Analysis of Logarithmic Amortised Complexity

Martin Hofmann †
Department of Computer Science
LMU Munich, Germany

Georg Moser¹
Department of Computer Science
University of Innsbruck, Austria
georg.moser@uibk.ac.at
http://orcid.org/0000-0001-9240-6128

Abstract
We introduce a novel amortised resource analysis based on a potential-based type system. This type system gives rise to logarithmic and polynomial bounds on the runtime complexity and is the first such system to exhibit logarithmic amortised complexity. We relate the thus obtained automatable amortised resource analysis to manual amortised analyses of self-adjusting data structures, like splay trees, that can be found in the literature.

2012 ACM Subject Classification F.3.2 Program Analysis

Keywords and phrases analysis of algorithms, amortised resource analysis, functional programming, self-adjusting data structures

1 Introduction
In a series of papers a number of researchers including the present authors, see [3,4,6,8–10,12,13,14] to name just a few, have explored the area of type-based automated amortised analysis which lead to several successful tools for deriving accurate bounds on the resource usage of functional [4,13] and also imperative programs [7,11], as well as term rewriting [1,14]. Accordingly type-based amortised analysis has been employed on a variety of cost metrics. While initially confined to linear resource bounds [8] the methods were subsequently extended to cover polynomial [6], multivariate polynomial [4], and also exponential bounds [11]. However, the automated analysis of sublinear, in particular logarithmic resource bounds remained elusive.

One notable exception is [15] where the correct amortised analysis of splay trees [19,20] and other data structures is certified in Isabelle/HOL with some tactic support. However, although the analysis is naturally coached in a type system (namely the type system of the proof assistant Isabelle, one cannot speak there of fully automated analysis. It is in fact not at all clear how a formalisation in an interactive theorem prover as such leads to automation.

It is the purpose of this paper to open up a route towards the automated, type-based derivation of logarithmic amortised cost. While we do not yet have a prototype implementation, let alone viable experimental data, we make substantial progress in that we present a system akin to the multivariate analysis from [3,4] which reduces the task of justifying a purported logarithmic complexity bound to the validity of a well-defined set of inequalities involving linear arithmetic and logarithmic terms. We also give concrete ideas as to how one can infer logarithmic bounds efficiently by using an Ansatz with unknown coefficients.

¹ Partly supported by DARPA/AFRL contract number FA8750-17-C-088.
Our analysis is coached in a simple core functional language just sufficiently rich to provide a full definition of our motivating example: splaying. We employ a big-step semantics, following similar approaches in the literature. However, this implies that our resource analysis requires termination, which is typically not the case. It is straightforward to provide a partial big-step semantics [5] or a small-step semantics [14] to overcome this assumption. Furthermore, the proposed type system is geared towards runtime as computation cost. Again it would not be difficult to provide a parametric type system. We consider both issues as complementary to our main agenda.

Organisation

The rest of this paper is organised as follows. In the next section we introduce a simple core language underlying our reasoning and provide a full definition of splaying, our running example. In Section 3 we provide background and a high-level description of our approach. The employed notion of potential function is provided in Section 4, while our main result is established in Section 5. In Section 6 we employ the established type system to splaying, while in Section 7 we clarify the aforementioned Ansatz to infer logarithmic bounds. Finally, we conclude in Section 8.

2 Motivating Example

In this section, we introduce the syntax of a suitably defined core (first-order) programming language to be used in the following. Furthermore, we recall the definition of splaying, following the presentation by Nipkow in [15]. Splaying constitutes the motivating examples for the type-based logarithmic amortised resource analysis presented in this paper.

To make the presentation more succinct, we assume only the following types: Booleans (\(\text{Bool} = \{\text{true}, \text{false}\}\)), an abstract base type \(B\), product types, and a type \(T\) of binary trees whose internal nodes are labelled with elements \(a:B\). Elements \(t:T\) are defined by the following grammar which fixes notation.

\[
t ::= \text{nil} \mid (t,a,t).
\]

The size of a tree is the number of leaves: \(|\text{nil}| := 1\), \(|(t,a,u)| := |t| + |u|\).

Expressions are defined as follows and given in let normal form to simplify the presentation of the semantics and typing rules. In order to ease the readability, we make use of some mild syntactic sugaring in the presentation of actual code.

\[\begin{align*}
\text{cmp} & ::= < \mid > \mid = \\
\text{e} & ::= \text{true} \mid \text{false} \mid x \mid e \text{ cmp } e \mid \text{if } e \text{ then } e \text{ else } e \\
& \quad \mid \text{let } x = e \text{ in } e \mid f(x,\ldots,x) \\
& \quad \mid \text{match } x \text{ with } \text{nil} \to e \mid \langle x, x, x \rangle \to e
\end{align*}\]

We skip the standard definition of integer constants \(n \in \mathbb{Z}\) as well as variable declarations, cf. [17]. Furthermore, we omit binary operations and focus on the bare essentials for the comparison operators. For the resource analysis these are not of importance, as long as we assume that no actual costs are emitted.

A typing context is a mapping from variables \(V\) to types. Type contexts are denoted by upper-case Greek letters. A program \(P\) consists of a signature \(F\) together with a set...
We write all operations rely on a tree-rotating operation dubbed Tarjan [19], Schoenmakers [18], Nipkow [15], Okasaki [16], among others. Below, we follow Nipkow’s approach, where the actual cost of splaying is measured by counting the number of calls to \texttt{splay}: \( B \times T \rightarrow T \).

\[
\begin{align*}
 b \in \{\text{true, false}\} & \quad \sigma \vdash b \Rightarrow b \\
 \sigma \vdash \text{nil} \Rightarrow \text{nil} \\
 x\sigma = v & \quad \sigma \vdash x \Rightarrow v \\
 f(x_1, \ldots, x_k) = e \in P & \quad \sigma \vdash e \Rightarrow v \\
 b \text{ is value of } x_1 \sigma \text{ cmp } x_2\sigma & \quad \sigma \vdash x_1 \text{ cmp } x_2 \Rightarrow b \\
 \sigma \vdash x \Rightarrow \text{true} & \quad \sigma \vdash e_1 \Rightarrow v \\
 \sigma \vdash x \Rightarrow \text{false} & \quad \sigma \vdash e_2 \Rightarrow v \\
 \sigma \vdash \text{if } x \text{ then } e_1 \text{ else } e_2 \Rightarrow v & \\
 \sigma \vdash \text{match } x \text{ with } |\text{nil} \rightarrow e_1 \Rightarrow v | \{x_1, x_2, x_3 \rightarrow e_2 \Rightarrow v\} & \\
 x\sigma = \text{nil} & \quad \sigma \vdash e \Rightarrow v \\
 \sigma \vdash \text{match } x \text{ with } |\text{nil} \rightarrow e_1 \Rightarrow v | \{x_1, x_2, x_3 \rightarrow e_2 \Rightarrow v\} & \\
 x\sigma = (t, a, u) & \quad \sigma' \vdash e_2 \Rightarrow v \\
 \sigma \vdash \text{match } x \text{ with } |\text{nil} \rightarrow e_1 \Rightarrow v | \{x_1, x_2, x_3 \rightarrow e_2 \Rightarrow v\} & \\
 \end{align*}
\]

Here \( \sigma[x \mapsto v'] \) denotes the update of the environment \( \sigma \) such that \( \sigma[x \mapsto v'](x) = v' \) and the value of all other variables remains unchanged. Furthermore, in the second \texttt{match} rule, we set \( \sigma' := \sigma \uplus \{ x_0 \mapsto t, x_1 \mapsto a, x_2 \mapsto u \} \).

\textbf{Figure 1} Big-Step Semantics

of function definitions of the form \( f(x_1, \ldots, x_n) = e \), where the \( x_i \) are variables and \( e \) an expression. A substitution or (environment) \( \sigma \) is a mapping from variables to values that respects types. Substitutions are denoted as sets of assignments: \( \sigma = \{ x_1 \mapsto t_1, \ldots, x_n \mapsto t_n \} \).

We write \( \text{dom}(\sigma) \) (\( \text{rg}(\sigma) \)) to denote the domain (range) of \( \sigma \). Let \( \sigma, \tau \) be substitutions such that \( \text{dom}(\sigma) \cap \text{dom}(\tau) = \emptyset \). Then we denote the (disjoint) union of \( \sigma \) and \( \tau \) as \( \sigma \uplus \tau \). We employ a simple cost-sensitive big-step semantics, whose rules are given in Figure 1. The judgement \( \sigma \vdash e \Rightarrow v \) means that under environment \( \sigma \), expression \( e \) is evaluated to value \( v \) in exactly \( m \) steps. Here only rule applications emit (unit) costs.

\textit{Splay trees} have been introduced by Sleator and Tarjan [19] [20] as self-adjusting binary search trees with strictly increasing inorder traversal. There is no explicit balancing condition. All operations rely on a tree-rotating operation dubbed \texttt{splaying}: \texttt{splay a t} is performed by rotating element \( a \) to the root of tree \( t \) while keeping inorder traversal intact. If \( a \) is not contained in \( t \), then the last element found before \texttt{nil} is rotated to the tree. The complete definition is given in Figure 2. Based on \texttt{splaying}, searching is performed by \texttt{splaying} with the sought element and comparing to the root of the result. Similarly, the definition of insertion and deletion depends on \texttt{splaying}. Exemplary the definition of insertion is given in Figure 3. See also [15] for full algorithmic, formally verified, descriptions.

All basic operations can be performed in \( O(\log n) \) amortised runtime. The logarithmic amortised complexity is crucially achieved by local rotations of subtrees in the definition of \texttt{splay}. Amortised cost analysis of \texttt{splaying} has been provided for example by Sleator and Tarjan [19], Schoenmakers [18], Nipkow [15], Okasaki [16], among others. Below, we follow Nipkow’s approach, where the actual cost of \texttt{splaying} is measured by counting the number of calls to \texttt{splay}: \( B \times T \rightarrow T \).
3 Background and Informal Presentation

To set the scene we briefly review the general approach up to and including the multivariate polynomial analysis from Hoffmann et al. [3,4,10].

Univariate Analysis

Suppose that we have types $A, B, C, \ldots$ representing sets of values. We write $[A]$ for the set of values represented by type $A$. Types may be constructed from base types by type formers such as list, tree, product, sum, etc.

For each type $A$ we have a, possibly infinite, set of basic potential functions $BF(A) : [A] \rightarrow \mathbb{R}_0^+$. Thus, if $p \in BF(A)$ and $v \in [A]$ then $p(v) \in \mathbb{R}_0^+$. It is often useful to regard $BF(A)$ as set of names for basic potential functions. In this case, we have a function $\langle -, - \rangle : BF(A) \times [A] \rightarrow \mathbb{R}_0^+$. To ease notation, one then sometimes writes $p(v)$, instead of $(p, v)$.

An annotated type is a pair of a type $A$ and a function $Q : BF(A) \rightarrow \mathbb{R}_0^+$ providing a coefficient for each basic potential function. The function $Q$ must be zero on all but finitely many basic potential functions. For each annotated type $A|Q$, the potential function

```plaintext
splay a t = match t with
| nil -> nil
| ⟨cl, c, cr⟩ ->
  if a = c then ⟨cl, c, cr⟩
  else if a < c then match cl with
  | nil -> ⟨cl, c, cr⟩
  | ⟨bl, b, br⟩ ->
    if a = b then ⟨bl, a, ⟨br, c, cr⟩⟩
    else if a < b
      then if bl=nil then ⟨bl, b, ⟨br, c, cr⟩⟩
      else match splay a bl with
        | ⟨al, a', ar⟩ -> ⟨al, a', ar, b, ⟨br, c, cr⟩⟩
      else if br=nil then ⟨bl, b, ⟨br, c, cr⟩⟩
      else match splay a br with
        | ⟨al, a', ar⟩ -> ⟨bl, a, ⟨ar, c, cr⟩⟩
  else match cr with
  | nil -> ⟨cl, c, cr⟩
  | ⟨bl, b, br⟩ ->
    if a = b then ⟨⟨cl, c, bl⟩, a, br⟩
    else if a < b
      then if bl=nil then ⟨⟨cl, c, bl⟩, b, br⟩
      else match splay a bl with
        | ⟨al, a', ar⟩ -> ⟨⟨cl, c, bl⟩, a', ar, b, br⟩⟩
      else if br=nil then ⟨⟨cl, c, bl⟩, b, ⟨ar, c, cr⟩⟩
      else match splay a br with
        | ⟨al, a', ar⟩ -> ⟨⟨cl, c, br⟩, b, ⟨ar, c, cr⟩⟩
```

Figure 2 Function splay.
M. Hofmann and G. Moser

Figure 3 Function `insert`.

```plaintext
define function insert {
  t = if t=nil then ⟨nil, a, nil⟩
  else match splay a t with
    | ⟨l, a', r⟩ =>
      if a=a' then ⟨l, a, ⟨nil, a', r⟩⟩
      else if a<a' then ⟨l, a, ⟨nil, a', r⟩⟩
      else ⟨⟨l, a', nil⟩, a, r⟩
}
```

Figure 4 Functions `delete` and `splay_max`.

```plaintext
define function delete {
  t = if t=nil then nil
  else match splay a t with
    | ⟨l, a', r⟩ =>
      if a=a' then if l=nil then r
      else match splay_max l with
        | ⟨l', m, r'⟩ => ⟨l', m, r⟩
      else ⟨⟨l, a', nil⟩, a, r⟩
}
```

```plaintext
define function splay_max {
  t = match t with
    | nil => nil
    | ⟨l, b, r⟩ => match r with
      | nil => ⟨l, b, nil⟩
      | ⟨rl, c, rr⟩ =>
        if rr=nil then ⟨⟨l, b, rl⟩, c, nil⟩
        else match splay_max rr with
          | ⟨rrl, x, xa⟩ => ⟨⟨⟨l, b, rl⟩, c, rrl⟩, x, xa⟩
}
```

φ : [A] → ℝ⁺ is given by

\[ \phi_Q(v) := \sum_{p \in BF(A)} Q(p) \cdot p(v) \cdot \]

Now suppose that we have a function \( f : A_1 \times \cdots \times A_n \rightarrow B \) and that the actual cost for computing \( f(v_1, \ldots, v_n) \) is given by \( c(v_1, \ldots, v_n) \) where \( c : [[A_1]] \times \cdots \times [[A_n]] \rightarrow ℝ⁺ \). The idea then is to choose annotations \( Q_1, \ldots, Q_n, Q \) of \( A_1, \ldots, A_n \) and \( B \) in such a way that the amortised cost of \( f \) becomes zero or constant, i.e.

\[ \phi_{Q_1}(v_1) + \cdots + \phi_{Q_n}(v_n) \geq c(v_1, \ldots, v_n) + \phi_Q(f(v_1, \ldots, v_n)) + d, \]

where \( d \in ℝ⁺ \). The potential of the input suffices to pay for the cost of computing \( f(v_1, \ldots, v_n) \) as well as the potential of the result. This allows one to compose such judgements in a syntax-oriented way without having to estimate sizes, let alone the precise form of intermediate results, which is often needed in competing approaches.

If we introduce product types, we can regard functions with several arguments as unary functions: Let \( A_1, \ldots, A_n \) be types, then so is \( A_1 \times \cdots \times A_n \). We conclusively define \( [[A_1 \times \cdots \times A_n]] := [[A_1]] \times \cdots \times [[A_n]] \) and \( BF(A_1 \times \cdots \times A_n) = BF(A_1) \uplus \cdots \uplus BF(A_n) \), where \( \uplus \) stands for disjoint union. Furthermore, we define \( ⟨\text{in}_i(p), (v_1, \ldots, v_n)⟩ := ⟨p, v_i⟩ \). If
we now regard $f$ above as a unary function from $A_1 \times \cdots \times A_n$ to $B$ then it is not hard to see that the notions of annotation and amortised cost agree with the multi-ary ones given above.

This approach has been key to lift earlier results on automated resource analysis, e.g. \cite{8}, restricted to linear bounds to polynomial bounds. In particular, in \cite{6} an automated amortised resource analysis has been introduced exploiting these ideas. This analysis employs binomial coefficients as basic potential functions. The approach generalises to general inductive data types, cf. \cite{9,10,14}.

Multivariate Analysis

In the multivariate version of automated amortised analysis \cite{3,4,10} one takes a more general approach to products. Namely, one then puts

$$\mathcal{BF}(A_1 \times \cdots \times A_n) := \mathcal{BF}(A_1) \times \cdots \times \mathcal{BF}(A_n)$$

$$\langle (p_1, \ldots, p_n), (v_1, \ldots, v_n) \rangle := \langle p_1, v_1 \rangle \cdots \langle p_n, v_n \rangle ,$$

i.e. the basic potential function for a product type is obtained as the multiplication of the basic potential functions of its constituents. In order to achieve backwards compatibility, that is, to recover all the potential functions available in the univariate case, it is necessary to postulate for each type $A$ a distinguished element $1 \in \mathcal{BF}(A)$ with $(1, a) = 1$ for all $a \in \mathbb{A}$.

Consider automatisation of the univariate or multivariate analysis. Suppose that it is possible to derive amortised costs for basic functions like constructors, if-then-else etc. Then one sets up annotations with indeterminate coefficients and solves for them so as to automatically infer costs. This is in particular possible when the basic potential functions for datatypes like lists or trees are polynomial functions of length and other size parameters.

One of the reasons why this works so well is that if $p(n)$ is a polynomial, so is $p(n+1)$ and in fact can be expressed as a linear combination of basic polynomials like, e.g., powers of $x$ or binomial coefficients, cf. \cite{3,14}. This approach also generalises to general inductive data types, cf. \cite{9,10,14}.

In contrast to the univariate system, the multivariate system provides for greater accuracy because it can derive bounds like $mn$ which in the univariate analysis would be over-approximated by $m^2 + n^2$. This, however, requires a more careful management of variables and contexts resulting in rather involved typing rules for composition (\texttt{let}) and sharing, where \texttt{sharing} refers to the multiple use of variables.

Since we need a similar mechanism in the present system we will explain this in a little more detail. Let $f : A \to B$ and $g : B \times C \to D$ be functions and suppose that evaluating $f(x)$ and $g(y, z)$ incurs costs $c(x)$ and $d(y, z)$, respectively. Suppose further the following constraints hold for all $x \in \mathbb{A}$, $y \in \mathbb{B}$, $z \in \mathbb{D}$ and potential functions $\phi_i$, $\phi'_i$, $\phi''_i$, $\psi$:

$$\phi_0(x) \geq c(x) + \phi'_0(f(x))$$

$$\phi_i(x) \geq \phi'_i(f(x)) \quad \text{for all } i \ (0 < i \leq n)$$

$$\phi'_0(y) + \sum_{i=1}^n \phi'_i(y)\phi''_i(z) \geq d(y, z) + \psi(g(y, z)) .$$

Then we conclude for all $x, y, z : \phi_0(x) + \sum_{i=1}^n \phi_i(x)\phi'_i(z) \geq c(x) + d(g(f(x), y)) + \psi(g(f(x), y))$ guaranteeing that a suitable combination of the potential of the arguments, suffices to pay for the cost $c(x)$ of computing $f(x)$, the cost $d(g(f(x), y))$ of the function composition $g(f(x), y)$, as well as for the potential $\psi(g(f(x), y))$ of the result $g(f(x), y)$. Here we multiply \cite{2} with $\phi''_i(z)$ for $i = 1 \ldots n$. 

We emphasise that this requires the possibility of deriving inequalities like (2), which only involve potentials, but no actual costs. That is, in the multivariate case we crucially employ a cost-free semantics to handle composition of functions. In a cost-free semantics the whose evaluation does not emit any costs.

### Logarithmic Amortised Costs

We can now explain at this high level the main ingredients of the proposed amortised resource analysis for logarithmic amortised costs, which also provides some intuition for the type system established in Section 5. Among other potential functions which we introduce later, we use (linear combinations of) functions of the form

$$p_{a_1, \ldots, a_n, b}(x_1, \ldots, x_n) = \log(a_1 x_1 + \cdots + a_n x_n + b),$$

where $a_1, \ldots, a_n, b \in \mathbb{N}$. We then have $p_{a_1, \ldots, a_n, b}(x_1+1, x_2, \ldots, x_n) = p_{a_1, \ldots, a_n, b+1}(x_1, \ldots, x_n)$, which constitutes the counterpart of the fact that shifts of polynomials are themselves polynomials. Similarly, $p_{a_0, a_1, \ldots, a_n, b}(x_1, x_1, x_2, \ldots, x_n) = p_{a_0+a_1, \ldots, a_n, b}(x_1, \ldots, x_n)$, which forms the basis of sharing. For composition we use the following reasoning. Suppose that

$$\phi_0(x) \geq c(x) + \phi_0(f(x)) \quad (4)$$

$$\log(a_i |x| + b_i) \geq \log(a'_i |f(x)| + b'_i) \quad \text{for all } 0 < i \leq n \quad (5)$$

$$\phi_0(y) + \sum_{i=1}^n \log(a'_i |y| + a''_i |z| + b'_i) \geq d(y, z) + \psi(g(y, z)), \quad (6)$$

where $|\cdot|$ are arbitrary nonnegative functions, and the potential functions $\phi, \psi$ are as before. Then we can conclude, arguing similarly as in the multivariate case, that the following inequality holds:

$$\phi_0(x) + \sum_{i=1}^n \log(a_i |x| + a''_i |z| + b_i) \geq c(x) + d(f(x), y) + \psi(g(f(x), y)).$$

Here we crucially use strict monotonicity of the logarithm function, in particular the fact that $\log(u) \geq \log(v)$ implies $\log(u + w) \geq \log(v + w)$ for $u, v, w \geq 1$, cf. Lemma 11. Again the potential of the arguments suffices to pay for the cost of computing $c(x)$, $d(f(x), y)$, respectively and covers in addition the potential of the result.

We emphasise that this requires the possibility of deriving inequalities like (2), which only involve potentials, but no actual costs. That is, in the multivariate case we crucially employ a cost-free semantics to handle composition of functions. In a cost-free semantics the whose evaluation does not emit any costs.

### Resource Functions

In this section, we detail the basic potential functions employed and clarify the definition of potentials used.

Only trees are assigned non-zero potential. This is not a severe restriction as potentials for basic datatypes would only become essential, if the construction of such types would emit actual costs. This is not the case in our context. Moreover, note that list can be conceived as trees of particular shape. The potential $\Phi(t)$ of a tree $t$ is given as a non-negative linear combination of basic functions, which essentially amount to “sums of logs”, cf. Schoenmakers 15. It suffices to specify the basic functions for the type of trees $T$. More precisely, the rank $p_\ast(t)$ of a tree is defined as follows:

$$p_\ast(\text{nil}) := 0$$

$$p_\ast([t, a, u]) := p_\ast(t) + \log(|t|) + \log(|u|) + p_\ast(u).$$
Logarithmic Amortised Complexity

Here \( \log'(n) := \log_2(\max\{n, 1\}) \), such that the (binary) logarithm function is defined for all numbers. This is merely a technicality, introduced to ease the presentation. Furthermore, recall that \( |t| \) denotes the number of leaves in tree \( t \). In the following, we will denote the modified logarithmic function, simply as \( \log \). The definition of “rank” is inspired by the definition of potential in \( [15][18] \), but subtly changed to suit it to our context.

**Definition 2.** The basic potential functions of \( T \) are either

\[
\begin{align*}
&= \lambda t. p_\ast(t), \\
&= p_{(a,b)} := \log(a \cdot |t| + b), \text{ where } a, b \text{ are numbers.}
\end{align*}
\]

The basic functions are denoted as \( BF \). Note that the constant function \( 1 \) is representable: \( 1 = \log(0 \cdot |t| + 2) \).

Following the recipe of the high-level description in Section 3, potentials or more generally resource functions become definable as linear combination of basic potential functions.

**Definition 3.** A resource function \( r: [T] \to \mathbb{R}_0^+ \) is a non-negative linear combination of basic potential functions, that is,

\[
r(t) := \sum_{i \in \mathbb{N}} q_i \cdot p_i(t),
\]

where \( p_i \in BF \). The set of resource functions is denoted as \( RF \).

We employ \( \ast \), natural numbers \( i \) and pairs of natural numbers \( (a, b)_{a,b \in \mathbb{N}} \) as indices of the employed basic potential functions. A resource annotation over \( T \), or simply annotation, is a sequence \( Q = [q_\ast] \cup [(q_{(a,b)})_{a,b \in \mathbb{N}}] \) with \( q_\ast, q_{(a,b)} \in \mathbb{Q}_0^+ \) with all but finitely many of the coefficients \( q_\ast, q_{(a,b)} \) equal to \( 0 \). It represents a (finite) linear combination of basic potential functions, that is, a resource function. The empty annotation, that is, the annotation where all coefficient are set to zero, is denoted as \( \emptyset \).

**Remark.** We use the convention that the sequence elements of resource annotations are denoted by the lower-case letter of the annotation, potentially with corresponding sub- or superscripts.

**Definition 4.** The potential of a tree \( t \) with respect to an annotation \( Q \), that is, \( Q = [q_\ast] \cup [(q_{(a,b)})_{a,b \in \mathbb{N}}] \), is defined as follows.

\[
\Phi(t|Q) := q_\ast \cdot p_\ast(t) + \sum_{a,b \in \mathbb{N}} q_{(a,b)} \cdot p_{(a,b)}(t),
\]

Recall that \( p_{(a,b)} = \log(a \cdot |t| + b) \) and that \( p_\ast \) is the rank function, defined above.

**Example 5.** Let \( t \) be a tree, then it’s potential could be defined as follows: \( p_\ast(t) + 3 \cdot \log(|t|) + 1 \). With respect to the above definition this potential becomes representable by setting \( q := 1, q_{1,0} := 3, q_{0,2} := 1 \). Conclusively \( \Phi(t|Q) = p_\ast(t) + 3 \cdot \log(|t|) + 1 \).

We emphasise that the linear combination defined above is not independent. Consider, for example \( \log(2|t| + 2) = \log(|t| + 1) + 1 \).

The potential of a sequence of trees \( t_1, \ldots, t_m \) is defined as the linear combination of \( p_\ast(t_i) \) and a straightforward extension of the basic potential functions \( p_{a,b} \) to \( m \) arguments, denoted as \( p_{(a_1, \ldots, a_m, b)} \). Here \( p_{(a_1, \ldots, a_m, b)}(t_1, \ldots, t_m) \) is defined as the logarithmic function \( \log(a_1 \cdot |t_1| + \cdots + a_m \cdot |t_m| + b) \), where \( a_1, \ldots, a_m, b \in \mathbb{N} \).

More precisely, we first generalise annotations to sequences of trees. An annotation for a sequence of length \( m \) is a sequence \( Q = [q_1, \ldots, q_m] \cup [(q_{(a_1, \ldots, a_m, b)})_{a_i \in \mathbb{N}}] \), again vanishing
almost everywhere. Note that an annotation of length 1 is simply an annotation, where the coefficient \( q_1 \) is set equal to the coefficient \( q_n \). Based on this, the potential of \( t_1, \ldots, t_m \) is
defined as follows.

**Definition 6.** Let \( t_1, \ldots, t_m \) be trees and let \( Q = [q_1, \ldots, q_m] \cup [(q_{a_1, \ldots, a_m, b})_{a_i \in \mathbb{N}}] \) be an annotation of length \( n \) as above. We define

\[
\Phi(t_1, \ldots, t_m|Q) := \sum_{i=1}^{m} q_i \cdot p_\ast(t_i) + \sum_{a_1, \ldots, a_m, b \in \mathbb{N}} q_{(a_1, \ldots, a_m, b)} \cdot p_{(a_1, \ldots, a_m, b)}(t_1, \ldots, t_m),
\]

where \( p_{(a_1, \ldots, a_m, b)}(t_1, \ldots, t_m) := \log(a_1 \cdot |t_1| + \cdots + a_m \cdot |t_m| + b) \). Note that \( \Phi(\emptyset|Q) = \sum_{b \in \mathbb{N}} q_b \log(b) \).

Let \( t \) be a tree. Note that the rank function \( p_\ast(t) \) amounts to the sum of the logarithms of the size of subtrees of \( t \). In particular if the tree \( t \) simplifies to a list of length \( n \), then \( p_\ast(t) = \sum_{i=1}^{n} \log(i) \). Moreover, as \( \sum_{i=1}^{n} \log(i) \in \Theta(n \log n) \), the above defined potential functions are sufficiently rich to express linear combinations of sub- and super-linear functions. For practical purposes it may be necessary to expand the class of potential functions further. Here, we emphasise that it is not difficult to see the basic potential functions \( p_{a_1, \ldots, a_m, b} \) can be generalised as to also incorporate linear dependencies on the size of arguments; this does not invalidate any of the results in this section.

Let \( \sigma \) denote a substitution, let \( \Gamma \) denote a typing context and let \( x_1 : T, \ldots, x_m : T \) denote all tree types in \( \Gamma \). A resource annotation for \( \Gamma \) or simply annotation is an annotation for the sequence of trees \( x_1 \sigma, \ldots, x_m \sigma \). We define the potential of \( \Phi(\Gamma|Q) \) with respect to \( \sigma \) as \( \Phi(\sigma; \Gamma|Q) := \Phi(x_1 \sigma, \ldots, x_m \sigma|Q) \).

**Definition 7.** An annotated signature \( \mathcal{F} \) is a mapping from functions \( f \) to sets of pairs consisting of the annotation type for the arguments of \( f \) \( A_1 \times \cdots \times A_n|Q \) and the annotation type \( A'|Q' \) for the result:

\[
\mathcal{F}(f) := \left\{ A_1 \times \cdots \times A_n|Q \to A'|Q' \right\} \text{ if } f \text{ takes } m \text{ trees as arguments, } Q \text{ is an annotation of length } m \text{ and } Q' \text{ a resource annotation}.
\]

Note that \( m \leq n \) by definition.

We confuse the signature and the annotated signature and denote the latter simply as \( \mathcal{F} \). Instead of \( A_1 \times \cdots \times A_n|Q \to A'|Q' \in \mathcal{F}(f) \), we typically write \( f : A_1 \times \cdots \times A_n|Q \to A'|Q' \).

As our analysis makes use of a cost-free semantics any function symbol is possibly equipped with a cost-free signature, independent of \( \mathcal{F} \). The cost-free signature is denoted as \( \mathcal{F}^\ast \).

**Example 8.** Consider the function \texttt{splay} : \( B \times T \to T \). The induced annotated signature is given as \( B \times T|Q \to T|Q' \), where \( Q := [q_*] \cup [(q_{a,b})_{a,b \in \mathbb{N}}] \) and \( Q' := [q'_{*}] \cup [(q'_{a,b})_{a,b \in \mathbb{N}}] \). The logarithmic amortised cost of splaying, is then suitably expressed through the following setting: \( q_* := 1, q_{(1,0)} = 3, q_{(0,2)} = 1, q'_{*} := 1 \). All other coefficients are zero.

This amounts to a potential of the arguments \( p_\ast(t) + 3 \log(|t|) + 1 \), while for the result we consider only its rank. The correctness of the induced logarithmic amortised costs for splaying is verified in Section 6.

Suppose \( \Phi(t_1, \ldots, t_n, u_1, u_2|Q) \) denotes an annotated sequence of length \( n + 2 \). Suppose \( u_1 = u_2 \) and we want to share the values \( u_i \), that is, the corresponding function arguments appear multiple times in the body of the function definition. Then we make use of the operator \( \gamma|Q \) that adapts the potential suitably. The operator is also called sharing operator.
Lemma 9. Let $t_1, \ldots, t_n, t, u_1, u_2$ denote a sequence of trees of length $n + 2$ with annotation $Q$. Then there exists a resource annotation $\gamma(Q)$ such that $\Phi(t_1, \ldots, t_n, u_1, u_2|Q) = \Phi(t_1, \ldots, t_n, u|\gamma(Q))$, if $u_1 = u_2 = u$.

Proof. Wlog. we assume $n = 0$. Thus, let $Q = \{q_1, q_2\} \cup \{(q_{(a_1, a_2, b)}, a, b \in \mathbb{N}\}$. By definition

$$\Phi(u_1, u_2|Q) = q_1 \cdot p_*(u_1) + q_2 \cdot p_*(u_2) + \sum_{a_1, a_2, b \in \mathbb{N}} q_{(a_1, a_2, b)} \cdot p_{(a_1, a_2, b)}(u_1, u_2),$$

where $p_{(a_1, a_2, b)}(u_1, u_2) = \log(a_1 \cdot |u_1| + a_2 \cdot |u_2| + b)$. By assumption $u = u_1 = u_2$. Thus, we obtain

$$\Phi(u, u|Q) = q_1 \cdot p_*(u) + q_2 \cdot p_*(u) + \sum_{a_1, a_2, b \in \mathbb{N}} q_{(a_1, a_2, b)} \cdot p_{(a_1, a_2, b)}(u, u)$$

$$= (q_1 + q_2)p_*(u) + \sum_{a_1 + a_2, b \in \mathbb{N}} q_{(a_1 + a_2, b)} \cdot p_{(a_1 + a_2, b)}(u)$$

$$= \Phi(u|\gamma(Q)).$$

for suitable defined annotation $\gamma(Q)$.

Remark. We observe that the definability of the sharing annotation $\gamma(Q)$ is based on the fact that the basic potential functions $p_{a_1, \ldots, a_m, b}$ have been carefully chosen so that

$$p_{a_0; a_1, \ldots, a_m, b}(x_1, x_1, \ldots, x_m) = p_{a_0 + a_1, \ldots, a_m, b}(x_1, x_1, \ldots, x_m),$$

holds, cf. Section 3.

Remark. We observe that the proof-theoretic analogue of the sharing operation constitutes in a contraction rule, if the type system is conceived as a proof system.

Let $Q = \{|q_*|\} \cup \{(q_{(a, b)}, a, b \in \mathbb{N}\}$ be an annotation and let $K \in Q_0^\dagger$. Then we define $Q' := Q + K$ as follows: $Q' = \{|q_*|\} \cup \{(q_{(a, b)'}, a, b \in \mathbb{N}\}$, where $q_{(0, 2)}' := q_{(0, 2)} + K$ and for all $(a, b) \neq (0, 2)$ $q_{(a, b)'} := q_{(a, b)}$. By definition the annotation coefficient $q_{(0, 2)'}$ is the coefficient of the basic potential function $p_{(0, 2)}(t) = \log(0t) + 2 = 1$, so the annotation $Q + K$, adds cost $K$ to the potential induced by $Q$.

Due to the involved form of the basic function underlying the definition of potential, cf. Definition 3 we cannot simply define weakening of potentials through the (pointwise) comparison of annotations. This is in contrast to results on resource analysis for constant amortised costs. Instead we compare potentials symbolically by fixing the shape of the considered logarithmic functions and perform coefficient comparisons, akin to similar techniques in the synthesis of polynomial interpretations 2. In addition we use basic laws of the log functions as well as properties of the size function.

Let $\Gamma$ denote a type context containing the type declarations $x_1 : T, \ldots, x_m : T$ and let $Q$ be an annotation of length $n$. The the symbolic potential, denoted as $\Phi(\Gamma|Q)$, is defined as:

$$\Phi(x_1, \ldots, x_m|Q) := \sum_{i=1}^{m} q_i \cdot p_*(x_i) + \sum_{a_1, \ldots, a_m, b \in \mathbb{N}} q_{(a_1, \ldots, a_m, b)} \cdot p_{(a_1, \ldots, a_m, b)}(x_1, \ldots, x_m),$$

where $p_{(a_1, \ldots, a_m, b)}(x_1, \ldots, x_m) = \log(a_1 \cdot |x_1| + \cdots + a_m \cdot |x_m| + b)$.

In order to automate the verification of the constraint $\Phi(\Gamma|Q) \geq \Phi(\Gamma|Q')$, we can rely on a suitably defined heuristics based on the following simplification steps:

1. simplifications, like e.g. $p_*(\langle u, b, v \rangle) = p_*(u) + p_*(v) + \log(|u|) + \log(|v|)$;
2. monotonicity of log;
3. simply estimations of the logarithm functions like the next lemma; and
4. properties of the size function.

Lemma 10. Let \( x, y \geq 1 \). Then \( 2 + \log(x) + \log(y) \leq 2\log(x + y) \).

Proof. We observe
\[
(x + y)^2 - 4xy = (x - y)^2 \geq 0 .
\]
Hence \( (x + y)^2 \geq 4xy \) and from the monotonicity of log we conclude \( \log(xy) \leq \log(\frac{(x+y)^2}{4}) \).

By elementary laws of log we obtain:
\[
\log(\frac{x+y}{2}) = \log \left( \frac{(x + y)^2}{4} \right) = 2\log(x + y) - 2 ,
\]
from which the lemma follows as \( \log(xy) = \log(x) + \log(y) \).

We leave the simple proof to the reader. A variant of this fact has already been observed by Okasaki, cf. [16]. The above heuristic is automatable, employing off-the-shelf SMT solvers, such that the required simplification rules are incorporated as (user-defined) axioms. However, this is not very efficient. In Section 7 we sketch an alternative path towards automation.

5 Logarithmic Amortised Resource Analysis

In this section, we present the central contribution of this work. We delineate a novel type system incorporating a potential-based amortised resource analysis capable of expressing logarithmic amortised costs. Soundness of the approach is established in Theorem 13.

The next auxiliary lemma is a direct consequence of the strict monotonicity of log. Note, that the assumption that \( a, b, c \) are strictly greater than is zero is necessary, even in the light of our use of a “modified” logarithm function, see page 8.

Lemma 11. Let \( u, v, w \geq 1 \). If \( \log(u) \leq \log(v) \), then \( \log(u + w) \leq \log(v + w) \).

From the lemma we conclude for coefficients \( q_i \) and positive rational number \( q_1, \ldots, q_n, b \) and \( c \), that we have:
\[
\sum_i q_i \cdot \log(a_i) \geq \log(b) \text{ implies } \sum_i q_i \cdot \log(a_i + c) \geq \log(b + c) .
\]

The above inequality is employed in the correct assessment of the transfer of potential in the case of function composition, see Figure 5 as well as the high-level description provided in Section 3.

Our potential-based amortised resource analysis is coached in a type system, which is given in Figure 5. If the type judgement \( \Gamma|Q \vdash e : A|Q' \) is derivable, then the cost of execution of the expression \( e \) is bound from above by the difference between the potential \( \Phi(\sigma; \Gamma|Q) \) before the execution and the potential \( \Phi(v|Q') \) of the value \( v \) obtained through the evaluation of the expression \( e \). The typing system makes use of a cost-free semantics, which does not attribute any costs to the calculation. I.e. the (app) is changed as no cost is emitted. The cost-free typing judgement is denoted as \( \Gamma|Q \vdash_{cf} e : A|Q' \).

Remark. Principally the type system can be parametrised in the resource metric (see eg. [4]). However, we focus on worst-case runtime complexity, symbolically measured through the number of rule applications.
We consider the typing rules in turn; recall the convention that sequence elements of annotations are denoted by the lower-case letter of the annotation. The variable rule (\texttt{var}) types a variable of unspecified type $A$. As no actual costs are required the annotation is unchanged. Similarly no resources are lost through the use of control operators. Conclusively, the definition of the rules (\texttt{cmp}) and (\texttt{ite}) is straightforward.

As exemplary constructor rules, we have rule (\texttt{nil}) for the empty tree and rule (\texttt{node}) for the node constructor. Both rules define suitable constraints on the resource annotations to guarantee that the potential of the values is correctly represented.

The application rule (\texttt{app}) represents the application of a rule in $P$; the required annotations for the typing context and the result can be directly read off from the annotated signature. Each application emits actual cost 1, which is indicated in the addition of 1 to the annotation $Q$.

In the pattern matching rule (\texttt{match}) the potential freed through the destruction of the tree construction is added to the annotation $R$, which is used in the right premise of the rule. Note that the length of the annotation $R$ is $m + 2$, where $m$ equals the number of tree types in the type context $\Gamma$.

The constraints expressed in the typing rule let, guarantee that the potential provided through annotation $Q$ is distributed among the call to $e_1$ and $e_2$. This typing rule takes care of function composition. Due to the sharing rule, we can assume wlog. that each variable in $e_1$ and $e_2$ occurs at most once. The numbers $m$, $k$, respectively, denote the number of tree types in $\Gamma$, $\Delta$. This rule necessarily employs the cost-free semantics. The premise $\Gamma[P \vdash c_1 e_1 : A][P_B]$ ($\vec{b} \neq \vec{0}$) expresses that for all non-zero vectors $\vec{b}$, the potentials $\Phi(\Gamma[P_B])$ suffices to cover the potential $\Phi(A[P_B])$, if not extra costs are emitted. Intuitively this represents the cost-free constraint $\mathbf{3}$ emphasised in Section 3.

Finally, the type system makes use of structural rules, like the sharing rule (\texttt{share}) and the weakening rules (\texttt{w: var}) and (\texttt{w}). The sharing rule employs the sharing operator, implicitly defined in Lemma 9. Note that the variables $x, y$ introduced in the assumption of the typing rule are fresh variables, that do not occur in $\Gamma$. The weakening rules embody changes in the potential of the type context of expressions considered. Weakening employs the symbolic potential expressions, introduced in Section 4.

\begin{definition}
A program $P$ is called well-typed if for any rule $f(x_1, \ldots, x_k) = e \in P$ and any annotated signature $A_1 \times \cdots \times A_k|Q \rightarrow A'|Q' \in F(f)$, we have $x_1 : A_1, \ldots, x_k : A_k|Q \vdash e : A'|Q'$. A program $P$ is called cost-free well-typed, if the cost-free typing relation is employed.
\end{definition}

We obtain the following soundness result.

\begin{theorem}[Soundness Theorem]
Let $P$ be well-typed and let $\sigma$ be a substitution. Suppose $\Gamma|Q \vdash e : A|Q'$ and $\sigma \vdash e \Rightarrow v$. Then $\Phi(\sigma; \Gamma|Q) - \Phi(v|Q') \geq m$.
\end{theorem}

\begin{proof}
The proof embodies the high-level description given in Section 3. It proceeds by main induction on $\Pi : \sigma \vdash e \Rightarrow v$ and by side induction on $\Sigma : \Gamma|Q \vdash e : A|Q'$. We consider only a few cases of interest. For example, for a case not covered: the variable rule (\texttt{var}) types a variable of unspecified type $A$. As no actual costs a required the annotation is unchanged and the theorem follows trivially.

\textbf{Case}. $\Pi$ derives $\sigma \vdash 0 \text{ nil} \Rightarrow \text{nil}$. Then $\Sigma$ consists of a single application of the rule (\texttt{nil}):

$$
\frac{q_c = \sum_{a+b=c} q'_{a,b}}{\varnothing|Q \vdash \text{nil} : T|Q' (\text{nil})}.
$$

\end{proof}
\[
q_e = \sum_{a,b} q_{a,b} \cdot \varepsilon_{a,b} (\text{nil})
\]
\[
\Gamma \vdash e : C \vdash Q \quad r_i = q_i \quad r_{\overline{a},b} = q_{\overline{a},0,b} (w : \text{var})
\]
\[
q_1 = q_2 = q' \quad q_{(1,0,0)} = q_{(0,1,0)} = q' \quad q_{a,a,b} = q'_{a,b} (\text{node})
\]
\[
\text{cmp } a \text{ comparison operator }
\]
\[
x_1 : T, x_2 : B, x_3 : T \vdash x_1 \text{ cmp } x_2 : B \vdash Q \quad (\text{cmp})
\]
\[
\Gamma \vdash e_1 : A \vdash Q' \quad \Gamma \vdash e_2 : A \vdash Q' \quad \Phi(\Gamma) \subseteq \Phi(\Gamma)
\]
\[
\Delta, x : A \vdash e_1 : C \vdash Q' \quad (\text{let})
\]
\[
\Phi(\Gamma) \subseteq \Phi(\Gamma)
\]
\[
\Phi(\Gamma) \subseteq \Phi(\Gamma)
\]
\[
x : A \vdash x : A (\text{var})
\]
\[
A_1 \times \cdots \times A_n : \mathbb{Q} \vdash \phi \quad (\text{app})
\]

To ease notation, we set \(a := a_1, \ldots, a_m, b := b_1, \ldots, b_k, i \in \{1, \ldots, m\}, j \in \{1, \ldots, k\}\) and \(a, b, c \in \mathbb{Q}_0^+\) to simplify notation. Sequence elements of annotations, which are not constraint are set to zero.

Figure 5 Type System for Logarithmic Amortised Resource Analysis

By assumption \(Q = [q_e]_{e \in \mathbb{N}}\) is an annotation for the empty sequence of trees. On the other hand \(Q' = [q'_{a,b}]_{a,b \in \mathbb{N}}\) is an annotation of length 1. Thus we obtain:

\[
\Phi(\sigma|Q) = \sum_{a} q_e \cdot \log(c) + \sum_{a,b} q'_{a,b} \cdot \log(a + b) + p_{\text{nil}} + \sum_{a,b} q'_{a,b} p_{a,b} = \Phi(\text{nil}|Q').
\]

Case. Suppose the last rule in II has the following from:

\[
x_1 \sigma = u \quad x_2 \sigma = b \quad x_3 \sigma = v
\]
\[
\sigma \vdash x_1, x_2, x_3 \Rightarrow \langle u, b, v \rangle.
\]

Wlog. \(\Xi\) consists of a single application of the rule node:

\[
q_1 = q_2 = q' \quad q_{(1,0,0)} = q_{(0,1,0)} = q' \quad q_{a,a,b} = q'_{a,b} (\text{node})
\]

By definition, we have \(Q = \{q_1, q_2\} \cup ([q_{a,a,b}]_{a,b \in \mathbb{N}}\) and \(Q' = [q'] \cup [q'_{a,b}]_{a,b \in \mathbb{N}}\). We set \(\Gamma := x_1 : T, x_2 : B, x_3 : T\) and \(\langle x_1, x_2, x_3 \rangle \sigma := \langle u, b, v \rangle\). Thus \(\Phi(\sigma; \Gamma|Q) = \Phi(u, v|Q)\) and
we obtain:

$$
\Phi(u,v|Q) = q_1 \cdot p_*(u) + q_2 \cdot p_*(v) + \sum_{a_1,a_2,b} q(a_1,a_2,b) \cdot \log(a_1 \cdot |u| + a_2 \cdot |v| + b)
$$

$$
\geq q'_1 \cdot p_*(u) + q'_2 \cdot p_*(v) + q(1,0,0) \cdot \log(|u|) + q(0,1,0) \cdot \log(|v|) +
+ \sum_{a,b} q(a,b) \cdot \log(a \cdot |u| + a \cdot |v| + b)
$$

$$
= q'_1 \cdot p_*(u) + p_*(v) + \log(|u|) + \log(|v|) +
+ \sum_{a,b} q'_1 \cdot p(a,b)(u,b,v) + \sum_{a,b} q(a,b) \cdot p(a,b) (u,b,v) = \Phi((u,b,v)|Q').
$$

**Case.** Consider the **match** rule, that is, \( \Pi \) ends as follows:

$$
\sigma \xrightarrow{\text{match } x \text{ with } \{ \text{nil} \mapsto e_1 \}} e_2 \Rightarrow v.
$$

Wlog. we may assume that \( \Xi \) ends with the related application of the (match):

$$
\begin{align*}
\Gamma &\vdash e_1 : A|Q' \\
\Gamma, x: T &\vdash \text{match } x \text{ with } \{ \text{nil} \mapsto e_1 \} (x_1, x_2, x_3) \Rightarrow e_2 : A|Q' \quad \text{(match)}.
\end{align*}
$$

We assume the annotations \( P, Q, R \), are of length \( m, m+1 \) and \( m+3 \), respectively, while \( Q' \) is of length 1. We write \( t := t_1, \ldots, t_m \) for the substitution instances of the variables in \( \Gamma \) and \( t := x\sigma = (u,b,v) \), where the latter equality follows from the assumption on \( \Pi \). By definition and the constraints given in the rule, we obtain:

$$
\Phi(\sigma; \sigma[\Gamma, x: T])Q = \sum_i q_i p_*(t_i) + q_{m+1} p_*(t) + \sum_{\tilde{a},a,c} \log(\tilde{a}[\tilde{t}] + c)
$$

$$
= \sum_i q_i p_*(t_i) + q_{m+1} p_*(\langle u,b,v \rangle) + \sum_{\tilde{a},a,c} q(\tilde{a},a,c)(t_i, t)
$$

$$
= \sum_i q_i p_*(t_i) + q_{m+1} (p_*(u) + \log(|u|) + \log(|v|) + p_*(v)) +
+ \sum_{\tilde{a},a,c} \log(\tilde{a}[\tilde{t}] + a[|u| + |v|] + c)
$$

$$
= \Phi(\sigma; \sigma[\Gamma, x_1: T, x_2: B, x_3: T]) R,
$$

where, we shortly write \( \tilde{a}[\tilde{t}] (\tilde{b}[\tilde{u}]) \) to denote componentwise multiplication.

Thus \( \Phi(\sigma; \sigma[\Gamma, x: T])Q = \Phi(\sigma; \sigma[\Gamma, x_1: T, x_2: B, x_3: T]) R \) and the theorem follows by an application of MIH.

**Case.** Consider the **let** rule, that is, \( \Pi \) ends in the following rule:

$$
\sigma \xrightarrow{e_1} v' \quad \sigma[x \mapsto v'] \xrightarrow{e_2} v.
$$

$$
\sigma \xrightarrow{\text{let } x = e_1 \text{ in } e_2 \Rightarrow v}.
$$
where \( m = m_1 + m_2 \). Wlog. \( \Xi \) ends in the following application of the let-rule.

\[
\Gamma \vdash P: T \mid P' \quad \Gamma \vdash P_\beta \xrightarrow{cf} e_1 : T \mid P'_\beta \quad \Delta, x : T \mid R \vdash e_2 : C \mid Q'
\]

\[
\Gamma, \Delta \mid Q \vdash \text{let } x = e_1 \text{ in } e_2 : C \mid Q'.
\]

Recall that \( \vec{a} = a_1, \ldots, a_m, \vec{b} = b_1, \ldots, b_k, i \in \{1, \ldots, m\}, j \in \{1, \ldots, k\} \) and \( a, c \in Q^+_T \). Further, the annotations \( Q, P, R \) are annotation of length \( m + k, m \) and \( k \), respectively, while \( Q', P', R' \) are of length 1. For each sequence \( b_1, \ldots, b_k \neq \vec{0}, P_\beta \) denotes an annotation of length \( m \). We tacitly assume that \( \vec{a} \neq 0 \), as well as \( \vec{b} \neq 0 \). Furthermore, recall the convention that the sequence elements of annotations are denoted by the lower-case letter of the annotation, potentially with corresponding sub- or superscripts.

By definition and due to the constraints expressed in the typing rule, we have

\[
\Phi(\sigma; \Gamma, \Delta; Q) = \sum_i q_i p_\sigma(t_i) + \sum_j q_j p_\sigma(u_j) + \sum_{\vec{a}, \vec{b}, c} q_{(\vec{a}, \vec{b}, c)} \log(\vec{a} [\vec{t}] + \vec{b} [\vec{u}] + c)
\]

\[
\Phi(\sigma; \Gamma; P) = \sum_i q_i p_\sigma(t_i) + \sum_{\vec{a}, c} q_{(\vec{a}, \vec{b}, c)} \log(\vec{a} [\vec{t}] + c)
\]

\[
\Phi(\sigma'; P') = r_{k+1} p_\sigma(\sigma') + \sum_{a, c} r_{(\vec{b}, a, c)} \log(a [\vec{v}] + c)
\]

\[
\Phi(\sigma; \Gamma; P'_\beta) = \sum_{\vec{a}, c} q_{(\vec{a}, \vec{b}, c)} \log(\vec{a} [\vec{t}] + c)
\]

\[
\Phi(\sigma'; P'_{\vec{b}}) = \sum_{\vec{a}, c} r_{(\vec{b}, a, c)} \log(a [\vec{v}] + c)
\]

\[
\Phi(\sigma; \Delta, x : T; R) = \sum_j q_j p_\sigma(u_j) + r_{k+1} p_\sigma(\sigma') + \sum_{\vec{b}, a, c} r_{(\vec{b}, a, c)} \log(\vec{b} [\vec{u}] + a [\vec{v}] + c),
\]

where we set \( \vec{t} := t_1, \ldots, t_m \) and \( \vec{u} := u_1, \ldots, u_k \), denoting the substitution instances of the variables in \( \Gamma, \Delta \), respectively.

By main induction hypothesis, we conclude that \( \Phi(\sigma; \Gamma; P) - \Phi(\sigma'; P') \geq m_1 \), while for all \( \vec{b} \neq \vec{0} \), \( \Phi(\sigma; \Gamma; P'_\beta) \geq \Phi(\sigma'; P'_{\vec{b}}) \). A second application of MH yields that \( \Phi(\sigma; \Delta, x : T; R) - \Phi(\sigma'; Q') \geq m_2 \). Due to Lemma \ref{lem:generalisation} we can combine these two results and conclude the theorem.

\begin{remark}
As remarked in Section \ref{sec:generalisation} the basic resource functions can be generalised to additionally represent linear functions in the size of the arguments. The above soundness theorem is not affected by this generalisation.
\end{remark}

\section{Analysis}

In this section, we exemplify the use of the type system presented in the last section on the function \texttt{splay}, cf. Figure \ref{fig:tt}. Our amortised analysis of splaying yields that the amortised cost of \texttt{splay} a \( t \) is bound by \( 1 + 3 \log(|t|) \), where the actual cost count the number of calls to \texttt{splay}, cf. \cite{15,18,19}. To verify this declaration, we derive

\[
a : B, t : T | Q \vdash e : T | Q', \tag{7}
\]

where the expression \( e \) is the definition of \texttt{splay} given in Figure \ref{fig:tt}. We restrict to the zig-zig case: \( t = (\langle b, b, br \rangle, c, cr) \) together with the recursive call \texttt{splay} a \( bl = (al, a', ar) \) and
\[ f : A_1 \times \cdots \times A_n |Q|Q' \]
\[ a : B, bl : T |Q + 1 \vdash \text{splay } a \text{ bl : } T |Q' \]
\[ \Delta, a : \Gamma, a' : B, ar : T |Q_4 + t' : T |Q' \]
\[ \Delta, x : T |Q_4 \vdash \text{match } x \text{ with } |(al, a', ar) \rightarrow t' : T |Q' \]
\[ \Gamma, cr : T, bl : T |Q_4 \vdash e_4 : T |Q' \]
\[ \Gamma, cr : T, bl : T |Q_4 \vdash e_4 : T |Q' \]
\[ a : B, b : B, cl : T, cr : T |Q_4 \vdash \text{match } cl \text{ with } |\langle cl, c \rangle \rightarrow Q_4 \]
\[ a : B, cl : T, c : B, cr : T |Q_4 \vdash \text{if } a = c \text{ then } |\langle cl, c, cr \rangle \rightarrow e_3 : T |Q' \]
\[ a : B, t : T |Q \vdash \text{match } t \text{ with } |\langle \text{nil} \rightarrow \text{nil}, (al, a', ar) \rightarrow t' \rightarrow e_1 : T |Q' \]

\( a < b < c \). Thus \text{splay } a \text{ t} yields \( (al, a', \langle ar, b, \langle br, c, cr \rangle \rangle) =: t' \). Recall that \( a \) need not occur in \( t \), in this case the last element \( a' \) before a leaf was found is rotated to the root.

Let \( e_1 \) denote the subexpression of the definition of splaying, starting in program line 4. On the other hand let \( e_2 \) denote the subexpression defined from line 5 to 15 and let \( e_3 \) denote the program code within \( e_2 \) starting in line 7. Finally the expression in lines 11 and 12, expands to the following, if we remove part of the the syntactic sugar.

\[ e_4 := \text{let } x = \text{splay } a \text{ bl in match } x \text{ with } |\langle \text{nil} \rightarrow \text{nil}, (al, a', ar) \rightarrow t' \rightarrow e_1 : T |Q' \]

Figure 6 shows a simplified derivation of \( \Phi \), where we have focused only on a particular path in the derivation tree, suited to the assumption on \( t \). Omission of premises is indicated by double lines in the inference step. We abbreviate \( \Gamma := a : B, b : B, c : C, \Delta := b : B, c : C, cr : T, br : T \). Here, we use the following annotations, induced by constraints in the type system, cf. Figure 5.

\begin{align*}
Q & : q = 1, q_{1.0} = 3, q_{0.2} = 1, \\
Q' & : q' = 1, \\
Q_1 & : q_1^1 = q_1^2 = q = 1, q_{1,1.0} = q_{1,0.0} = 1, q_{1,1.0} = q_{1,0.0} = 1, q_{0.2} = 1, \\
Q_2 & : q_2^2 = q_2^3 = q = 1, q_{1,1.0} = q_{1,0.0} = 1, q_{0.2} = 3, q_{0,1.1} = q_{1,0.0} = 1, \\
Q_3 & : q_3^3 = q_3^4 = q = 1, q_{0,0.2} = 2, q_{0,1.0} = 3, q_{0,0.0} = 1, q_{1,0.0} = q_{0,0.0} = 1, \\
Q_4 & : q_4^4 = q = 1, q_{0,0.0} = 1, q_{0,0.0} = q_{0,0.0} = q_{1,0.0} = 1.
\end{align*}

We emphasise that a simple symbolic calculation, following the heuristics outlined on page 11 suffices to conclude the following inequality, employed in the indicated weakening step in Figure 6.

\[ \Phi(\Gamma, cr : T, bl : T, br : T |Q_4) \geq \Phi(\Gamma, cr : T, bl : T, br : T |Q_4) \]

We verify the correctness of the weakening step through a direct comparision. Let \( \sigma \) be a
substitution. Then, we have

\[
\Phi(\sigma; cr: T, bl: T, br: T|Q_2) = 1 + p_\ast(cr) + p_\ast(bl) + p_\ast(br) + \\
+ 3\log(|cr|) + 3\log(|bl|) + 3\log(|br|) + \\
+ \log(|bl| + |br|) + \log(|cr|) + \log(|bl|) + \log(|br|)
\]

\[
= 1 + p_\ast(cr) + p_\ast(bl) + p_\ast(br) + 2\log(|t|) + \log(|t|) + \\
+ \log(|bl| + |br|) + \log(|cr|) + \log(|bl|) + \log(|br|)
\]

\[
\geq 1 + p_\ast(cr) + p_\ast(bl) + p_\ast(br) + \log(|bl|) + \log(|br|) + \\
+ \log(|bl| + |br|) + \log(|cr|) + \log(|bl|) + \\
+ \log(|br| + |cr|) + 2 + \log(|bl| + |br| + |cr|)
\]

\[
\geq p_\ast(bl) + 1 + 3\log(|bl|) + p_\ast(cr) + p_\ast(br) + \log(|br|) + \\
+ \log(|cr|) + \log(|br| + |cr|) + \\
+ \log(|bl| + |br| + |cr|) + 1 = \Phi(\sigma; cr: T, bl: T, br: T|Q_3).
\]

Note that we have used Lemma 10 in the third line to conclude

\[
2\log(|t|) \geq \log(|bl|) + \log(|br| + |cr|) + 2,
\]

as we have \(|t| = |\langle bl, br, c, cr \rangle| = |bl| + |br| + |cr|\). Furthermore, we have only used monotonicity of log and formal simplifications. In particular all necessary steps are covered in the simple heurstics introduced in Section 5.

Furthermore, the (let)-rule is applicable with respect to the following annotation \(Q_4:\)

\[
Q_4: q_1^4 = q_2^4 = q_3^4 = 1, q_{0.0.0.0} = q_{0.1.0.0} = q_{1.1.0.0} = q_{1.1.1.0} = 1.
\]

It suffices to verify the cost-free typing relation

\[
a : B, bl : T | P_b \vdash_{\text{cf}} \textbf{splay a bl : T} | P_{b}'
\]

(8)

where \(\vec{b} = (b_1, b_2) \neq \vec{0}\). Note that the condition 8 has been omitted from Figure 6 to allow for a condensed presentation. Informally speaking 8 requires that in a cost-free computation the potential is preserved. The interesting sub-case is the case for \(\vec{b} = (1, 1)\), governed by the annotations \(P_{1,1}\) and \(P'_{1,1}\), respectively. The corresponding potentials are \(\Phi(\sigma; a : B, bl : T|P_{1,1}) = \log(|bl|)\) and \(\Phi(\sigma; a : B, bl : T|P'_{1,1}) = \log(|\langle al, a', ar \rangle|)\). As \(|bl| = |\langle al, a', ar \rangle|\) by definition of splay, the potential remains unchanged as required.

Finally, one further application of the match-rule yields the desired derivation for suitable \(Q_5\).

7 Towards Automatisation

In this short section, we argue that the above introduced potential-based amortised resource analysis is automatable. As emphasised in Section 5 the principal approach to automatisation is to set up annotations with indeterminate coefficients and solve for them so as to automatically infer costs. The corresponding constraints are obtained through a syntax-directed type inference. In the context of the type system presented in Figure 5 an obvious challenge is the requirement to compare potentials symbolically (compare Section 4) rather than compare annotations directly. More generally, the presence of logarithmic basic functions necessitates the embodiment of nonlinear arithmetic.
A straightforward approach for automation would exploit recent advances in SMT solving. For this one can suitably incorporating the required nonlinear arithmetic as axioms to an off-the-shelf solver and pass the constraints to the solver. We have experimented with this approach, but the approach has turned out to be too inefficient. In particular, as we cannot enforce linear constraints.

However, a more refined and efficient approach which targets linear constraints is achievable as follows. All logarithmic terms, that is, terms of the form \( \log(\cdot) \) are replaced by new variables, focusing on finitely many. For the latter we exploit the condition that in resource annotation only finitely many coefficients are non-zero. With the example in the previous section, \( \log(|bl| + |br|) \), \( \log(|bl|) \) are replaced by the fresh (constraint) variables \( x, y \), respectively. Thus laws of the monotonicity function, like e.g. monotonicity of \( \log \), as well as properties like Lemma 10 can be expressed as inequalities over the introduced unknowns. E.g., the inequality \( x \geq y \) represents the axiom of monotonicity \( \log(|bl| + |br|) \geq \log(|bl|) \). All such obtained inequality are collected as “expert knowledge”. We can express the required expert knowledge succinctly in the form of a system of inequalities as \( Ax \leq b \), where \( A \) denotes a matrix with as many rows as we have expert knowledge, \( \vec{b} \) a column vector and \( \vec{x} \) the column vector of unknowns of suitable length. With the help of the variables in \( \vec{x} \), we construct linear combinations based on indeterminate coefficients giving rise to the potential functions fulfilling the constraints gathered from type inference. More precisely, we have to solve the implication

\[
\forall \vec{x} \quad A \vec{x} \leq \vec{b} \Rightarrow CY \vec{x} \leq \vec{d}.
\]

Here \( C \) is the matrix of coefficients and the matrix \( Y \) represents the required linear combinations.

In order to automate the derivation of (9), we exploit the following variant of Farkas’ Lemma.

**Lemma 14.** Suppose \( A \vec{x} \leq \vec{b} \) is solvable. Then the following assertions are equivalent.

\[
\forall \vec{x} \quad A \vec{x} \leq \vec{b} \Rightarrow \vec{d}^T \vec{x} \leq \lambda \tag{10}
\]

\[
\exists \vec{f} \quad f \geq 0 \land \vec{a}^T \leq \vec{f}^T A \land \vec{f}^T \vec{b} \leq \lambda \tag{11}
\]

**Proof.** It is easy to see that from \(11\), we obtain \(10\). Assume \(11\). Assume further that \( A \vec{x} \leq \vec{b} \) for some column vector \( \vec{x} \). Then we have

\[
\vec{a}^T \vec{x} \leq \vec{f}^T A \vec{x} \leq \vec{f}^T \vec{b} \leq \lambda.
\]

Note that for this direction the assumption that \( A \vec{x} \leq \vec{b} \) is solvable is not required.

With respect to the opposite direction, we assume \(10\). By assumption, the inequality \( A \vec{x} \leq \vec{b} \) is solvable. Hence, maximisation of \( \vec{a}^T \vec{x} \) under the side condition \( A \vec{x} \leq \vec{b} \) is feasible. Let \( w \) denote the maximal value. Due to \(10\), we have \( w \leq \lambda \).

Now, consider the dual asymmetric linear program to minimise \( \vec{g}^T \vec{b} \) under side condition \( \vec{g}^T A = \vec{a}^T \) and \( \vec{g} \geq 0 \). Due to the Dualisation Theorem, the dual problem is also solvable with the same solution

\[
\vec{g}^T \vec{b} = \vec{a}^T \vec{x} = w.
\]

We fix a vector \( \vec{f} \) which attains the optimal value \( w \), such that \( \vec{f}^T A = \vec{a}^T \) and \( \vec{f} \geq 0 \) such that \( \vec{f}^T \vec{b} = w \leq \lambda \). This yields \(11\).
Generalising Lemma 14, we obtain the following equivalence, which allows an efficient encoding of $\text{9}$.

\[ \forall \vec{x} \ A\vec{x} \leq \vec{b} \Rightarrow U\vec{x} \leq \vec{v} \iff \exists F \geq 0 \land U \leq FA \land F\vec{b} \leq \vec{v}. \]

As in the lemma, the equivalence requires solvability of the system $A\vec{x} \leq \vec{b}$. Note that the system expresses given domain knowledge and simple facts like Lemma 10, whose solvability is given a priori. We emphasise that the existential statement requires linear constraints only.

8 Conclusion

We have presented a novel amortised resource analysis based on the potential method. The method is rendered in a type system, so that resource analysis amounts to a constraint satisfaction problem, induced by type inference. The novelty of our contribution is that this is the first automatable approach to logarithmic amortised complexity. In particular, we show how the precise logarithmic complexity of splaying, a central operation of Sleator and Tarjan’s splay trees can be analysed in our system. Furthermore, we provide a suitable Ansatz to automatically infer logarithmic bounds on the runtime complexity.

In Memorium.

With deep sorrow, I report that Martin had a fatal hiking accident during the preparation of this work. He passed away in January, 2018. I’ve tried my best to finalise our common conceptions and ideas, any mistakes or other defects introduced are of course my responsibility. His work was revolutionary in a vast amount of fields and it will continue to inspire future researchers; like he inspired me.

References

1 M. Avanzini, G. Moser, and M. Schaper. TcT: Tyrolean Complexity Tool. In Proc. 22nd TACAS, volume 9636 of LNCS, pages 407–423, 2016. doi:10.1007/978-3-662-49674-9_24.
2 E. Contejean, C. Marché, A.-P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. JAR, 34(4):325–363, 2005.
3 J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. In Proc. 38th POPL, pages 357–370. ACM, 2011.
4 J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. TOPLAS, 34(3):14, 2012.
5 J. Hoffmann and M. Hofmann. Amortized resource analysis with polymorphic recursion and partial big-step operational semantics. In Proc. 8th APLAS, volume 6461 of LNCS, pages 172–187, 2010.
6 J. Hoffmann and M. Hofmann. Amortized resource analysis with polynomial potential. In Proc. 19th ESOP, volume 6012 of LNCS, pages 287–306, 2010.
7 J. Hoffmann and Z. Shao. Type-based amortized resource analysis with integers and arrays. In Proc. 12th FLOPS, volume 8475 of LNCS, pages 152–168, 2014.
8 M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In Proc. 30th POPL, pages 185–197. ACM, 2003.
9 M. Hofmann and G. Moser. Amortised resource analysis and typed polynomial interpretations. In Proc. of Joint 25th RTA and 12th TLCA, volume 8560 of LNCS, pages 272–286, 2014.
M. Hofmann and G. Moser. Multivariate amortised resource analysis for term rewrite systems. In Proc. 13th TLCA, volume 38 of LIPIcs, pages 241–256, 2015. doi:10.4230/LIPIcs.TLCA.2015.241

M. Hofmann and D. Rodriguez. Automatic type inference for amortised heap-space analysis. In Proc of 22nd ESOP, pages 593–613, 2013. doi:10.1007/978-3-642-37036-6_32

S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. Static determination of quantitative resource usage for higher-order programs. In Proc. 37th POPL, pages 223–236. ACM, 2010.

S. Jost, H.-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. “Carbon Credits” for resource-bounded computations using amortised analysis. In Proc. 2nd FM, volume 5850 of LNCS, pages 354–369, 2009.

G. Moser and M. Schneckenreither. Automated amortised resource analysis for term rewrite systems. In Proc. 14th FLOPS, volume 10818 of LNCS, pages 214–229, 2018. doi:10.1007/978-3-319-90686-7

T. Nipkow. Amortized complexity verified. In Proc. 6th ITP, volume 9236 of LNCS, pages 310–324, 2015. doi:10.1007/978-3-319-22102-1_21.

C. Okasaki. Purely functional data structures. Cambridge University Press, 1999.

B. Pierce. Types and programming languages. MIT Press, 2002.

B. Schoenmakers. A systematic analysis of splaying. IPL, 45(1):41–50, 1993.

D. Sleator and R. Tarjan. Self-adjusting binary search trees. JACM, 32(3):652–686, 1985. doi:10.1145/3828.3835

R.E. Tarjan. Amortized computational complexity. SIAM J. Alg. Disc. Meth, 6(2):306–318, 1985.