Resource Analysis for Lazy Evaluation with Polynomial Potential

Sara Moreira
up201404984@fc.up.pt
Faculdade de Ciências,
Universidade do Porto
Porto, Portugal

Pedro Vasconcelos
pbv@dcc.fc.up.pt
LIACC, Faculdade de Ciências,
Universidade do Porto
Porto, Portugal

Mário Florido
amf@dcc.fc.up.pt
LIACC, Faculdade de Ciências,
Universidade do Porto
Porto, Portugal

ABSTRACT

Space and time requirements of lazy functional programs are hard to predict for both programmers and compilers. Previous work in compile-time amortised analyses for lazy functional languages by Simões, Jost *et.al.* was limited to bounds that are *linear* on the sizes of inputs. This paper presents an extension of these analyses with the method of *polynomial potential* (due to Hofmann and Hoffmann), allowing the system to derive univariate polynomial resource bounds. We present the analysis as a type system for tracking allocations in a simple functional lazy functional language with lists and pairs, an operational semantics (serving as cost model), and worked examples of deriving resource bounds. We also highlight some limitations and conclude with further research directions.

CCS CONCEPTS

• **Theory of computation → Program analysis**; *Type theory*; • **Software and its engineering → Functional languages**.

KEYWORDS

Resource analysis, Amortised analysis, Type-based analysis, Lazy evaluation

ACM Reference Format:
Sara Moreira, Pedro Vasconcelos, and Mário Florido. 2020. Resource Analysis for Lazy Evaluation with Polynomial Potential. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL '20), September 2–4, 2020, Canterbury, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3462172.3462196

1 INTRODUCTION

Lazy evaluation offers known advantages in terms of modularity and higher abstraction [8]. However, the operational properties of lazy languages (such as time and space behaviour) are harder to predict than those of strict ones; this unpredictability remains "a thorn in the side of lazy evaluation" [12].

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 data and codata constructors [10, 16]. While these systems were important contributions, they were limited to linear resource bounds — meaning that any programs exhibiting polynomial or exponential resource use on input size would not admit a type. In this paper we present an extension to derive univariate polynomial bounds[1] using the polynomial potential of Hofmann and Hoffmann [4, 5]. As a motivating example, consider the two functions *attach* and *pairs* (adapted to Haskell from [4]):

```
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 elements of the given list; this uses an auxiliary definition *attach* that pairs a single element to every element of the argument list.

It is straightforward to see that fully evaluating *attach y xs* requires space that is linear on the length n of the input list xs. In fact, using the system in [10] we can automatically derive a precise bound for *allocations* as an annotated type (simplified for presentation):

$$A \to \mathsf{L}(3, B) \xrightarrow{1} \mathsf{L}(0, A \times B)$$

Each element of the input list is annotated with *potential* 3 and the function arrow is annotated with cost 1; this means that *attach* requires $3n + 1$ allocations. Function *pairs*, however, requires space and time that is quadratic on the length its input. Hence, it does not admit a type derivation in the mentioned system.

This paper extends type-based amortised analysis of lazy languages to *polynomial* resource bounds by following the approach done for the strict setting by Hoffman [1, 5]. The analysis is presented for a small lazy functional language with higher-order functions, pairs, lists and recursion. Finally, we give examples of the application of our analysis to programs exhibiting polynomial resource behaviour.

We have also developed a prototype implementation of this system and the code can be found in the Github repository:

`https://github.com/ohhisara/lazy-potential-analysis`

An example output (simplified for presentation) for *pairs* of the implementation is:

```
L@(2.0,3.0)(A) -> L((A,A))
```

This corresponds to a quadratic cost bound of $2 \times n + 3 \times \binom{n}{2} = 2 \times n + \frac{3}{2} \times n \times (n-1)$ expressed as a function of the input list

[1] E.g., allowing a bound $n^2 + m^2$ but not $n \times m$ where n, m are input sizes.

IFL '20, September 2–4, 2020, Canterbury, United Kingdom

ACM ISBN 978-1-4503-8963-1/20/09…$15.00
https://doi.org/10.1145/3462172.3462196

length n. Details about the derivation of this cost equation will be further explained in Section 5.

The rest of the paper is organised as follows. Section 2 surveys relevant background and related work about amortised analysis. Section 3 presents the language and its annotated operational semantics. Section 4 presents the main contribution of this paper: a type and system for resource analysis of lazy evaluation with polynomial bounds. In Section 5, we show several worked examples of the analysis. In section 6 we discuss cost overestimation and point out some solutions. Finally, we conclude and present some future work.

2 BACKGROUND AND RELATED WORK

2.1 Type-based Analysis

Type-based analysis [13] is an approach to static analysis that attaches information to types. One advantage of this approach is the fact that types are well-suited to interface information between software components, thus enabling modular analyses. Types also facilitate communication with programmers by extending an already-known notation. Finally, type theory provides a framework for formulating and proving correctness and for deriving algorithmic checking or inference methods.

2.2 Classic Amortisation

Amortised analysis is a method due to Tarjan for analysing the complexity of a sequence of operations [14, 17]. Rather than reason about the worst-case cost of individual operations, amortised analysis is concerned with the worst-case cost of a *sequence* of operations. The advantage of amortisation is that some operations can be more expensive than others; distributing the cost of expensive operations over the cheaper ones can simplify the analysis and produce better bounds than analysing worst-case for individual operations.

To obtain an amortised analysis it suffices to define an *amortised cost* for operations such that

$$\sum_{i=1}^{n} a_i \geq \sum_{i=1}^{n} t_i$$

i.e.for each sequence of operations the total amortised costs is an upper-bound on the total actual costs (where a_i and t_i are, respectively, the amortised costs and the actual costs of each operation). As a consequence, for each intermediate step of the sequence, the accumulated amortised cost is an upper bound on the accumulated actual cost. This allows the existence of operations with an actual cost that exceeds their amortised cost (*expensive operations*). Conversely, *cheap operations* have a lower actual cost than their amortised cost. Expensive operations can only occur when the difference between the accumulated amortised cost and the accumulated actual cost (*accumulated savings*) is enough to cover the extra cost.

There are three different methods for amortised analysis: the *aggregate method* (using the above relation directly), the *accounting method* (using credits and debits) and the *potential method*. The choice of method depends on how convenient each is to the situation. The type-based analyses that we extend use the potential method which we briefly review.

Potential method. This method defines a function Φ that maps each state of the data structure d_i to a real number (the *potential* 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 amortised cost of an operation is then defined as its actual cost t_i, plus the change in potential between d_{i-1} and d_i:

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

From this definition we get:

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

The sequence of potential function values forms a telescoping series and thus all terms except the initial and final values cancel in pairs. Because $\Phi(d_0) = 0$ and $\Phi(d_j) \geq 0$ then $\sum a_i \geq \sum t_i$. Note also that the above reasoning requires that *potential is used only once*, i.e.$\Phi(d_{i-1})$ cannot be re-used after state i. This will impact the structural proprieties of type systems for automatic amortisation (cf.Section 4.2).

2.3 Automatic Amortisation

In 2003, Hofmann and Jost [6] 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 automatic amortised resource analysis (AARA).

Further work has since then been done using this approach, which is, more specifically, based on the potential method of amortised analysis. The main idea behind this method is the association of *potential* to data structures. This potential is assigned using type annotations which express coefficients of the potential function. Because the inference of suitable annotations can be reduced to a linear programming problem, it is possible to automatically compute the type annotations.

Subsequent work by the same authors applied automatic amortisation to predict heap space for a small Java-like language with explicit deallocations [7]. The data is assigned a potential related to its input and layout, and the allocations are then paid 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. Jost and others also extended automatic amortised analysis to higher-order and polymorphic programs [9].

In 2010, Hoffmann and Hofmann extended automatic amortisation from *linear* to (univariate) *polynomial* bounds [5]. The key idea is the extension of potential annotations to vectors representing polynomial coefficients of polynomials in special basis. The type rules for the system with polynomial potential still generate only linear constraints on annotations, meaning that the polynomial analysis can still be implemented using an ordinary LP solver.

Hoffmann and others also extended the polynomial potential method from univariate to multivariate [2], e.g.capable of expressing bounds such as $n \times m$ where n and m are sizes of structures). This work was later developed into *Resource Aware ML* [3], a static analysis tool that derives polynomial resource bounds for a substantial subset of the OCaml language, including user-defined inductive types, pattern-matching and recursion and higher-order functions.

2.4 Lazy evaluation

While all the previous analyses considered eager languages, Simões *et.al.* extended automatic amortisation to a lazy language [16]. The key contribution is the representation of the delayed costs of thunks as type construtors with annotations and a structural rule that allows pre-paying costs in advance (thus preventing duplication). As with the previous analyses, the proof system generates linear constraints on annotations — thus allowing resource analysis using an off-the-self LP solver, but was limited to linear bounds. Following work address specific issues of co-recursive definitions [18] and to a parametric cost model [10]. In this section, we briefly explain the approach, focusing mainly on the key contributions that we build upon for our system.

The main contributions of this system deal with the particular 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.

$$\Gamma \left|\frac{z}{z'}\right. e : C$$

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. Because this pre-payment happens before the actual use of the variable, it is possible to prevent the same cost to be accounted for multiple times when there are multiple uses of the same variable, "simulating" the memoization of a call-by-need evaluation. This will become more clear later in this paper.

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

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 work 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. Having this system as a basis, we studied Hoffmann's contributions in [5] to understand how we could extend it to polynomial bounds. This will be explained in more detail in the next section.

2.5 Polynomial potential

In this section, we briefly explain Hoffman's approach to (univariate) polynomial potential [5]. This paper presents a technique that allows amortised analysis to obtain polynomial resource bounds but still requires only linear constraints on annotations as side-conditions. This is a important because is allows implementations to use of efficient LP solvers (as in the linear case).

The key idea is to annotate types of data structures with a vector $\vec{p} = (p_1, \ldots, p_k)$ of annotations where k represents the maximum degree of polynomial bounds to be derived, e.g.$k = 2$ corresponds to quadratic bounds; the case $k = 1$ subsumes the system with linear bounds.

A list of type $L(\vec{p}, A)$, where $\vec{p} = (p_1, \ldots, p_k)$, assigns potential p_1 to every element of the list, p_2 to every element of every suffix of the list, p_3 to every suffix of the suffixes of the list, and so on. This means that the polynomial potential is expressed using the binomial basis i.e., $\sum_{i=1}^{k} p_i \binom{n}{i}$ where n is the length of the list. For example, a type $L((3,2),A) \xrightarrow{1} B$ corresponds to a polynomial resource bound

$$3\binom{n}{1} + 2\binom{n}{2} + 1 = 3n + 2n(n-1)/2 + 1$$

The main advantage of using binomial coefficients is that simplifies the treatment of pattern matching using an *additive shift* operation. For a vector of coefficients $\vec{p} = (p_1, p_2, \ldots, p_k)$, the additive shift of $\vec{p}$ is

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

If xs admits type $L(\vec{p}, A)$, then the tail of xs admits type $L(\lhd\vec{p}, A)$. For the example above, if xs has type $L((3,2),A)$ then the tail xs has type $L((3+2,2),A) = L((5,2),A)$. When defining a function by case over a list, the potential assigned to the tail can then be used to pay for any auxiliary functions or the recursive calls.

3 LANGUAGE AND OPERATIONAL SEMANTICS

3.1 Syntax

We start by introducing a *simple lazy functional language* (SLFL) composed by the syntactical terms e for expressions and w for (weak head) normal forms. Expressions e include constants, variables, lambda expressions, list constructors, let-expressions, and pattern

matching. The values w are in *weak head normal form*[2] and include pairs, list constructors and lambda expressions.

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

$$w ::= n \mid \lambda x.\, e \mid (x_1, x_2) \mid \mathsf{cons}(x_h, x_t) \mid \mathsf{nil}$$

The term syntax is largely based on the semantics used by Jost *et. al.* [10]; the main departure is that we consider only of constructor and de-construction forms for pairs and lists rather than general algebraic data types. This was done to simplify the presentation, and we believe it should be straightforward to extend the system to more general data structures.

As a shorthand, we will sometimes uses semicolons for nested let-expressions, e.g.write

$$\mathsf{let}\ x = e_1; y = e_2\ \mathsf{in}\ e_3$$

instead of

$$\mathsf{let}\ x = e_1\ \mathsf{in}\ \mathsf{let}\ y = e_2\ \mathsf{in}\ e_3\ .$$

3.2 Operational semantics

In this section we present an operational semantics for SLFL. The semantics is based on Sestof's revision of Launchbury's semantics for lazy evaluation [11, 15] instrumented to count the *number of let-expressions* evaluated (corresponding to heap allocations).

The evaluation relation expresses judgements of the form

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

meaning that, under a heap $\mathcal{H}$, a set of bound variables $\mathcal{S}$ and a set of variables $\mathcal{L}$, an expression e evaluates to whnf w, provided m initial resources are available; after evaluation m' resources are left over and the final heap is $\mathcal{H}'$. The rules in Fig. 1 define the evaluation following the structure of the expression.

A heap $\mathcal{H}$ is a mapping from variables to (possibly unevaluated) expressions (*thunks*). To prevent cyclic evaluation, the set $\mathcal{L}$ is used to keep track of the variables that are pending evaluation (cf.rule $\textsc{Var}_{\Downarrow}$). Following Sestof [15], we also track the bound variables in $\mathcal{S}$ to express the freshness side-condition (cf. rule $\textsc{Let}_{\Downarrow}$).

The operational semantics is instrumented by counters that keep track of a specific resource; the objective of the cost analysis is to statically approximate these counters. For simplicity, we consider only the number of allocations; hence, only rule $\textsc{Let}_{\Downarrow}$ in Fig. 1 adds 1 unit of cost (corresponding to one allocation) and other rules simply thread costs between sub-expressions. This could easily be extended to consider different costs, such as the number of steps, number of applications, etc.by assigning parameters to each reduction rule to specifying how many resource units are required, as was done in [5, 10].

Discussing the evaluation rules. As mentioned above, these rules are largely based on the semantics from [10], their construction and meaning are mostly identical. The main differences can be seen in the definition for rules $\textsc{Match-L}_{\Downarrow}$, $\textsc{Match-P}_{\Downarrow}$ and $\textsc{Letcons}_{\Downarrow}$.

[2]That is, evaluated to the outermost constructor or lambda-abstraction.

$$\frac{}{\mathcal{H}, \mathcal{S}, \mathcal{L} \left|\frac{m}{m}\right. w \Downarrow w, \mathcal{H}} \quad (\textsc{whnf}_{\Downarrow})$$

$$\frac{\mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{l\} \left|\frac{m}{m'}\right. e \Downarrow w, \mathcal{H}' \qquad l \notin \mathcal{L}}{\mathcal{H}[l \mapsto e], \mathcal{S}, \mathcal{L} \left|\frac{m}{m'}\right. l \Downarrow w, \mathcal{H}'[l \mapsto w]} \quad (\textsc{Var}_{\Downarrow})$$

$$\frac{l \text{ is fresh wrt } \mathcal{H}, \mathcal{S} \qquad \mathcal{H}[l \mapsto e_1[l/x]], \mathcal{S}, \mathcal{L} \left|\frac{m}{m'}\right. e_2[l/x] \Downarrow w, \mathcal{H}'}{\mathcal{H}, \mathcal{S}, \mathcal{L} \left|\frac{m+1}{m'}\right. \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow w, \mathcal{H}'} \quad (\textsc{Let}_{\Downarrow})$$

$$\frac{\mathcal{H}, \mathcal{S}, \mathcal{L} \left|\frac{m}{m'}\right. e \Downarrow \lambda x.\, e', \mathcal{H}' \qquad \mathcal{H}', \mathcal{S}, \mathcal{L} \left|\frac{m'}{m''}\right. e'[y/x] \Downarrow w, \mathcal{H}''}{\mathcal{H}, \mathcal{S}, \mathcal{L} \left|\frac{m}{m''}\right. e\ y \Downarrow w, \mathcal{H}''} \quad (\textsc{App}_{\Downarrow})$$

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

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

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

Figure 1: Evaluation rules for SLFL

Rule $\textsc{whnf}_{\Downarrow}$: Lambda expressions and constructors are whnfs, so evaluation terminates immediately, leaving the heap unmodified and incurring no cost.

Rule $\textsc{Var}_{\Downarrow}$: Evaluating a variable requires looking up and evaluating the (possibly unevaluated) expression e associated to it in the heap. The cost for evaluating the variable is simply the cost for evaluating the expression. To correctly implement lazy evaluation, we update the location in the final heap in with the normal form.

Rule $\textsc{Let}_{\Downarrow}$: a new thunk for the bound expression e_1 is allocated on the heap and associated with a fresh location l before proceeding to evaluate the body e_2. This rule requires 1 unit of cost, plus the cost of evaluating e_2.

In a lazy language evaluation is forced by case expressions. Hence, rules $\textsc{Match-P}_{\Downarrow}$ and $\textsc{Match-L}_{\Downarrow}$ force the evaluation of

$$[\ldots] \,|^{0}_{0}\, \lambda x.x \Downarrow \lambda x.x, [\ldots] \quad \textsc{Whnf}_{\Downarrow} \quad (1)$$

$$[l_2 \mapsto \lambda x.x, \ldots] \,|^{0}_{0}\, l_2 \Downarrow \lambda x.x,\ [l_2 \mapsto \lambda x.x, \ldots] \quad \textsc{Var}_{\Downarrow}\ (1) \quad (2)$$

$$[\ldots] \,|^{0}_{0}\, \lambda y.y \Downarrow \lambda y.y,\ [\ldots] \quad \textsc{Whnf}_{\Downarrow} \quad (3)$$

$$[l_1 \mapsto \lambda y.y, \ldots] \,|^{0}_{0}\, l_1 \Downarrow \lambda y.y,\ [l_1 \mapsto \lambda y.y, \ldots] \quad \textsc{Var}_{\Downarrow}\ (3) \quad (4)$$

$$[l_1 \mapsto \lambda y.y,\ l_2 \mapsto \lambda x.x,\ \ldots] \,|^{0}_{0}\, l_1\ l_2 \Downarrow \lambda x.x,\ [l_1 \mapsto \lambda y.y,\ l_2 \mapsto \lambda x.x,\ \ldots] \quad \textsc{App}_{\Downarrow}\ (4,2) \quad (5)$$

$$[l_1 \mapsto \lambda y.y,\ l_2 \mapsto \lambda x.x,\ l_3 \mapsto l_1\ l_2,\ \ldots] \,|^{0}_{0}\, l_3 \Downarrow \lambda x.x,\ [l_1 \mapsto \lambda y.y,\ l_2 \mapsto \lambda x.x,\ l_3 \mapsto \lambda x.x, \ldots] \quad \textsc{Var}_{\Downarrow}\ (5) \quad (6)$$

$$[\ldots] \,|^{0}_{0}\, \lambda y.y \Downarrow \lambda y.y,\ [\ldots] \quad \textsc{Whnf}_{\Downarrow} \quad (7)$$

$$[\ldots] \,|^{0}_{0}\, \lambda x.\lambda y.y \Downarrow \lambda x.\lambda y.y,\ [\ldots] \quad \textsc{Whnf}_{\Downarrow} \quad (8)$$

$$[\ldots] \,|^{0}_{0}\, (\lambda x.\lambda y.y)\ l_4 \Downarrow \lambda y.y,\ [\ldots] \quad \textsc{App}_{\Downarrow}\ (8,7) \quad (9)$$

$$[\ldots] \,|^{1}_{0}\, \mathsf{let}\ z = z\ \mathsf{in}\ (\lambda x.\lambda y.y)\ z \Downarrow \lambda y.y,\ [l_4 \mapsto l_4,\ \ldots] \quad \textsc{Let}_{\Downarrow}\ (9) \quad (10)$$

$$[l_1 \mapsto \mathsf{let}\ z = z\ \mathsf{in}\ (\lambda x.\,\lambda y.y)\ z,\ \ldots] \,|^{1}_{0}\, l_1 \Downarrow \lambda y.y, [l_1 \mapsto \lambda y.y,\ l_4 \mapsto l_4,\ \ldots] \quad \textsc{Var}_{\Downarrow}\ (10) \quad (11)$$

$$[l_1 \mapsto \mathsf{let}\ z = z\ \mathsf{in}\ (\lambda x.\,\lambda y.y)\ z,\ l_3 \mapsto l_1\ l_2, \ldots] \,|^{1}_{0}\, l_1\ l_3 \Downarrow \lambda x.x,\ \mathcal{H}' \quad \textsc{App}_{\Downarrow}\ (11,6) \quad (12)$$

$$[l_1 \mapsto \mathsf{let}\ z = z\ \mathsf{in}\ (\lambda x.\,\lambda y.y)\ z,\ l_2 \mapsto \lambda x.x] \,|^{2}_{0}\, \mathsf{let}\ v = l_1\ l_2\ \mathsf{in}\ l_1\ v \Downarrow \lambda x.x, \mathcal{H}' \quad \textsc{Let}_{\Downarrow}\ (12) \quad (13)$$

$$[l_1 \mapsto \mathsf{let}\ z = z\ \mathsf{in}\ (\lambda x.\,\lambda y.y)\ z] \,|^{3}_{0}\, \mathsf{let}\ i = \lambda x.x;\ v = l_1\ i\ \mathsf{in}\ l_1\ v \Downarrow \lambda x.x, \mathcal{H}' \quad \textsc{Let}_{\Downarrow}\ (13) \quad (14)$$

$$|^{4}_{0}\ \ \mathsf{let}\ f = \mathsf{let}\ z = z\ \mathsf{in}\ (\lambda x.\,\lambda y.y)\ z\,; i = \lambda x.x\,; v = f\ i \quad \textsc{Let}_{\Downarrow}\ (14) \quad (15)$$
$$\mathsf{in}\ f\ v \Downarrow \lambda x.x,\ \underbrace{[l_1 \mapsto \lambda y.y, l_2 \mapsto \lambda x.x, l_3 \mapsto \lambda x.x, l_4 \mapsto l_4]}_{\mathcal{H}'}$$

Figure 2: Example of lazy evaluation

the scrutinized expression e_0 and proceed with evaluation of the suitable branch with the variables bound by the pattern matching replaced by the locations of arguments. The final value and heap are the result of evaluating the branch taken.

Example 3.1. Consider the term

$$\begin{array}{ll} \mathsf{let} & f = \mathsf{let}\ z = z\ \mathsf{in}\ (\lambda x.\,\lambda y.y)\ z\,; \\ & i = \lambda x.x\,; \\ & v = f\ i \\ \mathsf{in} & f\ v \end{array} \quad (1)$$

Fig. 2 represents the evaluation of (1) to the whnf $\lambda x.\,x$ and final heap $\mathcal{H}' = [l_1 \mapsto \lambda y.y, l_2 \mapsto \lambda x.x, l_3 \mapsto \lambda x.x, l_4 \mapsto l_4]$. For simplicity, we omit sets $\mathcal{S}, \mathcal{L}$ of bound variables and those under evaluation.

This example illustrate how the rules of Fig. 1 implement lazy evaluation: evaluating f causes the allocation of a cyclic thunk z that is never needed; the final judgement is annotated 4, 0 corresponding to the requirement of 4 resource units for allocating the four let-bound expressions.

4 ANALYSIS WITH POLYNOMIAL POTENTIAL

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.

4.1 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, functions, thunks, pairs and lists. Variables p, q represent potential and cost annotations and $\vec{p}, \vec{q}$ represent *vectors* of such annotations, $\vec{p} = (p_1, \ldots, p_n)$.

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

The primitive types and pairs have no annotations. The annotation q on a function type $A \xrightarrow{q} B$ represents an upper bound on the (constant part) of the cost of applying the function. The annotation q on a thunk type $\mathsf{T}^q(A)$ represents an upper bound on the cost of evaluating to whnf a thunk giving a value of type A. The list type $\mathsf{L}^q(\vec{p}, A)$ is annotated with q, representing the cost of evaluating each new constructor of the list (i.e.the spine cost) and a vector $\vec{p}$, which representing the potential associated with the list elements.[3]

Following Hoffmann (see section 2.5), we define the *additive shift* $\lhd\vec{p}$ of a vector of annotations:

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

We also define an addition operation and comparison on vectors $\vec{p}$ and $\vec{q}$ of equal length n:

$$\vec{p} + \vec{q} = (p_1 + q_1,\ p_2 + q_2,\ \ldots,\ p_n + q_n)$$
$$\vec{p} \geq \vec{q} \iff p_i \geq q_i\,, \text{for all } i$$

[3]Note that, unlike Hoffman [5], and for consistency with the annotations in thunk types, we use superscripts for delayed costs.

$$\frac{}{\mathsf{int} \between \{\mathsf{int}, \ldots, \mathsf{int}\}} \quad (\textsc{ShareInt})$$

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

$$\frac{A \between \{A_1, \ldots, A_n\} \qquad \vec{p} \geq \sum_i \vec{p}_i \qquad q_i \geq q}{\mathsf{L}^q(\vec{p}, A) \between \{\mathsf{L}^{q_1}(\vec{p}_1, A_1), \ldots, \mathsf{L}^{q_n}(\vec{p}_n, A_n)\}} \quad (\textsc{ShareList})$$

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

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

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

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

Figure 3: Sharing rules

4.2 Sharing and Subtyping

Because types can be annotated with potential, it is necessary to limit the duplication of typing assumptions (i.e.contraction) so that potential is not re-used; this is done by a *sharing relation*. Informally, a type A shares to a set of types $\{B_1, \ldots, B_n\}$ if any potential in A is distributed over $B_1, \ldots, B_n$. The sharing relation $A \between \{B_1, \ldots, B_n\}$ is defined inductively over the structure of the types A and B_i in Fig. 3. It is straightforward to verify that if $A \between \{B_i\}$ then A and all B_i differ only in annotations (i.e.have the same structure). Note that we also extend sharing to typing contexts (rules ShareCtx and ShareEmptyCtx).

Sharing also allows relaxing information by *lowering* potential on the left-hand side (rule ShareList) or *raising* costs on the right-hand side (rules ShareFun, ShareThunk and ShareList). This in turn allows defining a subtyping relation for approximations as a special case of sharing:

$$A <: B \iff \text{exists } B' \text{ such that } A \between \{B, B'\}$$

If $A <: B$ then the cost information in A is more precise but compatible with the one in B. It is straightforward to check that the following properties are admissible:

$$\mathsf{T}^q(A) <: \mathsf{T}^{q'}(A') \iff A <: A' \wedge q \geq q'$$
$$A \times B <: A' \times B' \iff A <: A' \wedge B <: B'$$
$$A \xrightarrow{q} B <: A' \xrightarrow{q'} B' \iff A' <: A \wedge B <: B' \wedge q \geq q'$$
$$\mathsf{L}^q(\vec{p}, A) <: \mathsf{L}^{q'}(\vec{p'}, A') \iff A <: A' \wedge q \geq q' \wedge \vec{p} \geq \vec{p'}$$

Note that, as usual, the subtyping relation is co-variant for pairs, lists and co-domain of functions but contra-variant for the domain of functions.

4.3 Type Rules

The type rules deriving judgements for the different syntax forms our term language are presented in Fig. 4; these are complemented with the structural rules in Fig. 5. Typing judgements have the form

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

and state that, under 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. While the type rules are based on previous work [5, 10], there are important differences in rules that concern the use of potential, namely, Letcons, Cons and Match. We 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 require any resources as evaluating a primitive value incurs no additional cost.

Rule Var deals with the elimination of a thunk type, so it suffices to pay the cost associated in the annotation.

Rules Let and Letcons require paying 1 unit of cost (corresponding to the newly allocated expression); note also that (as in the operational semantics) the bound variable x can be used recursively in the bound expression. Note that rule Letcons accounts not only for the allocation cost but also the potential q_1 in the type annotation; this why we need a separate rule rather than just allow a combination of Let and Cons. The side condition $A \between \{A, \ldots, A'\}$ is used to guarantee that the type A' is identical to A except that it cannot hold any potential; this is for soundness (so that self-referencing structures cannot consume arbitrary potential [10]). Rule Let counts the cost of e_1 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 [18].

Rules Cons and Pair require that constructors and pairs reference variables of the correct types. For Cons, it requires that the tail of a list be annotated with the additive shift of the list's potential. Note that the potential is accounted for in rule Letcons rather than Cons; this is done to distinguish referencing an existing constructor from allocating of a new one.

Rule App requires that the cost associated with a function type is count for each time the function is applied.

Rule Abs captures the cost of the expression as the type annotation of the function type. The side-condition $\Gamma \between \{\Gamma, \Gamma\}$ ensures that the typing context Γ can only have zero potential; this ensures that the function may be freely applied without re-use of potential.

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

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

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

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

$$\frac{\begin{array}{c} A \between \{A, A'\} \qquad x \notin \{\Gamma, \Delta\} \qquad e_1 \neq \mathsf{cons}(x_h, x_t) \\ \Gamma, x : \mathsf{T}^{0}(A') \left|\frac{p}{0}\right. e_1 : A \qquad \Delta, x : \mathsf{T}^{p}(A) \left|\frac{z}{z'}\right. e_2 : C \end{array}}{\Gamma, \Delta \left|\frac{1+z}{z'}\right. \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : C} \quad (\textsc{Let})$$

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

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

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

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

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

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

Figure 4: Syntax directed type rules

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

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

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

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

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

Figure 5: Structural type rules

Rule Match-P deconstructs a pair, introducing in the context the pattern variables with the corresponding types.

Rule Match-L deconstructs a list, considering the empty and non-empty cases. Note that, when typing the right-hand side e_1 for the non-empty case, we gain access to both the excess potential q_1 and the variable x_t for the tail associated with the additive shift of the original potential; to allows paying for (possible recursive) application. This rule also requires that both branches admit the same type C and that the resources z'' available after each branch are identical; this may require the use of structural rules for costs or types (cf.rules Relax and Subtype in Fig. 5).

Regarding the structural type rules in Fig. 5: rule Prepay allow paying ahead (part of) the cost annotated in thunk; this is identical to the rules in [10, 16]. Note also that the type system allows weakening Weak but requires a constrained contraction rule Share. As mentioned in Section 4.2, this is done to prevent re-using potential annotated in types.

4.4 Soundness

We do not have a formal proof of soundness yet, but we believe that previous work in [10] can be adapted to the polynomial potential case; in particular, we need some auxiliary definitions:

- the (shallow) potential of an expression e wrt an annotated type A;
- a *global type* of each location ℓ that justifies the overall potential for $\mathcal{H}(\ell)$;
- a *global context* of each location ℓ used for typing the expression $\mathcal{H}(\ell)$;

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

Figure 6: Translation of the *pairs* function and auxiliary definitions into SLFL.

- a *type consistency relation* between configurations of the operational semantics and global types and contexts.

Informally the soundness theorem should state that if an expression e admits an (annotated) type A, the heap $\mathcal{H}$ can be typed, and the evaluation of e is successful, then the result also admits type A. Furthermore, the potential in global types is preserved, the final type can also be typed, and the static bounds from the typing for e give safe upper-bounds on the evaluation costs.

5 WORKED EXAMPLES

To better understand how the resource analysis works, derive annotated type judgements for the example presented in Section 1.

5.1 Examples

Example 5.1. Let us consider function *pairs* in Fig. 6; this is a translation into SLFL of the example from Section 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. The auxiliary function app' is the list append argument order flipped, i.e.$app' = \texttt{flip (++)}$; this is done so that the type rules allow assigning potential to this argument.[4].

To facilitate the presentation of annotated type assignments, we have added potential annotations to list variables in Fig. 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. 6, we derive the following constraints:

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

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

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

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:

$$app' : \mathsf{T}^0(\mathsf{L}^0(0, B \times B)) \xrightarrow{0} \mathsf{T}^0(\mathsf{L}^0(1, B \times B)) \xrightarrow{0} \mathsf{L}^0(0, B \times B)$$

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

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:

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

$$\Sigma = attach : B \xrightarrow{0} \mathsf{T}^0(\mathsf{L}^0(3, B)) \xrightarrow{0} \mathsf{L}^0(1, B \times B)$$

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:

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

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

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

$$\Gamma, \Sigma \left|\frac{1}{0}\right. \text{let } pairs = \lambda l.\, e_1 \text{ in } pairs{:}\mathsf{T}^0(L_{In}) \xrightarrow{p} L_{Out} \tag{2}$$

By rule Let, we need to prove:

$$\Gamma, \Sigma, \Theta \left|\frac{0}{0}\right. \lambda l.\, e_1 : \mathsf{T}^0(L_{In}) \xrightarrow{p} L_{Out} \tag{3}$$

The later follows from rule Abs if we prove:

$$\Gamma, \Sigma, \Theta,\ l{:}\mathsf{T}^0(L_{In}) \left|\frac{p}{0}\right. e_1 : L_{Out} \tag{4}$$

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:

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

$$\left|\frac{0}{0}\right. \mathsf{nil}{:}L_{Out} \tag{Nil}$$

The remaining case for non-empty lists is:

$$\Gamma, \Sigma, \Theta,\ x{:}B,\ xs{:}\mathsf{T}^0(\mathsf{L}^0((q_1+q_2, q_2), B)) \left|\frac{q_1}{0}\right. e_2{:}L_{Out} \tag{5}$$

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:

$$\mathsf{L}^0((q_1+q_2, q_2), B) \curlyvee \{\mathsf{L}^0((p_1, p_2), B),\ \mathsf{L}^0((s_1, s_2), B)\} \tag{6}$$

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:

$$\Gamma, \Sigma, \Theta,\ x{:}B,\ \Delta_1, \Delta_2 \left|\frac{2}{0}\right. \begin{array}{ll} \text{let} & f_1 = pairs\ xs; \\ & f_2 = attach\ x\ xs \\ \text{in} & app'\ f_1\ f_2 \end{array} : L_{Out} \tag{7}$$

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:

$$\Theta, \Delta_1 \left|\frac{0}{0}\right. pairs\ xs : \mathsf{L}^0((0,0), B \times B) \tag{8}$$

$$\Sigma, \Delta_2, x{:}B \left|\frac{0}{0}\right. attach\ x\ xs : \mathsf{L}^0(0, B \times B) \tag{9}$$

Judgments (8) and (9) 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:

$$\Delta_2, \Gamma, \Theta, f_1{:}\mathsf{T}^0(L_{Out}) \left|\frac{1}{0}\right. \text{let } f_2 = attach\ xxs \text{ in } app'\ f_1 f_2{:}L_{Out} \tag{10}$$

This follows from the rules Var and App twice:

$$\Gamma, f_1{:}\mathsf{T}^0(L_{Out}), f_2{:}\mathsf{T}^0(\mathsf{L}^0(1, B \times B)) \left|\frac{0}{0}\right. app'\ f_1 f_2{:}L_{Out} \tag{11}$$

With this detailed illustration it is easy to see where the constraints mentioned before come from. From (8), (9) and (10) we get $p_1 = 3, p_2 = 0, s_1 = q_1$ and $s_2 = q_2$. From (5) and (7) we get $q_1 \geq 2$. From (6) 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:

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

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 5.2. 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:[5]

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

$$\Sigma = attach : B \xrightarrow{0} \mathsf{T}^0(\mathsf{L}^1(4, B)) \xrightarrow{0} \mathsf{L}^0(1, B \times B)$$

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.

$$\Gamma, \Sigma, \Theta,\ x{:}B,\ xs{:}\mathsf{T}^p(\mathsf{L}^1((q_1+q_2, q_2), B)) \left|\frac{q_1}{0}\right. e_2{:}L_{Out} \tag{12}$$

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:

$$\Gamma, \Sigma, x{:}B,\ xs{:}\mathsf{T}^1(\mathsf{L}^1((q_1+q_2, q_2), B)) \left|\frac{3}{0}\right. e_2{:}L_{Out} \tag{Prepay}$$

$$\Gamma, \Sigma, x{:}B,\ xs{:}\mathsf{T}^0(\mathsf{L}^1((q_1, q_2), B)) \left|\frac{2}{0}\right. e_2{:}L_{Out} \tag{Share}$$

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

The use of Share generates the following side condition:

$$\mathsf{T}^0(\mathsf{L}^1((q_1+q_2,q_2),B)) \curlyvee \{\mathsf{T}^0(\mathsf{L}^1((p_1,p_2),B)),\ \mathsf{T}^0(\mathsf{L}^1((s_1,s_2),B))\} \tag{13}$$

Note that, although the outermost thunks have been reduced by the use of Prepay, the list spine thunks still cost 1 because sharing distributes potential but not costs (See Fig. 3).

The remaining derivation is:

$$\Gamma,\Sigma,x{:}B,\ xs{:}\mathsf{T}^0(\mathsf{L}^1((p_1,p_2),B)),\ xs{:}\mathsf{T}^0(\mathsf{L}^1((s_1,s_2),B)) \mid^2_0 e_2{:}L_{Out} \tag{14}$$

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 assumption for *attach*) and $q_1 \geq 3$ (because of the use of Prepay after (14)). 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.

5.2 Cost Overestimation

Comparing the bounds obtained for the examples we note an overestimation. The example 5.2 is identical to 5.1 except for the extra cost for each list constructors; hence we would expect to pay only extra n units. However, the difference between the bounds is $3\times n + 2\times n\times(n-1) - (\frac{3}{2}\times n\times(n-1)) = n + \frac{1}{2}\times n\times(n-1)$.

The overestimation results from the two uses of *xs* in the body of *pairs*: sharing allows distributing the list potential but not the spine thunk costs. Recall equation (13):

$$\mathsf{T}^0(\mathsf{L}^1((q_1+q_2,q_2),B)) \curlyvee \{\underbrace{\mathsf{T}^0(\mathsf{L}^1((p_1,p_2),B))}_{attach},\ \underbrace{\mathsf{T}^0(\mathsf{L}^1((s_1,s_2),B))}_{pairs}\}$$

The uses for *attach* and the recursive call each pay for the unit thunk cost for the spine even though thunks are evaluated once. It might appear that we should always allow costs to be shared, i.e.replace the side condition $q_i \geq q$ of rule ShareList (cf.Fig. 3):

$$\frac{A \curlyvee \{A_1,\ldots,A_n\} \qquad \vec{p} \geq \sum_i \vec{p}_i \qquad \sum_i q_i \geq q}{\mathsf{L}^q(\vec{p},A) \curlyvee \{\mathsf{L}^{q_1}(\vec{p}_1,A_1),\ldots,\mathsf{L}^{q_n}(\vec{p}_n,A_n)\}}\ (\textsc{ShareList}')$$

In the above example the revised rule would allow deriving e.g.

$$\mathsf{T}^0(\mathsf{L}^1((q_1+q_2,q_2),B)) \curlyvee \{\underbrace{\mathsf{T}^0(\mathsf{L}^1((p_1,p_2),B))}_{attach},\ \underbrace{\mathsf{T}^0(\mathsf{L}^0((s_1,s_2),B))}_{pairs}\}$$

meaning only *attach* would pay for the thunk costs (alternatively, we could also have the cost in the recursive call). However, note that rule ShareList' is, in general, unsound: firstly, we may discard assumptions with positive cost (possibly leading to cost underestimating); secondly, even if assumptions are used, there is no guarantee that two uses evaluate the list to the same depth.

The first issue can be solved by making the type system *relevant* i.e., removing the rule for weakening and introducing binders that do not add variables to typing contexts. We conjecture that the second issue could be avoided in some cases, e.g.in a tail strict context [19]. Note that *pairs* is not tail strict but nonetheless, we believe ShareList' should be sound; we leave further research into the characterisation of sound contexts as an open problem.

6 CONCLUSION AND FURTHER WORK

In this paper, we present the first amortised resource analysis for higher-order lazy functional programs with polynomial bounds. We show how we combine main concepts from previous systems in order to reach this goal: the usage of thunk types and prepaying for lazy evaluation and the additive shift for polynomial potential. Our type system has been successfully applied to some small examples.

Our analysis does not allow resource polymorphic recursion, i.e., recursive calls with different resource annotations. As it happens 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 [1, 5]. 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 already 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 can be used in our system.

REFERENCES

[1] Jan Hoffmann. 2011. *Types with potential: polynomial resource bounds via automatic amortized analysis.* Ph.D. Dissertation. Ludwig Maximilians University Munich.

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

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

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

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

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

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

[8] John Hughes and Chalmers Hogskola. 1999. Why Functional Programming Matters. (05 1999).

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

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

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

[12] Neil Mitchell. 2013. Leaking space. *Commun. ACM* 56, 11 (2013), 44–52.

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

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

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

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

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

[18] Pedro Vasconcelos, Steffen Jost, Mário Florido, and Kevin Hammond. 2015. Type-Based Allocation Analysis for Co-recursion in Lazy Functional Languages. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 787–811.

[19] Philip Wadler and R. J. M. Hughes. 1987. Projections for strictness analysis. In *Functional Programming Languages and Computer Architecture*, Gilles Kahn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 385–407.