geq p \quad (1 \leq i \leq n)}{A \xrightarrow{p} C \curlyvee \{A_1 \xrightarrow{q_1} C_1, \ldots, A_n \xrightarrow{q_n} C_n\}} \quad (\textsc{ShareFun})$$

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

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

$$\frac{A \curlyvee \{B_1, \ldots, B_n\} \quad \Gamma \curlyvee \Delta}{x\colon A, \Gamma \curlyvee (x\colon B_1, \ldots, x\colon B_n, \Delta)} \quad (\textsc{ShareCtx})$$

Figure 3.5: Sharing rules

cost associated with that thunk.

Rules LET and LETCONS deal with the allocation of a thunk for sub-expressions. Both rules require at least 1 unit to be available (corresponding to the newly allocated thunk) and recursive use of the bound variable x is allowed. Note also that the side condition $A \curlyvee \{A, \ldots, A'\}$ that guarantees that the type A' does not have potential is required to ensure soundness (so that self-referencing structures are assigned zero potential [22]). Rule LET allows the cost of e_1 to be paid for only once, even in the case of self-reference; the intuition for this is that any productive uses of the bound variable in self-referencing definitions must be to an evaluated form [32].

Rule LETCONS formalises the fact that one has to pay for the allocation of a new list constructor, which requires paying for the potential associated with the new list. We do so by requiring q_1 units to be available and complementing it with rule CONS, to be applied on the first expression e_1, which must be a list constructor.

Rules CONS and PAIR are simple references to a constructor so they do not consume any resources. In rule CONS we do require the tail of the list to be annotated with

the additive shift of its potential, complementing rule LETCONS.

Rule APP requires that the cost associated with a function is paid for each time the function is applied.

Rule ABS captures the cost of the expression in the type annotation of the function.

Rule MATCH-L shows how to use the potential of a list to pay for resource consumption. To do so, we require that the branch matching with the list constructor gains the excess potential q_1. We also annotate the tail of the list with the additive shift of the list potential, to allow future recursive calls or calls to auxiliary functions. This rule requires that both branches are of the same type C and that the amount of resources z'', available after the evaluation of each branch, is the same, which may require *relaxation* of the costs (See structural rule RELAX in Fig. 3.4).

Rule MATCH-P deals with pattern matching against a pair constructor. Like in MATCH-P, we require that both branches are of the same type C and that the amount of resources z'' is the same.

The structural rules in Fig.3.4 can be used in any part of a type derivation. As we have explained before, PREPAY allows the payment in advance of the cost associated with a thunk, preventing that same cost to be payed for more than once. WEAK introduces a new hypothesis in the environment. SHARE allows the use of the sharing rules in 3.5 to split potential in an hypothesis. RELAX, as the name indicates, allows the relaxation of cost bounds. SUBTYPE allows subtyping in a result, and SUPERTYPE allows supertyping in an environment.

3.4 Worked Examples

To better understand how the analysis works, let us take a look at some examples.

Example 3.4.1. Let us consider function *pairs* in Fig. 3.6. This function is a translation into SLFL of the example from Section 1.1. Function *pairs* takes a list as an argument and computes a list of pairs that are two-element sub-lists the given list, while function *attach* combines each element of a list with the first argument. Note that the auxiliary function app' is the translation of list append with the argument order flipped, i.e. $app' =$ `flip (++)`; this is done so that recursion is over the second argument and the type rules allow assigning potential to this argument.[1].

[1]In particular, the side condition for rule ABS requires that the typing context Γ has no potential.

$$
\begin{array}{l}
\mathit{attach} = \lambda n.\, \lambda l.\, \mathsf{match}\ l^{k_1}\ \mathsf{with} \\
\quad \mathsf{nil\text{-}{>}nil} \\
\quad \mathsf{cons}(x, xs^{j_1})\mathsf{\text{-}{>}}\ \mathsf{let}\ p = (x, n); f = \mathit{attach}\ n\ xs^{n_1} \\
\qquad\qquad \mathsf{in}\ \mathsf{cons}(p, f) \\
\mathit{app}' = \lambda l_1.\, \lambda l_2.\, \mathsf{match}\ l_2^{v_1}\ \mathsf{with} \\
\quad \mathsf{nil\text{-}{>}}l_1 \\
\quad \mathsf{cons}(x, xs^{w_1})\mathsf{\text{-}{>}}\ \mathsf{let}\ f = \mathit{app}'\ l_1\ xs^{m_1} \\
\qquad\qquad \mathsf{in}\ \mathsf{cons}(x, f) \\
\mathit{pairs} = \lambda l.\, \mathsf{match}\ l^{(q_1,q_2)}\ \mathsf{with} \\
\quad \mathsf{nil\text{-}{>}nil} \\
\quad \mathsf{cons}(x, xs^{(r_1,r_2)})\mathsf{\text{-}{>}}\ \mathsf{let}\ f_1 = \mathit{pairs}\ xs^{(s_1,s_2)}; \\
\qquad\qquad f_2 = \mathit{attach}\ x\ xs^{(p_1,p_2)} \\
\qquad\qquad \mathsf{in}\ \mathit{app}'\ f_1\ f_2
\end{array}
$$

Figure 3.6: Translation of the *pairs* function and auxiliary definitions into SLFL.

To facilitate the presentation of annotated type assignments, we have added potential annotations to list variables in Fig. 3.6: $l^{\vec{q}}$ means that variable l has type $\mathsf{L}^0(\vec{q}, B)$ for some B, i.e. l is a list with potential $\vec{q}$ and zero thunk cost for the spine. Since we expect function *pairs* has quadratic cost on the argument list length, we annotate it with pair of coefficients $\vec{q} = (q_1, q_2)$. Conversely, we expect functions *attach* and app' to have linear cost, hence we annotated these with a single coefficient.

Function app' is defined by structural recursion on the second argument l_2 and uses a single let-expression for each constructor in the argument; this means that l_2 should have a potential of at least 1 resource unit for each constructor. In *attach* we can see two let-expressions being used, which means the input potential should be at least 2. However, when analysing the body of function *pairs*, we can see that the output of *attach* is also the second input of app'. This means that to be able to type *pairs*, the output of *attach* must be compatible with the input of app', and because of that, its potential should be at least 1. Because the output potential needs to be accounted for in the input, we need to add it to the potential 2 we mentioned before.

Using the annotations for *attach* and app' in Fig. 3.6, we derive the following con-

straints:

$$
\begin{aligned}
j_1 &= k_1 && \text{(additive shift)}\\
j_1 &= n_1 && \text{(share)}\\
n_1 &= k_1 && \text{(recursive call)}\\
k_1 &\geq 2 + v_1
\end{aligned}
$$

(two let-expressions plus the potential of the output of *attach*/input of app')

$$
\begin{aligned}
w_1 &= v_1 && \text{(additive shift)}\\
w_1 &= m_1 && \text{(share)}\\
m_1 &= v_1 && \text{(recursive call)}\\
v_1 &\geq 1 && \text{(single let-expression)}
\end{aligned}
$$

We can solve this system of equations with $v_1 = m_1 = w_1 = 1$ and $q_1 = r_1 = s_1 = 3$ and derive the following annotated types:

$$
\begin{aligned}
app' &: \mathsf{T}^0(\mathsf{L}^0(0, B \times B)) \xrightarrow{0} \mathsf{T}^0(\mathsf{L}^0(1, B \times B)) \xrightarrow{0} \mathsf{L}^0(0, B \times B)\\
attach &: B \xrightarrow{0} \mathsf{T}^0(\mathsf{L}^0(3, B)) \xrightarrow{0} \mathsf{L}^0(1, B \times B)
\end{aligned}
$$

To better understand how the analysis works, we are going to illustrate the inference steps with more detail. The rules are applied in a very straightforward way, but it is important to pay attention to how resource usage is passed from and onto the judgements. Let us start by assuming:

$$
\begin{aligned}
\Gamma &= app' : \mathsf{T}^0(\mathsf{L}^0(0, B \times B)) \xrightarrow{0} \mathsf{T}^0(\mathsf{L}^0(1, B \times B)) \xrightarrow{0} \mathsf{L}^0(0, B \times B)\\
\Sigma &= attach : B \xrightarrow{0} \mathsf{T}^0(\mathsf{L}^0(3, B)) \xrightarrow{0} \mathsf{L}^0(1, B \times B)
\end{aligned}
$$

We will derive a type for *pairs* as follows:

$$
\Theta = pairs : \mathsf{T}^0(\underbrace{\mathsf{L}^0((q_1, q_2), B)}_{L_{In}}) \xrightarrow{p} \underbrace{\mathsf{L}^0((0,0), B \times B)}_{L_{Out}}
$$

For simplicity, sometimes we omit certain elements of the type context that are not needed for the derivation in question. We also divide the definition of *pairs* into two

sub-expressions as shown:

$$pairs = \lambda l.\ \overbrace{\textsf{match } l \textsf{ with} \quad \textsf{nil->nil} \quad \textsf{cons}(x, xs)\textsf{->}\ \overbrace{\textsf{let } f_1 = pairs\ xs;\ f_2 = attach\ x\ xs\ \textsf{in } app'\ f_1\ f_2}^{e_2}}^{e_1}$$

We start by stating the typing obligation for the outer part of the recursive definition:

$$\Gamma, \Sigma \left|\tfrac{1}{0}\right. \textsf{let } pairs = \lambda l.\, e_1 \textsf{ in } pairs{:}\mathsf{T}^0(L_{In}) \xrightarrow{p} L_{Out} \tag{3.1}$$

By rule LET, we need to prove:

$$\Gamma, \Sigma, \Theta \left|\tfrac{0}{0}\right. \lambda l.\, e_1 : \mathsf{T}^0(L_{In}) \xrightarrow{p} L_{Out} \tag{3.2}$$

The later follows from rule ABS if we prove:

$$\Gamma, \Sigma, \Theta,\ l{:}\mathsf{T}^0(L_{In}) \left|\tfrac{p}{0}\right. e_1 : L_{Out} \tag{3.3}$$

By rule MATCH-L we get three new obligations; the first two correspond to the scrutinised list and the right-hand side of nil-case:

$$l{:}\mathsf{T}^0(L_{In}) \left|\tfrac{0}{0}\right. l{:}L_{In} \tag{VAR}$$

$$\left|\tfrac{0}{0}\right. \textsf{nil}{:}L_{Out} \tag{NIL}$$

The remaining case for non-empty lists is:

$$\Gamma, \Sigma, \Theta,\ x{:}B,\ xs{:}\mathsf{T}^0(\mathsf{L}^0((q_1+q_2, q_2), B)) \left|\tfrac{q_1}{0}\right. e_2{:}L_{Out} \tag{3.4}$$

We now apply the SHARE rule to distribute the potential of the tail xs for the two uses in right-hand side expression e_2. The side condition is:

$$\mathsf{L}^0((q_1+q_2, q_2), B) \curlyvee \{\mathsf{L}^0((p_1, p_2), B),\ \mathsf{L}^0((s_1, s_2), B)\} \tag{3.5}$$

for some annotations p_1, p_2, s_1, s_2 such that $q_1 + q_2 \geq p_1 + s_1 \wedge q_2 \geq p_2 + s_2$. The two contexts are:

$$\Delta_1 = xs{:}\mathsf{T}^0(\mathsf{L}^0((s_1, s_2), B)) \quad \text{(for the recursive call to } pairs\text{)}$$

$$\Delta_2 = xs{:}\mathsf{T}^0(\mathsf{L}^0((p_1, p_2), B)) \quad \text{(for the call to } attach\text{)}$$

We can now type the recursive right-hand side e_2:

$$\Gamma, \Sigma, \Theta,\ x{:}B,\ \Delta_1, \Delta_2 \left|\tfrac{2}{0}\right. \begin{array}{l}\textsf{let } f_1 = pairs\ xs; \\ \quad\ \ f_2 = attach\ x\ xs \\ \textsf{in } app'\ f_1\ f_2\end{array} : L_{Out} \tag{3.6}$$

The cost annotation on the turnstile correspond to the two uses of *let* for f_1 and f_2, as will be confirmed from the remaining derivation. We continue by typing the bound sub-expressions:

$$\Theta, \Delta_1 \,\vert\!\frac{0}{0}\; \mathit{pairs}\ \mathit{xs} : \mathsf{L}^0((0,0), B \times B) \tag{3.7}$$

$$\Sigma, \Delta_2, x{:}B \,\vert\!\frac{0}{0}\; \mathit{attach}\ x\ \mathit{xs} : \mathsf{L}^0(0, B \times B) \tag{3.8}$$

Judgments (3.7) and (3.8) follow immediately from VAR and APP. Note that, while the annotations on the turnstile are zero, the uses of APP impose constraints on the annotations in Δ_1 and Δ_2: $p_1 = 3$, $p_2 = 0, s_1 = q_1$ and $s_2 = q_2$. It remains to type the inner expression:

$$\Delta_2, \Gamma, \Theta, f_1{:}\mathsf{T}^0(L_{Out}) \,\vert\!\frac{1}{0}\; \mathsf{let}\ f_2 = \mathit{attach}\ x\ \mathit{xs}\ \mathsf{in}\ \mathit{app}'\ f_1\ f_2{:}L_{Out} \tag{3.9}$$

This follows from the rules VAR and APP twice:

$$\Gamma, f_1{:}\mathsf{T}^0(L_{Out}), f_2{:}\mathsf{T}^0(\mathsf{L}^0(1, B \times B)) \,\vert\!\frac{0}{0}\; \mathit{app}'\ f_1\ f_2{:}L_{Out} \tag{3.10}$$

With this detailed illustration it is easy to see where the constraints mentioned before come from. From (3.7), (3.8) and (3.9) we get $p_1 = 3$, $p_2 = 0, s_1 = q_1$ and $s_2 = q_2$. From (3.4) and (3.6) we get $q_1 \geq 2$. From (3.5) we get that $q_1 + q_2 = s_1 + p_1$ and $q_2 = s_2 + p_2$. These constraints admit the solution $p_1 = s_2 = q_2 = 3$, $s_1 = q_1 = 2, p_2, p = 0$, giving us the following typing:

$$\mathit{pairs} : \mathsf{T}^0(\mathsf{L}^0((2,3), B)) \xrightarrow{0} \mathsf{L}^0(0, B \times B)$$

This typing ensures that *pairs* can be applied to an input list l with potential $2 \times |l| + 3 \times \binom{|l|}{2}$ leaving no leftover potential. This corresponds to a quadratic cost bound of $2 \times n + 3 \times \binom{n}{2} + 0 = 2 \times n + \frac{3}{2} \times n \times (n-1)$ expressed as a function of the input list length $n = |l|$.

Example 3.4.2. In the previous derivation we choose zero annotations for the thunks in the list spine; this corresponds to deriving a cost bound for the case where the spine of the input list is fully evaluated. Let us now consider the case where the input list l is annotated with $\mathsf{L}^1((q_1, q_2), B)$, i.e., evaluating each list successive constructor costs 1.

Because of the rule MATCH, when we introduce the tail element of the list to our environment it will be associated with a unitary cost thunk. We can use the structural rule PREPAY to pay for its thunk cost only once, rather than for each use, before using SHARE to duplicate it. Because the rule PREPAY is structural, we could have chosen not to use it and the inference would still have obtained an acceptable but less precise type.

Again, we are going to illustrate the inference steps with more detail. Note that, again,

we omit certain elements of the type context that are not needed for the derivation in question. The expression is divided into 3 sub-expressions as illustrated before.

As before we assume annotated type for the auxiliary functions:[2]

$$\Gamma = app' : \mathsf{T}^0(\mathsf{L}^0(0, B \times B)) \xrightarrow{0} \mathsf{T}^0(\mathsf{L}^0(1, B \times B)) \xrightarrow{0} \mathsf{L}^0(0, B \times B)$$
$$\Sigma = attach : B \xrightarrow{0} \mathsf{T}^0(\mathsf{L}^1(4, B)) \xrightarrow{0} \mathsf{L}^0(1, B \times B)$$

let us derive a type for *pairs* as follows:

$$\Theta = pairs : \mathsf{T}^p(\underbrace{\mathsf{L}^1((q_1, q_2), B)}_{L_{In}}) \xrightarrow{a} \underbrace{\mathsf{L}^0((0,0), B \times B)}_{L_{Out}}$$

The derivation is very similar to the previous example. It is when we reach the point of sharing the potential of the list that the main difference appears.

$$\Gamma, \Sigma, \Theta,\ x{:}B,\ xs{:}\mathsf{T}^p(\mathsf{L}^1((q_1 + q_2, q_2), B)) \,\Big|\!\frac{q_1}{0}\ e_2{:}L_{Out} \tag{3.11}$$

Because this time the list is associated with a unitary cost thunk rather than a 0 annotated thunk, if we applied the rule SHARE as before, that cost would be replicated for both lists, meaning that we would have to pay for both uses. To prevent this from happening, we use the structural rule PREPAY right before we use SHARE. We can see how the lists that result from sharing end up associated with a 0 annotated thunk:

$$\Gamma, \Sigma, x{:}B,\ xs{:}\mathsf{T}^1(\mathsf{L}^1((q_1 + q_2, q_2), B)) \,\Big|\!\frac{3}{0}\ e_2{:}L_{Out} \qquad \text{(Prepay)}$$
$$\Gamma, \Sigma, x{:}B,\ xs{:}\mathsf{T}^0(\mathsf{L}^1((q_1, q_2), B)) \,\Big|\!\frac{2}{0}\ e_2{:}L_{Out} \qquad \text{(Share)}$$

The use of SHARE creates the following condition:

$$\mathsf{T}^0(\mathsf{L}^1((q_1 + q_2, q_2), B)) \curlyvee \{\mathsf{T}^0(\mathsf{L}^1((p_1, p_2), B)),\ \mathsf{T}^0(\mathsf{L}^1((s_1, s_2), B))\} \tag{3.12}$$

Note that, although the outermost thunks have been reduced by the use of PREPAY, the list spine thunks still cost 1. This is because sharing distributes list potential but not thunk costs (See Fig. 3.5).

The remaining derivation is:

$$\Gamma, \Sigma, x{:}B,\ xs{:}\mathsf{T}^0(\mathsf{L}^1((p_1, p_2), B)),\ xs{:}\mathsf{T}^0(\mathsf{L}^1((s_1, s_2), B)) \,\Big|\!\frac{2}{0}\ e_2{:}L_{Out} \tag{3.13}$$

The main constraints that result from this derivation are very similar to the ones from the example above, with the exception of $p_1 = 4$ (because of the different type

[2]Note that we need a slightly different annotation for the input list of *attach*.

assumption for *attach*) and $q_1 \geq 3$ (because of the use of PREPAY after (3.13)). These constraints can be solved by $p_1 = s_2 = q_2 = 4$, $s_1 = q_1 = 3, p_2 = 0, p = 0$, giving us the type

$$pairs : \mathsf{T}^0(\mathsf{L}^1((3,4), B)) \xrightarrow{0} \mathsf{L}^0(0, B \times B)$$

This type corresponds to a cost bound of $3 \times n + 4 \times \binom{n}{2} + 0 = 3 \times n + 2 \times n \times (n-1)$ for list of length n.

3.5 Further Discussion

The analysis of these examples demonstrates how our system can infer reliable strict cost bounds. Although we believe we have reached positive results, there are some issues worth mentioning.

When comparing the result from Example 3.4.2 with the bound obtained for Example 3.4.1, we note an over-estimation of the cost: we would expect paying only extra n units for evaluating a list spine of length n; instead the difference between the bounds is $3 \times n + 2 \times n \times (n-1) - (\frac{3}{2} \times n \times (n-1)) = n + \frac{1}{2} \times n \times (n-1)$.

This overestimation results from the sharing of the list tail xs between *pairs* and *attach*: the two uses do not account for the repeated evaluation of xs. Note, however, that simply changing the sharing rule to distribute the list spine costs, i.e. sharing $xs{:}\mathsf{T}^0(\mathsf{L}^1(\ldots, B))$ to $xs_1{:}\mathsf{T}^0(\mathsf{L}^0(\ldots, B))$ and $xs_2{:}\mathsf{T}^0(\mathsf{L}^1(\ldots, B))$ would, in general, be unsound because we may discard the variable xs_2 and use only xs_1, thus underestimating the cost.

This is an issue that we believe will not be trivial to address. We have not yet tried to approach it, as we were not aware of it until very recently. We leave it as future work.

Another limitation of our analysis as presented here is the fact that it does not allow resource polymorphic recursion, i.e., recursive calls with different resource annotations; as in the strict setting, we expect that this will cause many programs that are not in tail-recursive form to fail to admit an annotated type [12, 8]. For example, if we consider our definition of *pairs* and change the order in which the arguments are sent to `app'`, the inference of annotations eventually reaches some inconsistency. This problem was addressed by Hoffmann in the strict setting by using a cost-free resource metric that assigns zero costs for each evaluation step and extending the algorithmic type rules with resource polymorphic recursion. We believe that the same approach

could be used in our system.

To summarise this chapter, we have presented a system that can successfully infer polynomial bounds for a lazy functional language and provided a demonstration on a simple but relevant example. We explained how we used main keys from previous systems in order to reach our goal (usage of thunk types and the concept of prepaying for lazy evaluation; additive shift for polynomial potential). We also present some of the limitations of our analysis with a brief reflection on how they could be approached in the future. In the next chapter we will explain how we implemented a prototype of this system.

Chapter 4

Exploratory Implementation

In this chapter we give a brief overview of how we implemented a prototype of our system. We also show how the system operates on two examples. The full implementation can be found in `https://github.com/ohhisara/lazy-potential-analysis`. All code can be seen in Appendix A.

4.1 Overview

Much like our theoretical system, our implementation was built over a previous prototype implementation (see `https://github.com/pbv/lazy-amortised-analysis`) of the system for lazy evaluation with linear bounds [22]. This system receives a program as an input and produces an annotated type for that program (or fails, when the program is not typable by the system). The inference of this annotated type is fully automatic, and works as a type reconstruction algorithm with the following phases:

1. Derive an unnanotated type for the given program using Damas-Milner type inference [5];
2. Annotate the resulting type with fresh annotation variables;
3. Go over the type derivation tree to collect constraints over the annotation variables;
4. Solve the collected constraints using an off-the-shelf LP solver (the GLPK library for Haskell [1]).

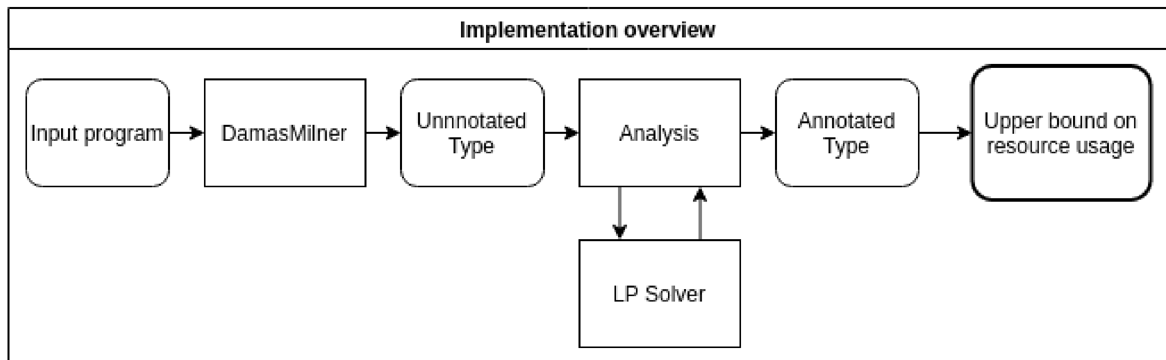


Figure 4.1: Overview of our system flow

Our implementation follows these same steps and an overview of our how the modules of our system interact with each other can be seen in Fig. 4.1. A difference worth mentioning is that our system requires the user to provide a degree for the analysis, so it is not fully automatic. This degree represents the maximum degree for the polynomial function (or the size of the vector of annotations on a list type).

We haven't yet extended the parser from the previous implementation for our syntax, so the input program for our system must be provided in abstract syntax. Fig. 4.2 shows the syntax of our expressions and types.

It goes without saying that the most important module of the system is the Analysis module, it corresponds to the final three phases of the algorithm, where the annotation constraints are collected and solved. One important new concept implemented in our system in this module is the additive shift.

```
1  -- additive shift of a vector of annotations
2  -- second argument is additive shift of first argument
3  additive_shift::[Ann] -> [Ann] -> CLP ()
4  additive_shift (t1:t2:ts) (p1:ps)
5    | length (t1:t2:ts) == length (p1:ps) = do
6      (var t1 + var t2 - var p1) `equalTo` 0
7      additive_shift (t2:ts) ps
8    | otherwise = error "Different sizes for potential"
9  additive_shift (tn:[]) (pn:[]) = do
10     (var tn - var pn) `equalTo` 0
```

The structural rules also play a role in this module, and most heuristic choices for their use are the same as in the previous system. Prepay is used after bound variables are

```haskell
-- language expressions
data Term a
    = Var Ident
    | Nil
    | Lambda Ident (Term a)
    | App (Term a) Ident
    | Pair Ident Ident
    | Let Ident (Term a) (Term a)
    -- ^ letcons is a special use case of let
    | Match (Term a) (Term a) (Term a) (Term a) (Term a)
    -- ^ optinal term is the "otherwise" alternative
    | ConsApp Ident Ident       -- ^ constructor application
    | Const !Integer             -- ^ primitive integers
    | Coerce SrcAnn (Term a)     -- ^ source level annotation
    | (Term a) :@ a              -- ^ type checker annotation

-- type expressions with annotations 'a'
data TyExp a
  = TyVar TVar                   -- ^ variables
  | TyThunk a (TyExp a)          -- ^ thunks
  | TyFun a (TyExp a) (TyExp a)  -- ^ functions
  | TyCon TConst                 -- ^ base types (e.g. Int)
  | TyPair (TyExp a) (TyExp a)
  | TyList a [a] (TyExp a)       -- ^ list types
  deriving (Eq, Show, Functor, Foldable, Traversable)

type TVar = String       -- ^ type variables
type TConst = String     -- ^ basic types

-- Hindley-Milner types (without annotations)
type HMtype = TyExp ()

-- annotated types
type Atype = TyExp Ann

-- annotation variables
newtype Ann = Ann Int deriving (Eq,Ord,Enum, Num)
instance Show Ann where
  showsPrec _ (Ann n) = ('a':) . shows n
```

Figure 4.2: Abstract syntax for terms and types

introduced in the environment (after a lambda expression, let expression or match). Because the rule allows for all the cost to be paid for, or only part of it, the choice of how much to pay is left for the solver. SUBTYPE is used for the result and argument of applications. SHARE is used to split the potential of variables that occur more than once in an environment before applications, let expressions and match.

A special case of share had to be implemented in our system to share the potential of lists. `gather_all_potential` collects the potential of each degree of lists `ts`, as arrays of annotations. `share_potential` restricts the potential of each degree on the first input list to be bigger than the sum of the annotations in the collected arrays sent in the second input.

```
1  share (TyList a1 l1 t) ts = do
2    sequence_ [ do { var ai `geq` var a1
3                   ; share t [ti]
4                   }
5                 | TyList ai _ ti <- ts]
6    share_potential l1 (gather_all_potential ts)
```

As in the previous implementation, an heuristic had to be chosen for the objective function sent to the constraint solver. The choice was to minimize the sum of cost annotations in the result type and judgement.

4.2 Examples

Let us take a look at function *pairs* from the introduction, written in SLFL syntax. To define this function we need to define the body of functions *attach* and *app*′ as well.

```
1  pairs =
2  (Let "attach"
3   (Lambda "n"
4    (Lambda "l"
5     (Match (Term.Var "l")
6      (ConsApp "x" "xs")
7       (Let "p" (Pair "x" "n")
8        (Let "f" (App (App (Term.Var "attach") "n") "xs")
```

```
9           (ConsApp "p" "f")))
10        (Nil) (Nil) )))
11   (Let "app'"
12    (Lambda "l1"
13     (Lambda "l2"
14      (Match (Term.Var "l2")
15       (ConsApp "x1" "xs1")
16        (Let "fn" (App (App (Term.Var "app'") "l1") ("xs1"))
17         (ConsApp "x1" "fn"))
18      (Nil) (Term.Var "l1"))) )
19    (Let "pairs"
20     (Lambda "list"
21      (Match (Term.Var "list")
22       (ConsApp "x2" "xs2" )
23        (Let "f1" (App (Term.Var "pairs") "xs2")
24         (Let "f2" (App (App (Term.Var "attach") "x2") "xs2")
25          (App (App (Term.Var "app'") "f1") "f2")))
26       (Nil) (Nil)))
27      (Term.Var "pairs")))
28  )
```

Example 4.2.1. If we apply our analysis to this function, we get the following annotated type:

```
1  TyFun (-0.0)
2      (TyThunk 0.0 (TyList 0.0 [2.0,3.0] (TyVar "t1")))
3      (TyList (-0.0) [0.0,0.0]
4       (TyThunk (-0.0)
5        (TyThunk (-0.0)
6         (TyPair (TyThunk (-0.0) (TyVar "t1")) (TyThunk (-0.0) (TyVar "t1")))))
7      )
8  ann_in = 3.0, ann_out = 0.0
```

If we look at the type of the input list `(TyThunk 0.0 (TyList 0.0 [2.0,3.0] (TyVar "t1")))` we can see how it is exactly the same type as the one we derived in Example 3.4.1. The annotation 2 takes account of the two uses of let expressions in the function, and the annotation 3 denotes the cost of the auxiliary function *attach*. The annotation `ann_in = 3.0` counts the three let expressions used in the outer part of each function (e.g `let pairs = (...)`) .

Example 4.2.2. Our system also allows the user to manually define restrictions for annotations. Using this, we can replicate the analysis from Example 3.4.2, where the

spine thunk cost annotation is restricted to 1. Applying the analysis with this new restriction we get the following type:

```
TyFun (-0.0)
   (TyThunk 0.0 (TyList 1.0 [3.0,4.0] (TyVar "t1")))
   (TyList (-0.0) [0.0,0.0]
    (TyThunk (-0.0)
        (TyThunk (-0.0)
            (TyPair (TyThunk (-0.0) (TyVar "t1")) (TyThunk (-0.0) (TyVar "t1")))))
    )
ann_in = 3.0, ann_out = 0.0
```

We can see once again how it results exactly in the same types as in Example 3.4.2 (and the same overestimation), where the annotation 3 denotes again the two uses of let expressions in the function, plus the extra cost of each constructor of the list. The annotation 4 represents the cost of the auxiliary function attach. The cost of this function changed as well, because of the extra cost of the spine. Again, the annotation `ann_in = 3.0` counts the three let expressions used in the outer part of each function.

This prototype is still at an early phase of implementation so we have not yet experimented with many examples. However, when looking at these two applications of our implementation, we observe that we get exactly the cost bounds predicted, so we believe the heuristics chosen result in good cost bounds. Further development on the implementation should be useful to experiment on more complex examples.

To summarise this chapter, we have presented an overview of a prototype implementation of the system described in Chapter 3. We clarified the main steps of the implemented algorithm and pointed out relevant concepts introduced in this implementation that were not necessary for the previous system. We finished with a demonstration on two examples.

Chapter 5

Conclusion

This is the closing chapter for our thesis. We go over our main goal and contributions, explaining the overall approach taken to reach that goal. We wrap up with some concluding remarks and directions for future work.

This thesis aimed to extend the previous resource analysis system for lazy evaluation to polynomial bounds. We researched and studied existing literature on automatic amortised resource analysis to understand how the combination of amortisation with a type-based analysis can be used to achieve automatic resource analysis. We then focused on the existing systems that address lazy evaluation with linear bound and strict evaluation with polynomial bounds. We studied their key contributions, trying to understand how we could combine them to reach our goal. The result of our research and study is a type system able to derive polynomial cost bounds on the number of allocation for a simple lazily-evaluated language.

There are some limitations to our system, as we mentioned above. Example 3.4.2 illustrated a cost overestimation caused by duplication of thunk costs inside data structures. We decided to leave investigating mitigations for this issue as future work. Additionally, resource polymorphic recursion is not allowed in our system, we hope that an adaptation of Hoffman's approach to this issue can be used in a straightforward fashion for our system.

Another important issue to mention is the fact that we do not yet have a formal proof of soundness yet. This is, of course, an important step to be taken in the future, and would be the logical next step to take. We believe previous work in [22] could be adapted to our polynomial potential case.

Furthermore, we must say that, although the prototype implementation of our system (Chapter 4) is still at its initial phase, we have successfully applied it to two simple examples and achieved the predicted results for both. We should add that the system as it is now is not expressive enough to be integrated in an industrial compiler like GHC for the analysis of realistic programs. This does not mean, however, that it could not be used as a basis for future development in this sense. It also does not mean that it could not be used to analyse small portions of more complex programs.

Having considered all these points, we believe the outcome of this work was a positive one. We reached our main goal and a secondary one, and left directions for future work. We hope this system can serve as a starting point for further extensions or implementations that can be used to analyse more complex examples.

References

[1] glpk-hs: Comprehensive GLPK linear programming bindings for Haskell. https://hackage.haskell.org/package/glpk-hs, Last accessed on 28.09.2020.

[2] Module lazy of OCaml's standard library. https://caml.inria.fr/pub/docs/manual-ocaml/libref/Lazy.html, Last accessed on 26.09.2020.

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

[4] Henk (Hendrik) Barendregt and E. Barendsen. Introduction to lambda calculus. *Nieuw archief voor wisenkunde*, 4:337–372, 01 1984.

[5] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, 1982.

[6] George B Dantzig and Mukund N Thapa. *Linear programming 1: introduction*. Springer Science & Business Media, 2006.

[7] Michael T Goodrich and John G Kloss. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In *Workshop on Algorithms and Data Structures*, pages 205–216. Springer, 1999.

[8] Jan Hoffmann. *Types with potential: polynomial resource bounds via automatic amortized analysis*. PhD thesis, Ludwig Maximilians University Munich, 2011.

[9] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *ACM SIGPLAN Notices*, volume 46, pages 357–370. ACM, 2011.

[10] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for ocaml. In *ACM SIGPLAN Notices*, volume 52, pages 359–373. ACM, 2017.

[11] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polymorphic recursion and partial big-step operational semantics. In *Asian Symposium on Programming Languages and Systems*, pages 172–187. Springer, 2010.

[12] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In *European Symposium on Programming*, pages 287–306. Springer, 2010.

[13] Jan Hoffmann and Zhong Shao. Type-based amortized resource analysis with integers and arrays. *Journal of Functional Programming*, 25, 2015.

[14] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *ACM SIGPLAN Notices*, volume 38, pages 185–197. ACM, 2003.

[15] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In *European Symposium on Programming*, pages 22–37. Springer, 2006.

[16] Martin Hofmann and Georg Moser. Amortised resource analysis and typed polynomial interpretations. In *Rewriting and Typed Lambda Calculi*, pages 272–286. Springer, 2014.

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

[18] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[19] Steffen Jost. *Automated amortised analysis*. PhD thesis, lmu, 2010.

[20] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *ACM Sigplan Notices*, volume 45, pages 223–236. ACM, 2010.

[21] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. “carbon credits” for resource-bounded computations using amortised

analysis. In *International Symposium on Formal Methods*, pages 354–369. Springer, 2009.

[22] Steffen Jost, Pedro Vasconcelos, Mário Florido, and Kevin Hammond. Type-based cost analysis for lazy functional languages. *Journal of Automated Reasoning*, 59(1):87–120, 2017.

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

[24] Christopher League. Lambda calculi: A guide for computer scientists by chris hankin. *ACM SIGACT News*, 31(1):8–13, 2000.

[25] Anders Møller and Michael I Schwartzbach. Static program analysis. *Notes. Feb*, 2012.

[26] Sara Moreira, Pedro Vasconcelos, and Mário Florido. Resource analysis for lazy evaluation with polynomial potential. In *The 32nd Symposium on Implementation and Application of Functional Languages*. IFL, 2020.

[27] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[28] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

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

[30] Hugo Simões, Pedro Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12*, pages 165–176, 2012.

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

[32] Pedro Vasconcelos, Steffen Jost, Mário Florido, and Kevin Hammond. Type-Based Allocation Analysis for Co-recursion in Lazy Functional Languages. In J. Vitek, editor, *ESOP 2015, European Symposium on Programming*, volume 9032 of *Lecture Notes in Computer Science*, pages 787–811, 2015.

Appendix A

Analysis for a Simple Lazy Functional Language

Implementation of the system described in Chapter 3.

All code presented in this appendix can be found in the GitHub repository `https://github.com/ohhisara/lazy-potential-analysis`. The code of the previous system can be found in `https://github.com/pbv/lazy-amortised-analysis`.

A.1 Abstract Syntax for Terms

```
module Term where

import           Types
import           Data.LinearProgram hiding (Var)
import           Data.Set (Set)
import qualified Data.Set as Set

-- | identifiers
type Ident = String

-- terms with sub-terms with annotations `a'
-- used to keep the type information and
-- avoid the need for "guessing" during constraint collection
data Term a
  = Var Ident
```

```
16    | Nil
17    | Lambda Ident (Term a)
18    | App (Term a) Ident
19    | Pair Ident Ident
20    | Let Ident (Term a) (Term a)
21      -- ^ letcons is a special use case of let
22    | Match (Term a) (Term a) (Term a) (Term a) (Term a)
23    | ConsApp Ident Ident       -- ^ constructor application
24    | Const !Integer             -- ^ primitive integers
25    | Coerce SrcAnn (Term a)     -- ^ source level annotation
26    | (Term a) :@ a              -- ^ type checker annotation
27    deriving (Show, Functor, Foldable, Traversable)
28
29  -- | source annotations
30  type SrcAnn  = (SrcType, [SrcConstr])
31  type SrcType = TyExp String
32  type SrcConstr = Constraint String Int
33
34  -- | a typing judgment for a closed term
35  -- parameterized by annotations 'a'
36  data Typing a
37    = Typing { aterm :: Term (TyExp a)
38             , atype :: TyExp a
39             , ann_in :: a
40             , ann_out :: a
41             }
42    deriving (Functor, Foldable, Traversable, Show)
43
44  -- | collect free variables from a term
45  freevars :: Term a -> Set Ident
46  freevars (Nil)        = Set.empty
47  freevars (Var x)        = Set.singleton x
48  freevars (Lambda x e)   = Set.delete x (freevars e)
49  freevars (App e y)      = Set.insert y (freevars e)
50  freevars (Pair v1 v2) = (Set.singleton v1) `Set.union` (Set.singleton v2 )
51  freevars (ConsApp v1 v2) = (Set.singleton v1) `Set.union` (Set.singleton v2 )
52  freevars (Let x e1 e2)  = Set.delete x (freevars e1 `Set.union` freevars e2)
53  freevars (Match e1 e2 e3 e4 e5) = freevars e1 `Set.union`
54                                   freevars e2 `Set.union`
55                                   freevars e3 `Set.union`
56                                   freevars e4 `Set.union`
57                                   freevars e5
58  freevars (Const _)         = Set.empty
59  freevars (Coerce _ e)      = freevars e
60  freevars (e :@ _)          = freevars e
```

```
-- | rename an identifier
rename :: Ident -> Ident -> Term a -> Term a
rename _ _ e@(Nil) = Nil
rename x y e@(Var v) | v==x       = Var y
                     | otherwise = e
rename x y e@(Lambda x' e')
  | x==x'      = e
  | otherwise = Lambda x' (rename x y e')
rename x y (App e' v) = App (rename x y e') v'
  where v' | v==x       = y
           | otherwise = v
rename x y (ConsApp v1 v2)
  = ConsApp (if x==v1 then y else v1) (if x==v2 then y else v2)
rename x y (Pair v1 v2)
  = Pair (if x==v1 then y else v1) (if x==v2 then y else v2)
rename x y e@(Let x' e1 e2)
  | x'==x      = e
  | otherwise = Let x' (rename x y e1) (rename x y e2)

rename x y (Match e1 e2 e3 e4 e5)
  = Match (rename x y e1) (rename x y e2) (rename x y e3)
          (rename x y e4) (rename x y e5)

rename x y e@(Const n) = e

-- annotations
rename x y (Coerce a e) = Coerce a (rename x y e)
rename x y (e :@ a) = rename x y e :@ a

-- | rename many identifiers
renames :: [Ident] -> [Ident] -> Term a -> Term a
renames (x:xs) (y:ys) e = renames xs ys (rename x y e)
renames [] [] e = e
renames _  _  _ = error "renames: variable lists must have equal length"
```

A.2 Abstract Syntax for Types

```
module Types where

import           Data.Foldable (toList)
```

```
4  import           Data.Map (Map)
5  import qualified Data.Map as Map
6
7  infixr 4 ~>     -- function type constructor
8
9  -- | type expressions with annotations 'a'
10 data TyExp a
11   = TyVar TVar                     -- ^ variables
12   | TyThunk a (TyExp a)            -- ^ thunks
13   | TyFun a (TyExp a) (TyExp a)    -- ^ functions
14   | TyCon TConst                   -- ^ base types (e.g. Int)
15   | TyPair (TyExp a) (TyExp a)
16   | TyList a [a] (TyExp a)         -- ^ list types
17   deriving (Eq, Show, Functor, Foldable, Traversable)
18
19 type TVar = String       -- ^ type variables
20 type TConst = String     -- ^ basic types
21
22 -- | Hindley-Milner types (without annotations)
23 type HMtype = TyExp ()
24
25 -- | annotated types
26 type Atype = TyExp Ann
27
28 -- | annotation variables
29 newtype Ann = Ann Int deriving (Eq,Ord,Enum, Num)
30
31 instance Show Ann where
32   showsPrec _ (Ann n) = ('a':) . shows n
33
34 -- | auxiliary functions to make simple types
35 tycon = TyCon
36 tyvar  = TyVar
37 tyfun = TyFun
38 hmfun = TyFun ()
39 (~>) = hmfun
40 thunk   = TyThunk
41 hmthunk = TyThunk ()
42 hmalt c t = (c, (), t)
43 hmint = TyCon "Int"
44
45 annotations :: Foldable t => t a -> [a]
46 annotations = toList
47
48 -- | collect all type variables
```

```
49 -- generic foldable and for plain type expressions
50 typevars :: Foldable f => f (TyExp a) -> [TVar]
51 typevars = foldMap tyvars
52
53 tyvars :: TyExp a -> [TVar]
54 tyvars (TyCon _) = []
55 tyvars (TyVar x) = [x]
56 tyvars (TyPair t1 t2) = tyvars t1 ++ tyvars t2
57 tyvars (TyThunk _ t) = tyvars t
58 tyvars (TyFun _ t1 t2) = tyvars t1 ++ tyvars t2
59 tyvars (TyList _ _ t) = tyvars t
60
61 -- | type substitutions
62 type Tysubst a = Map TVar (TyExp a)
63 type HMsubst = Tysubst ()
64
65
66 -- | apply a substitution to a type
67 appsubst :: Tysubst a -> TyExp a -> TyExp a
68 appsubst _ t@(TyCon _) = t
69 appsubst s t@(TyVar v) = Map.findWithDefault t v s
70 appsubst s (TyThunk q t') = TyThunk q (appsubst s t')
71 appsubst s (TyFun q t1 t2) = TyFun q (appsubst s t1) (appsubst s t2)
72 appsubst s (TyPair t1 t2) = TyPair(appsubst s t1) (appsubst s t2)
73 appsubst s (TyList q p t) = TyList q p (appsubst s t)
```

A.3 Damas-Milner Type Inference

```
1  module DamasMilner where
2
3  import           Term
4  import           Types
5  import Debug.Trace
6  import qualified Data.Map as Map
7  import           Control.Monad.State
8  import           Control.Monad.Except
9
10 -- type schemes
11 newtype HMscheme = Gen ([TVar], HMtype) deriving (Eq,Show)
12
13 -- inject a type into a scheme
```

```
14 nogen :: HMtype -> HMscheme
15 nogen t = Gen ([], t)
16
17 gen :: [TVar] -> HMtype -> HMscheme
18 gen vs t = Gen (vs, t)
19
20 -- Hindley-Milner context assigning types schemes to variables
21 type HMcontext = [(Ident, HMscheme)]
22
23 -- a monad for Hindler-Milner type inference/checking
24 -- combination of state and failure
25 type Tc = StateT TcState (Either String)
26
27 -- type checking state: fresh var generation and current unifier
28 data TcState = TcState { counter :: Int,  unifier :: HMsubst }
29              deriving (Eq, Show)
30
31 -- generate fresh variables
32 freshvars :: Int -> Tc [TVar]
33 freshvars n = do i <- gets counter
34                  modify $ \s -> s {counter=counter s+n}
35                  return ['t':show n | n<-[i..i+n-1]]
36
37 freshvar :: Tc TVar
38 freshvar = liftM head (freshvars 1)
39
40 -- lookup in context and instantiate type scheme
41 lookupTc :: Ident -> HMcontext -> Tc HMtype
42 lookupTc x ctx
43   = case lookup x ctx of
44     Nothing -> throwError ("unbound variable: " ++ show x)
45     Just (Gen (vs,t)) -> do vs' <- freshvars (length vs)
46                             let s = Map.fromList $ zip vs (map TyVar vs')
47                             return (appsubst s t)
48
49
50 -- assert a unification constraint
51 unify :: HMtype -> HMtype -> Tc ()
52 unify t1 t2 = do u <- gets unifier
53                  case unifyEqs u [(t1, t2)] of
54                    Left err -> throwError err
55                    Right u' -> modify $ \s -> s {unifier=u'}
56
57 -- unification algorithm
58 -- takes the current unifier and a list of term equations
```

```
59  -- result is extended unifier or failure
60  unifyEqs :: HMsubst -> [(HMtype, HMtype)] -> Either String HMsubst
61  unifyEqs s [] = return s
62  unifyEqs s ((t,t'):eqs) = unifyEqs' s (appsubst s t) (appsubst s t') eqs
63
64  -- worker function to unify two types and more equations
65  -- pre-condition: substitution has been applyed to the types
66  -- unifyEqs' s TySelf TySelf eqs
67  --    = unifyEqs s eqs
68  unifyEqs' s t@(TyCon c) t'@(TyCon c') eqs
69
70    | c==c'      = unifyEqs s eqs
71    | otherwise = throwError $
72                  unlines ["type mismatch 1: ||" ++ (show t) ++ "|| "++ (show t')]
73
74  unifyEqs' s (TyVar x) (TyVar y) eqs
75    = case compare x y of    -- fix bindings from higher to lower variables
76      EQ -> unifyEqs s eqs
77      LT -> unifyEqs (extend y (TyVar x) s) eqs
78      GT -> unifyEqs (extend x (TyVar y) s) eqs
79
80  unifyEqs' s v@(TyVar x) t eqs
81    | x `notElem` tyvars t = unifyEqs (extend x t s) eqs
82    | otherwise = throwError $ unlines ["occur check failed:" ++(show v) ++
83                                         (show t)]
84
85  unifyEqs' s t (TyVar x) eqs = unifyEqs' s (TyVar x) t eqs
86
87  unifyEqs' s (TyFun _ t1 t2) (TyFun _ t1' t2') eqs
88    = unifyEqs s ((t1,t1'):(t2,t2'):eqs)
89
90  unifyEqs' s (TyThunk _ t) (TyThunk _ t') eqs
91    = unifyEqs s ((t,t'):eqs)
92
93  unifyEqs' s (TyList _ _ t1) (TyList _ _ t2) eqs
94   = unifyEqs s ((t1,t2):eqs)
95
96  unifyEqs' s (TyPair t1 t2) (TyPair t3 t4) eqs
97   = unifyEqs s ((t1,t3):(t2,t4):eqs)
98  -- distinct type structures
99  unifyEqs' _ t t' _   = throwError $ unlines ["type mismatch 2: ||" ++ (show t) ++ "|| "
100                                               ++ (show t')]
101
102 -- extend a type substitution maintaining idempotency
103 extend :: TVar -> HMtype -> HMsubst -> HMsubst
```

```
104 extend v t s = Map.insert v t $ Map.map (appsubst s') s
105   where s' = Map.singleton v t
106
107 -- Damas-Milner type inference
108 -- takes a context and term;
109 -- computes the Hindley-Milner annotated term
110 -- discards annotations in the original term
111 hm_infer :: HMcontext -> Term () -> Tc (Term HMtype)
112 hm_infer ctx (Var x)
113   = do t <- lookupTc x ctx
114        a <- freshvar
115        unify t (TyThunk () (TyVar a))
116        return (Var x :@ (TyVar a))
117
118 hm_infer _ Nil
119   = do a <- freshvar
120        return (Nil :@ TyList () [()] (TyVar a))
121
122 hm_infer ctx (Lambda x e)
123   = do a <- freshvar
124        let ctx' = (x, nogen (hmthunk (tyvar a))):ctx
125        (e' :@ t) <- hm_infer ctx' e
126        return (Lambda x e' :@ (hmthunk (tyvar a) ~> t))
127
128 hm_infer ctx (App e y)
129   = do t1 <- lookupTc y ctx
130        (e' :@ te) <- hm_infer ctx e
131        b <- freshvar
132        unify te (t1 ~> tyvar b)
133        return (App (e':@te) y :@ tyvar b)
134
135 hm_infer ctx (Pair x1 x2)
136   = do t1 <- lookupTc x1 ctx
137        t2 <- lookupTc x2 ctx
138        return (Pair x1 x2 :@ (TyPair t1 t2))
139
140 hm_infer ctx (ConsApp x1 x2)
141   = do b <- lookupTc x1 ctx
142        l <- lookupTc x2 ctx
143        unify l (TyThunk () (TyList () [()] b))
144        return (ConsApp x1 x2 :@ (TyList () [()] b))
145
146 hm_infer ctx (Let x (ConsApp x1 x2) e2)
147   = do a <- freshvar
148        b <- freshvar
```

```
149         unify (TyVar a) (TyList () [()] (TyVar b))
150         let ctx' = (x, nogen (TyThunk () (TyVar a))):ctx
151         (e1':@t1) <- hm_infer ctx' (ConsApp x1 x2)
152         unify (TyVar a) t1
153         (e2':@t2) <- hm_infer ctx' e2
154         return (Let x (e1':@t1) e2' :@t2)
155
156 hm_infer ctx (Let x e1 e2)
157   = do a <- freshvar
158         let ctx' = (x, nogen (TyThunk () (TyVar a))):ctx
159         (e1' :@ t1) <- hm_infer ctx' e1
160         unify (tyvar a) t1
161         (e2' :@ t2) <- hm_infer ctx' e2
162         return (Let x (e1':@t1) e2' :@ t2)
163
164 hm_infer ctx (Match e0 (ConsApp x1 x2) e1 (Nil) e2)
165   = do (e0' :@ t0) <- hm_infer ctx e0
166         a <- freshvar
167         let ctx' = (x1, nogen (hmthunk (TyVar a))):ctx
168         let ctx'' = (x2, nogen (TyThunk () (TyList () [()] (TyVar a)))):ctx'
169         (e1' :@ t1) <- hm_infer ctx'' e1
170         (e2' :@ t2) <- hm_infer ctx e2
171         unify t1 t2
172         return (Match (e0':@t0) (ConsApp x1 x2) (e1':@t1) Nil (e2':@t2) :@ t1)
173
174 -- constants
175 hm_infer _ (Const n)
176   = return (Const n :@ hmint)
177
178
179 hm_infer ctx (Coerce a@(t',_) e)
180   = do (e' :@ t) <- hm_infer ctx e
181         -- ensure the annotated type has the same HM structure
182         let t'' = fmap (\_ -> ()) t'
183         unify t t''
184         return (Coerce a e' :@ t)
185
186
187 hm_infer _ _ = error "hm_infer: invalid argument"
188
189 -- perform HM type inference and annotate the term with types
190 hm_inference :: Term () -> Either String (Term HMtype)
191 hm_inference e
192   = do (e',tc) <- runStateT (hm_infer [] e) tc0
193         return (let s = unifier tc
```

```
194                 in fmap (appsubst s) e')
195   where  -- initial state for the type checker
196     tc0 = TcState { counter=0, unifier = Map.empty }
```

A.4 Resource Analysis

```
1  module Analysis where
2
3  import           Prelude hiding (Num(..))
4  import           Algebra.Classes hiding (zero)
5
6  import           Term
7  import           Types
8  import           Control.Monad.State
9  import           Control.Monad.Reader
10 import           Data.LinearProgram hiding (Var,zero)
11 import           Data.LinearProgram.GLPK.Solver
12 import           Control.Monad.LPMonad hiding (Var)
13 import           Data.Map (Map)
14 import qualified Data.Map as Map
15 import qualified Data.Set as Set
16 import           Data.List (transpose, partition, nubBy)
17 import           Data.Char (isSymbol)
18 import           Options
19 import           Cost (CostModel(..))
20 import           Debug.Trace
21 import           DamasMilner
22
23 -- | Degree of polynomial function
24 type Degree = Int
25
26 -- | typing contexts for annotated types
27 type Context a = [(Ident,TyExp a)]
28
29 -- | context for the lazy amortized analysis
30 type Acontext = Context Ann
31
32 -- | a monad for constructing linear programs
33 type CLP = LPT Ann Int (StateT Ann (Reader Options))
34
35 -- | fixed zero annotation variable
```

```
36 zero_ann :: Ann
37 zero_ann = Ann 0
38
39 zero :: LinFunc Ann Int
40 zero = linCombination []
41
42 -- singleton annotation variables
43 var :: Ann -> LinFunc Ann Int
44 var x = linCombination [(1,x)]
45
46 -- sum a list of annotations
47 vars :: [Ann] -> LinFunc Ann Int
48 vars xs = linCombination $ zip (repeat 1) xs
49
50 -- generate a fresh annotation variable
51 fresh_ann :: CLP Ann
52 fresh_ann = do a <- lift (do {modify succ; get})
53                varGeq a 0   -- impose non-negativity
54                return a
55
56 -- generate list of fresh annotation variables
57 fresh_anns:: Degree -> CLP [Ann]
58 fresh_anns 0 = return []
59 fresh_anns k = do
60   a <- fresh_ann
61   anns <- fresh_anns (k-1)
62   return (a:anns)
63
64 -- additive shift of a vector of annotations
65 -- second argument is additive shift of first argument
66 additive_shift::[Ann] -> [Ann] -> CLP ()
67 additive_shift (t1:t2:ts) (p1:ps)
68   | length (t1:t2:ts) == length (p1:ps) = do
69     (var t1 + var t2 - var p1) `equalTo` 0
70     additive_shift (t2:ts) ps
71   | otherwise = error "Different lengths"
72 additive_shift (tn:[]) (pn:[]) = do
73     (var tn - var pn) `equalTo` 0
74
75 -- decorate a type with fresh anotation variables
76 decorate_type :: Degree -> TyExp a -> CLP Atype
77 decorate_type k (TyVar x) = return (TyVar x)
78 decorate_type k (TyThunk _ t)
79   = do q <- fresh_ann
80        t'<- decorate_type k t
```

```
81         return (TyThunk q t')
82
83  decorate_type k (TyFun _  t1 t2)
84    = do p <- fresh_ann
85         t1' <- decorate_type k t1
86         t2' <- decorate_type k t2
87         return (TyFun p t1' t2')
88
89  decorate_type k (TyPair t1 t2)
90    = do p <- fresh_ann
91         t1' <- decorate_type k t1
92         t2' <- decorate_type k t2
93         return (TyPair t1' t2')
94
95  decorate_type k (TyCon b) = return (TyCon b)
96
97  decorate_type k (TyList _ _ t)
98        = do an <- fresh_ann
99             p <- fresh_anns k
100            t1 <- decorate_type k t
101            return (TyList an p t1)
102
103
104 -- | decorate a term with annotation variables
105 decorate_term :: Degree -> Term HMtype -> CLP (Term Atype)
106 decorate_term k Nil = return Nil
107 decorate_term k (Var x) = return (Var x)
108
109 decorate_term k (Lambda x e)
110   = do e' <- decorate_term k e
111        return (Lambda x e')
112
113 decorate_term k (App e y)
114   = do e' <- decorate_term k e
115        return (App e' y)
116
117 decorate_term k (ConsApp x1 x2)
118   = return (ConsApp x1 x2)
119
120 decorate_term k (Pair x1 x2)
121   = return (Pair x1 x2)
122
123 decorate_term k (Let x e1 e2)
124   = do e1'<- decorate_term k e1
125        e2'<- decorate_term k e2
```

```
126         return (Let x e1' e2')
127
128 decorate_term k (Match e0 e1 e2 e3 e4)
129   = do e0' <- decorate_term k e0
130         e1' <- decorate_term k e1
131         e2' <- decorate_term k e2
132         e3' <- decorate_term k e3
133         e4' <- decorate_term k e4
134         return (Match e0' e1' e2' e3' e4')
135
136
137 -- | primitive operations
138 decorate_term k (Const n)        = return (Const n)
139
140 decorate_term k (Coerce a e)
141   = do e' <- decorate_term k e
142         return (Coerce a e')
143
144 -- | annotations
145 decorate_term k (e :@ t)
146   = do e' <- decorate_term k e
147         t' <- decorate_type k t
148         return (e' :@ t')
149
150
151 decorate_term k e = error ("non exaustive " ++ (show e))
152
153
154 -- | sharing relation between types
155 -- pre-condition: types have the same Hindley-Milner structure
156 share :: Atype -> [Atype] -> CLP ()
157
158 share _ [] = return ()
159
160 -- thunks
161 share (TyThunk q t0) ts
162   = do sequence_ [ do { var qi `geq` var q
163                       }
164                  | TyThunk qi ti <- ts]
165         share t0 [ ti | TyThunk qi ti <- ts]
166
167 -- function types
168 share (TyFun q a b) ts
169   =  sequence_ [ do { var qi `geq` var q
170                     ; share ai [a]
```

```
171                    ; share b [bi]
172                    }
173               | TyFun qi ai bi <- ts]
174
175 -- constants
176 share (TyCon b) ts = return ()
177
178 -- variables
179 share (TyVar x) ts = return ()
180
181 -- pairs
182 share (TyPair t1 t2) ts = sequence_ [ do { share t1 [t1']
183                    ; share t2 [t2']
184                   }
185                | TyPair t1' t2' <- ts]
186
187 -- lists
188 share (TyList a1 l1 t) ts = do
189   sequence_ [ do { var ai `geq` var a1
190                    ; share t [ti]
191                   }
192                | TyList ai _ ti <- ts]
193   share_potential l1 (gather_all_potential ts) --special case to share potential
194
195 -- to collect potential from list of lists
196 gather_all_potential::[Atype] -> [[Ann]]
197 gather_all_potential [] = []
198 gather_all_potential l = (gather_potential l):(gather_all_potential (shift_right l))
199
200 -- gather potential os same degree
201 gather_potential::[Atype] -> [Ann]
202 gather_potential []= []
203 gather_potential ((TyList ai [] ti):ts )= []
204 gather_potential ((TyList ai (l1:xsi) ti):ts )= l1:(gather_potential ts)
205
206 -- share potential of one list to others
207 share_potential::[Ann] -> [[Ann]] -> CLP ()
208 share_potential _ [] = return ()
209 share_potential [] _ = return ()
210 share_potential (a:as) (l:ls) = do
211   (var a  `geq` vars l)
212   share_potential as ls
213
214 shift_right::[Atype] -> [Atype]
215 shift_right [] = []
```

```
216 shift_right ((TyList ai [] ti):xs) = []
217 shift_right ((TyList ai (l1:xsi) ti):xs) = (TyList ai xsi ti):(shift_right xs)
218
219 -- | subtyping is a special case of sharing
220 subtype, equaltype :: Atype -> Atype -> CLP ()
221 t1 `subtype` t2 = share t1 [t2]
222
223 -- NB: the following is not needed
224 t1 `equaltype` t2 = do t1 `subtype` t2; t2 `subtype` t1
225
226 -- | sharing a context against itself
227 share_self :: Acontext -> CLP ()
228 share_self ctx = sequence_ [share t [t,t] | (x,t)<-ctx]
229
230
231 -- | split a context for typing a subexpression
232 split_context :: Degree -> Acontext -> CLP (Acontext, Acontext)
233 split_context k ctx
234   = let newctx = sequence [do {t'<-decorate_type k t; return (x,t')}
235                           | (x,t)<-ctx]
236     in do ctx1 <- newctx
237           ctx2 <- newctx
238           sequence_ [ share t [t1,t2] |
239                       (t,t1,t2)<-zip3 (map snd ctx) (map snd ctx1) (map snd ctx2)]
240           return (ctx1, ctx2)
241
242
243 -- | trim context to vars with free occurences in a term
244 trim_context :: Term b -> Context a -> Context a
245 trim_context e
246   = filter (\(x,_) -> x`Set.member`vars) . nubBy (\(x,_) (y,_) -> x==y)
247   where vars = freevars e
248
249
250 -- relax cost annotations
251 -- if \Gamma |-p0/p0'- e : C then  \Gamma |-p/p'- e : C
252 relaxcost :: (LinFunc Ann Int, LinFunc Ann Int) ->
253              (LinFunc Ann Int, LinFunc Ann Int) -> CLP ()
254 (p,p') `relaxcost` (p0,p0') = do {p `geq` p0;  (p - p0) `geq` (p' - p0')}
255
256
257 -- lookup a name in a context
258 lookupId :: Ident -> Context a -> TyExp a
259 lookupId x ctx
260   = case lookup x ctx of
```

```
261      Nothing -> error ("unbound identifier: "++show x)
262      Just t -> t
263
264  -- as above but enforces sharing and returns remaining context
265  lookupShare :: Degree -> Ident -> Acontext -> CLP (Atype,Acontext)
266  lookupShare k x ctx
267    = case span (\(x',_) -> x'/=x) ctx of
268      (_, []) -> error ("unbound identifier: "++show x)
269      (ctx', (_,t):ctx'') -> do t1 <-decorate_type k t
270                                t2 <- decorate_type k t
271                                share t [t1,t2]
272                                return (t1, ctx' ++ (x,t2):ctx'')
273
274
275  -- get a cost model constant
276  askC :: (CostModel -> Int) -> CLP Int
277  askC k = fmap k $ asks optCostModel
278
279  --  Amortised Analysis
280  -- collects linear constraints over annotations
281  aa_infer :: Degree -> Acontext -> Term Atype -> Atype -> Ann -> Ann -> CLP ()
282  aa_infer k ctx (Nil) (TyList _ _ _) p p'
283    = var p `geq` var p'
284
285  -- Var rule
286  aa_infer k ctx (Var x) t p p'
287    = let TyThunk q t' = lookupId x ctx
288      in do  ((var p - var p') - var q) `equalTo` 0
289             t' `subtype` t                          -- allow subtyping
290
291  -- Abs rule
292  -- allow prepaying for the argument
293  aa_infer k ctx (Lambda x e) (TyFun q t t') p p'
294    = do share_self (trim_context e ctx)
295         var p `geq` var p' -- allow relaxing
296         aa_infer_prepay k [(x,t)] ctx e t' q zero_ann
297
298
299  -- App rule
300  aa_infer k ctx (App (e :@ te) y) t0 p p'
301    | TyFun q t' t <- te
302    =  do (ty, ctx') <- lookupShare k y ctx
303          pe <- fresh_ann
304          let potential = look_for_potential t
305          new_type <- decorate_type k ty
```

```
306         t1 <- out_potential k new_type potential
307         t' `subtype` t1
308         aa_infer k ctx' e (TyFun q new_type t) pe p'
309         -- allow subtyping the argument and result
310         ty `subtype` t'
311         t `subtype` t0
312         ((var p - var pe) - var q) `equalTo` 0
313
314
315 -- Letcons rule
316 aa_infer k ctx (Let x ((ConsApp c ys) :@ (TyList an pot tL)) e2) tC p p'
317       = do pe <- fresh_ann
318            tL'<- decorate_type k (TyList an pot tL)
319            share (TyList an pot tL) [(TyList an pot tL), tL']
320            (ctx1,ctx2) <- split_context k ctx
321            ((var p - var pe ) - var (head pot)) `equalTo` 1
322            aa_infer k ((x,TyThunk zero_ann tL'):ctx1)  (ConsApp c ys)
323                     (TyList an pot tL) zero_ann zero_ann
324            aa_infer k ((x,TyThunk zero_ann (TyList an pot tL)):ctx2) e2 tC pe p'
325
326 -- Let rule
327 aa_infer k ctx (Let x (e1 :@ tA) e2) tC p p'
328   = do
329        tA' <- decorate_type k tA
330        share tA [tA, tA']
331        q <- fresh_ann
332        pe <- fresh_ann
333        (var p - var pe) `equalTo` 1
334        (ctx1, ctx2) <- split_context k ctx
335        aa_infer k ((x,TyThunk zero_ann tA'):ctx1) e1 tA q zero_ann
336        aa_infer_prepay k [(x,TyThunk q tA)] ctx2 e2 tC pe p'
337
338 aa_infer k ctx (Pair x1 x2) (TyPair t1 t2) z z'
339  = do (var z `geq` var z')
340
341 -- Cons rule
342 aa_infer k ctx (ConsApp x1 x2) (TyList a p t) z z'
343    = do
344      var z `geq` var z'
345      let tx1 = lookupId x1 ctx
346      let tx2 = lookupId x2 ctx
347      case tx1 of
348        TyThunk q ttype -> do
349          case tx2 of
350             TyThunk ann (TyList an po ty) -> do
```

```
351                 additive_shift p po
352                 (var ann - var an) `equalTo` 0
353               _ -> error "Not a list"
354          _ -> error "Not a thunk"
355
356
357 aa_infer k ctx (ConsApp x1 x2) t z z' = error ("Wrong type for ConsApp: "++ show t)
358
359 -- Match rule
360 aa_infer k ctx (Match (e0 :@ (TyList a p t0)) (ConsApp x1 x2) (e2 :@ t2) Nil  (e4 :@ t4))
361                                                                                     t z z''
362   = do new_ann <- fresh_ann
363        (ctx1,ctx2) <- split_context k ctx
364        p1 <- fresh_anns k
365        additive_shift p p1
366        let ctx' = [(x2, TyThunk a (TyList a p1 t0)),(x1, TyThunk new_ann t0)]
367        pt <- fresh_ann
368        z' <- fresh_ann
369        (var z' + var (head p)) `geq` var pt
370        aa_infer k ctx1 e0 (TyList a p t0) z z'
371        aa_infer_prepay k ctx' ctx2 e2 t2 pt z''
372        aa_infer k ctx2 e4 t4 z' z''
373
374
375 -- Constants
376 aa_infer k ctx (Const n) t p p'    -- t must be hmint
377   = var p `geq` var p' -- allow relaxing
378
379 -- user annotations and constraints
380 -- checking that t and t' match is done during Damas-Milner inference
381 aa_infer k ctx (Coerce (t,cs) e) t' p p'
382   = let s = Map.fromList $ zip (annotations t) (annotations t')
383          ren x = Map.findWithDefault undefined x s
384     in do sequence_ [ constrain lf' bds
385                     | Constr _ lf bds <- cs,
386                       let lf' = Map.mapKeys ren lf
387                     ]
388           aa_infer k ctx e t' p p'
389
390
391 aa_infer k ctx e t p p' = error ("aa_infer: undefined for " ++ show e)
392
393 -- create new type with potential >= the given potential
394 out_potential::Degree -> Atype -> [Ann] -> CLP Atype
395 out_potential k (TyList a1 p1 t1) p2 = do
```

```
396     new_potential <- fresh_anns k
397     share_potential_simple new_potential p1 p2
398     new_type <- out_potential k t1 p2
399     return (TyList a1 new_potential new_type)
400   out_potential k (TyFun a t1 t2) p2 = do
401     new_type1 <- out_potential k t1 p2
402     new_type2 <- out_potential k t2 p2
403     return (TyFun a new_type1 new_type2)
404   out_potential k (TyPair t1 t2) p2 = do
405     new_type1 <- out_potential k t1 p2
406     new_type2 <- out_potential k t2 p2
407     return (TyPair new_type1 new_type2)
408   out_potential k (TyThunk a t1) p2 = do
409     new_type1 <- out_potential k t1 p2
410     return (TyThunk a new_type1)
411   out_potential k (TyVar a) p2 = do
412     return (TyVar a)
413   out_potential k (TyCon a) p2 = do
414     return (TyCon a)
415
416   -- potential os first argument is shared with the other
417   share_potential_simple::[Ann] -> [Ann] -> [Ann]-> CLP ()
418   share_potential_simple _ [] _ = return ()
419   share_potential_simple [] _ _= return ()
420   share_potential_simple (a:as) (l:ls) (q:qs) = do
421     var a `geq` (var l + var q)
422     share_potential_simple as ls qs
423
424   -- look for potential in a type
425   look_for_potential:: Atype -> [Ann]
426   look_for_potential (TyList _ p t1)= p
427   look_for_potential (TyThunk _ ti) = look_for_potential ti
428   look_for_potential (TyFun _ t1 t2) = (look_for_potential t2)
429   look_for_potential (TyVar _ ) = []
430   look_for_potential (TyCon _ ) = []
431   look_for_potential (TyPair t1 t2 ) = (look_for_potential t1)++(look_for_potential t2)
432
433   --
434   -- allow the prepay for all variables in the first context
435   -- followed by type inference
436   aa_infer_prepay :: Degree -> Acontext -> Acontext ->
437                      Term Atype -> Atype -> Ann -> Ann -> CLP ()
438   aa_infer_prepay k [] ctx e t' p p' = aa_infer k ctx e t' p p'
439   aa_infer_prepay k ((x,TyThunk q t) : ctx1) ctx2 e t' p p'
440     = do q0 <- fresh_ann
```

```
441         q1 <- fresh_ann
442         p0 <- fresh_ann
443         var q `equal` (var q0 + var q1)
444         var p `equal` (var p0 + var q1)
445         aa_infer_prepay k ctx1 ((x,TyThunk q0 t):ctx2) e t' p0 p'
446 aa_infer_prepay k ctx _ e t' p p'
447   = error ("aa_infer_prepay: invalid context\n " ++ show ctx)
448
449 -- lookup and share many identifiers in sequence
450 lookupMany :: Degree -> [Ident] -> Acontext -> CLP ([Atype], Acontext)
451 lookupMany k [] ctx = return ([],ctx)
452 lookupMany k (x:xs) ctx
453   = do (t, ctx') <- lookupShare k x ctx
454        (ts, ctx'')<- lookupMany k xs ctx'
455        return (t:ts, ctx'')
456
457 -- leave only type annotations for let-bindings
458 let_annotations :: Term a -> Term a
459 let_annotations (Let x (e1 :@ t1) e2)
460   = Let x (let_annotations e1 :@ t1) (let_annotations e2)
461 let_annotations (Let x e1 e2)
462   = Let x (let_annotations e1) (let_annotations e2)
463 let_annotations (Lambda x e)
464   = Lambda x (let_annotations e)
465 let_annotations (Match e0 e1 e2 e3 e4)
466   = Match (let_annotations e0) (let_annotations e1) (let_annotations e2)
467           (let_annotations e3) (let_annotations e4)
468 let_annotations (App e y)
469   = App (let_annotations e) y
470 let_annotations (Var x)           = Var x
471 let_annotations (ConsApp x1 x2)    = ConsApp x1 x2
472 let_annotations (Pair x1 x2)    = Pair x1 x2
473 let_annotations (Const n)         = Const n
474 let_annotations Nil        = Nil
475 let_annotations (Coerce a e)      = Coerce a (let_annotations e)
476 let_annotations (e :@ a)          = let_annotations e
477
478
479 -- toplevel inference
480 -- generates a typing \Gamma |-p/p'- e : C
481 -- and a linear program for constraints over annotations
482 -- set p'=0 and solves to minimize p (i.e. the whnf cost of the expression)
483 aa_inference :: Degree -> Options -> Term HMtype -> HMtype -> (Typing Ann, LP Ann Int)
484 aa_inference degree opts e t
485   = flip runReader opts $
```

```
486     flip evalStateT (Ann 1) $
487     runLPT $
488      do { varEq zero_ann 0
489         ; p <- fresh_ann
490         ; e'<- decorate_term degree e
491         ; t'<- decorate_type degree t
492         ; setDirection Min
493         ; setObjective  (vars (p:annotations t'))
494         ; aa_infer degree [] e' t' p zero_ann
495         ; return Typing {aterm = let_annotations e',
496                          atype = t',
497                          ann_in  = p,
498                          ann_out = zero_ann}
499         }
500
501 -- solve linear constraints and instantiate annotations
502 aa_solve :: (Typing Ann, LP Ann Int) -> IO (Typing Double)
503 aa_solve (typing, lp)
504   = do print typing
505        answer <- glpSolveVars simplexDefaults{msgLev=MsgOff} lp
506        case answer of
507          (Success, Just (obj, vars)) ->
508            let subst a = Map.findWithDefault 0 a vars
509            in return (fmap subst typing)
510          (other, _) -> error ("LP solver failed: " ++ show other)
```