ATLAS: Automated Amortised Complexity Analysis of Self-Adjusting Data Structures

Lorenz Leutgeb[2], Georg Moser[1], and Florian Zuleger[2]

[1] Department of Computer Science, Universität Innsbruck
[2] Institute of Logic and Computation 192/4, Technische Universität Wien

Abstract. Being able to argue about the performance of self-adjusting data structures such as splay trees has been a main objective, when Sleator and Tarjan introduced the notion of *amortised* complexity.
Analysing these data structures requires sophisticated potential functions, which typically contain logarithmic expressions. Possibly for these reasons, and despite the recent progress in automated resource analysis, they have so far eluded automation. In this paper, we report on the first fully-automated amortised complexity analysis of self-adjusting data structures. Following earlier work, our analysis is based on potential function templates with unknown coefficients.
We make the following contributions: 1) We encode the search for concrete potential function coefficients as an optimisation problem over a suitable constraint system. Our target function steers the search towards coefficients that minimise the inferred amortised complexity. 2) Automation is achieved by using a linear constraint system in conjunction with suitable lemmata schemes that encapsulate the required non-linear facts about the logarithm. We discuss our choices that achieve a scalable analysis. 3) We present our tool ATLAS and report on experimental results for *splay trees*, *splay heaps* and *pairing heaps*. We completely automatically infer complexity estimates that match previous results (obtained by sophisticated pen-and-paper proofs), and in some cases even infer better complexity estimates than previously published.

Keywords: amortised cost analysis · functional programming · self-adjusting data structures · automation · constraint solving

1 Introduction

Amortised analysis, as introduced by Sleator and Tarjan [47, 49], is a method for the worst-case cost analysis of data structures. The innovation of amortised analysis lies in considering the cost of a single data structure operation as part of a sequence of data structure operations. The methodology of amortised analysis allows one to assign a low (e.g., constant or logarithmic) amortised cost to a data structure operation even though the worst-case cost of a single operation might be high (e.g., linear, polynomial or worse). The setup of amortised analysis guarantees that for a sequence of data structure operations the worst-case cost is indeed the number of data structure operations times the amortised cost.

In this way amortised cost analysis provides a methodology for worst-case cost analysis. Notably, the cost analysis of self-adjusting data structures, such as splay trees, has been a main objective already in the initial proposal of amortised analysis [47,49]. Analysing these data structures requires sophisticated potential functions, which typically contain logarithmic expressions. Possibly for these reasons, and despite the recent progress in automated complexity analysis, they have so far eluded automation.

In this paper, we present the first fully-automated amortised cost analysis of self-adjusting data structures, that is, of *splay trees*, *splay heaps* and *pairing heaps*, which so far have only (semi-) manually been analysed in the literature. We implement and extend a recently proposed type-and-effect system for amortised resource analysis [26, 27]. This system belongs to a line of work (see [20, 22–25, 28] and the references therein), where types are template potential functions with unknown coefficients and the type-and-effect system extracts constraints over these coefficients in a syntax directed way from the program under analysis. Our work improves over [26, 27] in three regards: 1) The approach of [26, 27] only supports *type checking*, i.e. verifying that a manually provided type is correct. In this paper, we add an optimisation layer to the set-up of [26, 27] in order to support *type inference*, i.e. our approach does not rely on manual annotations. Our target function steers the search towards coefficients that minimise the inferred amortised complexity. 2) The only case study of [26,27] is partial, focusing on the zig-zig case of the splay tree function `splay`, while we report on the full analysis of the operations of several data structures. 3) [26, 27] does not report on a fully-automated analysis. Besides the requirement that the user needs to provide the resource annotation, the user also has to apply the structural rules of the type system manually. Our tool ATLAS is able to analyse our benchmarks fully automatically. Achieving full automation required substantial implementation effort as the structural rules need to be applied carefully—as we learned during our experiments—in order to avoid a size explosion of the generated constraint system. We evaluate and discuss our design choices that lead to a scalable implementation.

With our implementation and the obtained experimental results we make two contributions to the complexity analysis of data structures:

1.) We automatically infer complexity estimates that match previous results (obtained by sophisticated pen-and-paper proofs), and in some cases even infer better complexity estimates than previously published. In Table 1, we state the complexity bounds computed by ATLAS next to results from the literature. We match or improve the results from [37, 41, 42]. To the best of our knowledge, the bounds for splay trees and splay heaps represent the state-of-the-art. In particular, we improve the bound for the `delete` function of splay trees and all bounds for the splay heap functions. For pairing heaps, Iacono [29,30] has proven (using a more involved potential function) that `insert` and `merge` have constant amortised complexity, while the other data structure operations continue to have an amortised complexity of $k \log_2(|t|)$; while we leave an automated analysis based on Iacono's potential function for future work, we note that his coefficients

function name	ATLAS (automated)	[42] (manual)[3]	[37] (semi-automated)
`ST.splay`	$3/2 \log_2(\|t\|)$	$3/2 \log_2(\|t\|) + 1$	$3/2 \log_2(\|t\|) + 1$
`ST.splay_max`	$3/2 \log_2(\|t\|)$	-	$3/2 \log_2(\|t\|) + 1$
`ST.insert`	$2 \log_2(\|t\|) + 3/2$	$2 \log_2(\|t\| + 1) + O(1)$	$2 \log_2(\|t\|) + 3/2$
`ST.delete`	$5/2 \log_2(\|t\|) + 3$	$3 \log_2(\|t\| + 1) + O(1)$	$3 \log_2(\|t\|) + 2$
`SH.partition`	$3/4 \log_2(\|t\|) + \log_2(\|t\| + 1)$	-	$2 \log_2(\|t\| + 1) + 1$
`SH.insert`	$3/4 \log_2(\|t\|) + \log_2(\|t\| + 1) + 3/2$	-	$3 \log_2(\|t\| + 2) + 1$
`SH.del_min`	$\log_2(\|t\|)$	-	$2 \log_2(\|t\| + 1) + 1$
`PH.merge_pairs`	$3/2 \log_2(\|h\|)$	-	$3 \log_2(\|h\|) + 4$
`PH.insert`	$1/2 \log_2(\|h\|)$	-	$\log_2(\|h\| + 1) + 1$
`PH.merge`	$1/2 \log_2(\|h_1\| + \|h_2\|) + 1$	$1/2 \log_2(\|h_1\| + \|h_2\|)$	$\log_2(\|h_1\| + \|h_2\| + 1) + 2$
`PH.del_min`	$\log_2(\|h\|)$	$\log_2(\|h\|)$	$3 \log_2(\|h\| + 1) + 4$

Table 1: Amortised complexity bounds for splay trees (module name `SplayTree`, abbrev. `ST`), splay heaps (`SplayHeap`, `SH`) and pairing heaps (`PairingHeap`, `PH`).

k in the logarithmic terms are large, and that therefore the small coefficients in Table 1 are still of interest. We will detail below that we used a simpler potential function than [37,41,42] to obtain our results. Hence, also the new proofs of the confirmed complexity bounds can be considered a contribution.

2.) We establish a new approach for the complexity analysis of data structures. Establishing the prior results in Table 1 required considerable effort. Schoenmakers studied in his PhD thesis [42] the best amortised complexity bounds that can be obtained using a parameterised potential function $\phi(t)$, where t is a binary tree, defined by $\phi(\texttt{leaf}) := 0$ and $\phi(\texttt{(}l\texttt{, }d\texttt{, }r\texttt{)}) := \phi(l) + \beta \log_\alpha(|l| + |r|) + \phi(r)$, for real-valued parameters $\alpha, \beta > 0$. Carrying out a sophisticated optimisation with pen and paper, he concluded that the best bounds are obtained by setting $\alpha = \sqrt[3]{4}$ and $\beta = \frac{1}{3}$ for splay trees, and by setting $\alpha = \sqrt{2}$ and $\beta = \frac{1}{2}$ for pairing heaps (splay heaps were proposed only some years later by Okasaki in [38]). Brinkop and Nipkow verify his complexity results for splay trees in the theorem prover Isabelle [37]. They note that manipulating the expressions corresponding to $\beta \log_\alpha(|t|)$ could only partly be automated [4]. For splay heaps, there is to the best of our knowledge no previous attempt to optimise the obtained complexity bounds, which might explain why our optimising analysis was able to improve all bounds. For pairing heaps, Brinkop and Nipkow did not use the optimal parameters reported by Schoenmakers—probably in order to avoid reasoning about polynomial inequalities—, which explains the

[1] [42] uses a different cost metric, i.e. the numbers of arithmetic comparisons, whereas we and [37] count the number of (recursive) function applications. We adapted the results of [42] to our cost metric to make the results easier to compare, i.e. the coefficients of the logarithmic terms are by a factor 2 smaller compared to [42].

[4] Nipkow et al. [37] state "The proofs in this subsection require highly nonlinear arithmetic. Only some of the polynomial inequalities can be automated with Harrison's sum-of-squares method [16]."

worse complexity bounds. In contrast to the discussed approaches, we were able to verify and improve the previous results fully automatically. Our approach uses a variation of Schoenmakers' potential function, where we roughly fix $\alpha = 2$ and leave β as a parameter for the optimisation phase (see Section 2 for more details). Despite this choice, our approach was able to derive bounds that match and improve the previous results, which came as a surprise to us. Looking back at our experiments and interpreting the obtained results, we recognise that we might have been in luck with the particular choice of the potential function (because we can obtain the previous results despite fixing $\alpha = 2$). However, we would not have expected that an automated analysis is able to match and improve all previously reported coefficients, which shows the power of the optimisation phase. *Thus, we believe that our results suggest a new approach for the complexity analysis of data structures.* So far, self-adjusting data structures had to be analysed manually. This is possibly due to the use of sophisticated potential functions, which may contain logarithmic expressions. Both features are challenging for automated reasoning. Our results suggest that the following alternative (see Sections 2 and 4.2 for more details): (i) Fix a parameterised potential function; (ii) derive a (linear) constraint system over the function parameters from the AST of the program; (iii) capture the required non-linear reasoning in lemmata, and use Farkas' lemma to integrate the application of these lemmata into the constraint system (in our case two lemmata, one about an arithmetic property and one about the monotonicity of the logarithm, were sufficient for all of our benchmarks); and finally (iv) find values for the parameters by an (optimising) constraint solver. We believe that our approach will carry over to other data structures: one needs to adapt the potential functions and add suitable lemmata, but the overall setup will be the same. We compare the proposed methodology to program synthesis by sketching [48], where the synthesis engineer communicates her main insights to the synthesis engine (in our case the potential functions plus suitable lemmata), and a constraint solver then fills in the details. As conclusion from our benchmarking, we observe that an automated analysis of sophisticated data structures are possible without the need to (i) resort to user guidance; (ii) forfeit optimal results; or (iii) be bogged down in computation times. These results also show how dependencies on properties of functional correctness of the code can be circumvented.

Related Work. To the best of our knowledge the here presented automated amortised analysis of self-adjusting data-structures is novel and unparalleled in the literature. However, there is a vast amount of literature on (automated) resource analysis. Without hope for a completeness, we briefly mention [1–7, 9–11, 14, 15, 17, 18, 20, 22–25, 39, 44–46, 52] for an overview of the field. Logarithmic and sublinear bounds are typically not in the focus of the cited approaches, but can be inferred by some tools. In the recurrence relations based approach to cost analysis [1] refinements of linear ranking functions are combined with criteria for divide-and-conquer patterns; this allows the tool PUBS to recognise logarithmic bounds for some problems, but examples such as *mergesort* or *splaying* are beyond the scope of this approach. Logarithmic and exponential terms are inte-

grated into the synthesis of ranking functions in [8], making use of an insightful adaption of Farkas' and Handelman's lemmas. The approach is able to handle examples such as *mergesort*, but again not suitable to handle self-balancing data structures. A type based approach to cost analysis for an ML-like language is presented in [50], which uses the Master Theorem to handle divide-and-conquer-like recurrences. Recently, support for the Master Theorem was also integrated for the analysis of rewriting systems [51], extending [4] on the modular resource analysis of rewriting to so-called logically constrained rewriting systems [12]. The resulting approach also supports the fully automated analysis of *mergesort*.

Structure. In Sections 2 and 3 we review the type system of [26,27]. We sketch the challenges to automation in Section 4 and present our contributions in Sections 5 and 6. Finally, we conclude in Section 7.

2 Step by Step to an Automated Analysis of Splaying

In this and the next section we sketch the theory developed by Hofmann et al. in [27], in order to be able to present the contributions of this article in Section 4 and 5. For brevity, we restrict our exposition to those parts essential in the analysis of a particular program code. As motivating example consider *splay trees*, introduced by Sleator and Tarjan [47, 49]. *Splaying* is the most important operation on splay trees, which performs rotation. Consider Figure 1, a depiction of the zig-zig case of `splay`, which implements *splaying*.

The analysis of [27] (see also [26]) is formulated in terms of the physicist's method of amortised analysis in the style of Sleator and Tarjan [47, 49]. The central idea of this approach is to assign a *potential* to the data structures of interest such that the difference in potential before and after executing a function is sufficient to pay for the actual cost of the function, i.e. one chooses potential functions ϕ, ψ such that $\phi(v) \geqslant c_f(v) + \psi(f(v))$ holds for all inputs v to a function f, where $c_f(v)$ denotes the *worst-case cost* of executing function f on v. This generalises the original formulation, which can be seen by setting $\phi(v) := a_f(v) + \psi(v)$, where $a_f(v)$ denotes the *amortised cost* of f.

In order to be able to analyse self-adjusting data structures such as splay trees, one needs potential functions that can express *logarithmic* amortised cost. Hofmann et al. [26, 27] propose to make use of a variant of Schoenmakers' potential, $\mathsf{rk}(t)$ for a tree t, cf. [37, 41, 42], defined inductively by

$$\mathsf{rk}(\texttt{leaf}) := 1 \qquad \mathsf{rk}(\texttt{(}l\texttt{, }d\texttt{, }r\texttt{)}) := \mathsf{rk}(l) + \log_2(|l|) + \log_2(|r|) + \mathsf{rk}(r)\ ,$$

where l, r are the left resp. right child of the tree `(`l`, `d`, `r`)`, $|t|$ denotes the size of a tree (defined as the number of leaves of the tree), and d is some data element that is ignored by the potential function. Besides Schoenmakers' potential, further basic potential functions need to be added to the analysis: For a sequence of m trees $t_1, \ldots, t_m$ and coefficients $a_i, b \in \mathbb{N}$, the potential function

$$p_{(a_1,\ldots,a_m,b)}(t_1, \ldots, t_m) := \log_2(a_1 \cdot |t_1| + \cdots + a_m \cdot |t_m| + b)$$

```
splay a t = match t with
 | (cl, c, cr) -> match cl with
   | (bl, b, br) -> let s = splay a bl in match s with
     | (al, a', ar) -> (al, a', (ar, b, (br, c, cr)))
```

Fig. 1: Zig-zig case of the `splay` function.

denotes the logarithm of a linear combination of the sizes of the tree.

Following [37], we set the cost $c_{\texttt{splay}}(t)$ of splaying a tree t to be the number of recursive calls to `splay`. Splaying and all operations that depend on splaying can be done in $O(\log_2 n)$ amortised cost. Employing the above introduced potential functions, the analysis of [27] is able verify the following cost annotation for splaying (the annotation needs to be provided by the user):

$$\mathsf{rk}(t) + 3 \cdot p_{(1,0)}(t) + 1 \geqslant c_{\texttt{splay}}(t) + \mathsf{rk}(\texttt{splay a t}) \,. \tag{1}$$

From this result, one directly reads off $3 \cdot p_{(1,0)}(t) + 1 = 3 \cdot \log_2(|t|) + 1$ as bound on the amortised cost of splaying.[5]

Based on earlier work [6, 20, 22–25, 28], [27] employs a *type-and-effect system* that uses *template potential functions*, i.e. functions of a fixed shape with indeterminate coefficients. The key challenge is to identify templates that are suitable for logarithmic analysis and that are closed under the basic operations of the considered programming language. For example, one introduces the coefficients $q_*, q_{(1,0)}, q_{(0,2)}, q'_*, q'_{(1,0)}, q'_{(0,2)}$ and introduces the potential function templates

$$\begin{aligned} \Phi(t : \mathsf{T} | Q) &:= q_* \cdot \mathsf{rk}(t) + q_{(1,0)} \cdot p_{(1,0)}(t) + q_{(0,2)} \cdot p_{(0,2)}(t) \\ \Phi(\texttt{splay a t} : \mathsf{T} | Q') &:= q'_* \cdot \mathsf{rk}(\texttt{splay a t}) + \\ &\quad + q'_{(1,0)} \cdot p_{(1,0)}(\texttt{splay a t}) + q'_{(0,2)} \cdot p_{(0,2)}(\texttt{splay a t}) \,, \end{aligned}$$

for the input and output of the `splay` function. The type system then derives constraints on the template function coefficients, as indicated in the sequel. We take up further discussion of the constraint system, in particular how to maintain a scalable analysis, in Section 4.

We explain the use of the type system on the motivating example. For brevity, type judgements and the type rules are presented in a simplified form. In particular, we restrict our attention to tree types, denoted as T. This omission is inessential to the actual complexity analysis. For the full set of rules see [27].

Let e denote the body of the function definition of `splay a t`, depicted in Figure 1. Our automated analysis infers an *annotated type* of splaying, by verifying that the type judgement

$$t : \mathsf{T} | Q \vdash e : \mathsf{T} | Q' \,, \tag{2}$$

is derivable. As above, types are decorated with *annotations* $Q := [q_*, q_{(1,0)}, q_{(0,2)}]$ and $Q' := [q'_*, q'_{(1,0)}, q'_{(0,2)}]$—employed to express the potential carried by the arguments to `splay` and its results.

[5] For ease of presentation, we elide the underlying semantics for now and simply write "`splay a t`" for the resulting tree t', obtained after evaluating `splay a t`.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\texttt{splay}:\mathsf{T}|Q \to \mathsf{T}|Q'}{bl:\mathsf{T}|Q \vdash \texttt{splay a bl}:\mathsf{T}|Q'-1}\ (\mathsf{app}) \qquad \Delta|R \vdash^{\mathrm{cf}} \texttt{splay a bl}:\mathsf{T}|R'}{cr:\mathsf{T},br:\mathsf{T},s:\mathsf{T}|Q_4 \vdash \texttt{match}\ x\ \texttt{with}\ \texttt{|(}al\texttt{,}a'\texttt{,}ar\texttt{)}\ \texttt{->}\ t':\mathsf{T}|Q'}\ (\mathsf{let}:\mathsf{T})}{cr:\mathsf{T},bl:\mathsf{T},br:\mathsf{T}|Q_3 \vdash e_1':\mathsf{T}|Q'}\ (\mathsf{w})}{cr:\mathsf{T},bl:\mathsf{T},br:\mathsf{T}|Q_2 \vdash e_1':\mathsf{T}|Q'}\ (\mathsf{match})}{cl:\mathsf{T},cr:\mathsf{T}|Q_1 \vdash \texttt{match}\ cl\ \texttt{with}\ \texttt{|(}bl\texttt{,}b\texttt{,}br\texttt{)}\ \texttt{->}\ e_1':\mathsf{T}|Q'}\ (\mathsf{match})}{t:\mathsf{T}|Q \vdash \texttt{match}\ t\ \texttt{with|(}cl\texttt{,}c\texttt{,}cr\texttt{)}\ \texttt{->}\ e_1:\mathsf{T}|Q'}\ (\mathsf{match})$$

Fig. 2: Partial Typing Derivation for the motivating example `splay`.

The soundness theorem of the type system (Theorem 1) expresses that if the above type judgement is derivable, then the total cost $c_{\texttt{splay}}(t)$ of splaying is bound by the difference between $\Phi(t:\mathsf{T}|Q)$ and $\Phi(\texttt{splay a t}:\mathsf{T}|Q')$, i.e. $\Phi(t:\mathsf{T}|Q) \geqslant c_{\texttt{splay}}(t) + \Phi(\texttt{splay a t}:\mathsf{T}|Q')$. In particular, Equation (1) can be derived in this way.

We now provide an intuition on the type-and-effect system, stepping through the code of Figure 1. The corresponding type derivation tree is depicted in Figure 2. We note that the tree contains further annotations Q_1, Q_2, Q_3, Q_4 (besides the annotations Q and Q') which again represent the unknown coefficients of potential function templates. The goal of the type-and-effect system is to provide constraints for each programming construct that connect the annotations in subsequent derivation steps, e.g. Q_2 and Q_3. The type-and-effect system operates *syntax-directed* and formulates one rule per programming languages construct. We now discuss some of these rules for the partial derivation for `splay`.

The outermost command of e is a `match` statement, for which the following rule is applied:

$$\dfrac{cl:\mathsf{T},cr:\mathsf{T}|Q_1 \vdash e_1:\mathsf{T}|Q'}{t:\mathsf{T}|Q \vdash \texttt{match}\ t\ \texttt{with}\ \texttt{|}\ \texttt{(}cl\texttt{,}c\texttt{,}cr\texttt{)}\ \texttt{->}\ e_1:\mathsf{T}|Q'}\ (\mathsf{match})\ .$$

Here e_1 denotes the subexpression of e, which constitutes the nested pattern match. Primarily, this is a standard type rule for pattern matching. The novelty are the constraints on the annotations Q, Q' and Q_1. More precisely, (match) induces the constraints

$$q_1^1 = q_2^1 = q_* \qquad q^1_{(1,1,0)} = q_{(1,0)} \qquad q^1_{(1,0,0)} = q^1_{(0,1,0)} = q_* \qquad q^1_{(0,0,2)} = q_{(0,2)}\ ,$$

which can be directly read-off the definition of $\mathsf{rk}(t) = \mathsf{rk}(cl) + \log_2(|cl|) + \log_2(|cr|) + \mathsf{rk}(cr)$. Similarly, the nested `match` command, starting expression e_1', is subject to the same rule; the resulting constraints amount to

$$\begin{aligned} q_1^2 = q_2^2 = q_3^2 \qquad & q^2_{(0,0,0,2)} = q^1_{(0,0,2)} \qquad & q^2_{(1,1,1,0)} = q^1_{(1,1,0)} \\ q^2_{(0,1,1,0)} = q^1_{(1,0,0)} \qquad & q^2_{(1,0,0,0)} = q^1_{(0,1,0)} \qquad & q^2_{(0,1,0,0)} = q^2_{(0,0,1,0)} = q^1_1\ . \end{aligned}$$

Besides the rules for programming language constructs, the type-and-effect system contains *structural rules*, which operate on the type annotations themselves. The *weakening* rule allows a suitable adaptation of the coefficients of the potential function $\Phi(\Gamma|Q_2)$ to obtain a new potential function $\Phi(\Gamma|Q_3)$, where we use the shorthand $\Gamma := cr:\mathsf{T}, bl:\mathsf{T}, br:\mathsf{T}$:

$$\frac{\Gamma|Q_3 \vdash e_1' : \mathsf{T}|Q' \quad \Phi(\Gamma|Q_2) \geqslant \Phi(\Gamma|Q_3)}{\Gamma|Q_2 \vdash e_1' : \mathsf{T}|Q'} \ (\mathsf{w})$$

The difficulty in applying the *weakening* rule, consists in discharging the constraint:

$$\Phi(\Gamma|Q_2) \geqslant \Phi(\Gamma|Q_3) \tag{3}$$

Note, that the comparison is to be performed *symbolically*, that is, abstracted from the concrete value of the variables. We emphasise that this step can neither be avoided, nor easily moved to the axioms of the derivation, as in related approaches in the literature [19,21–23,28,31,35]. We use Farkas' Lemma in conjunction with two facts about the logarithm to linearise this symbolic comparison, namely the monotonicity of the logarithm and the fact that $2+\log_2(x)+\log_2(y) \leqslant 2\log_2(x+y)$ for all $x, y \geqslant 1$. For example, for the facts $\log_2(|bl|) \leq \log_2(|bl|+|br|)$ and $2+\log_2(|bl|)+\log_2(|cr|+|br|) \leq 2\log_2(|cr|+|bl|+|br|)$, we use Farkas' Lemma to generate the constraints

$$\begin{aligned} q^2_{(0,0,0,2)} + 2f &\geqslant q^3_{(0,0,0,2)} \\ q^2_{(1,0,1,0)} + f &\geqslant q^3_{(1,0,1,0)} \\ q^2_{(1,1,1,0)} - 2f &\geqslant q^3_{(1,1,1,0)} \end{aligned} \qquad \begin{aligned} q^2_{(0,1,0,0)} + f + g &\geqslant q^3_{(0,1,0,0)} \\ q^2_{(0,1,1,0)} - g &\geqslant q^3_{(0,1,1,0)} \end{aligned}$$

for some coefficients $f, g \geqslant 0$ introduced by Farkas' Lemma. We note that Farkas' Lemma can be interpreted as systematically exploring all positive-linear combinations of the considered mathematical facts. This can be seen on the above example: one can combine g times the first fact with f times the second fact.

Next, we apply the rule for the `let` expression. This rule is the most involved typing rule in the system proposed by Hofmann et al. [27].

$$\frac{\Delta|Q \vdash e_2 : \mathsf{T}|Q' - 1 \quad \Delta|R \vdash^{\mathrm{cf}} e_2 : \mathsf{T}|R' \quad \Theta|Q_4 \vdash e_3 : \mathsf{T}|Q'}{cr:\mathsf{T}, bl:\mathsf{T}, br:\mathsf{T}|Q_3 \vdash \texttt{let } s \texttt{ = } e_2 \texttt{ in } e_3 : \mathsf{T}|Q'} \ (\mathsf{let} : \mathsf{T})$$

Ignoring the annotations and in particular the second premise for a moment, the type rule specifies a standard typing for a `let` expression. We note that, as required by the rule, all variables in the type context Γ occur at most once in the let-expression. Γ can then be split into contexts $\Delta := bl:\mathsf{T}$ and $\Theta := cr:\mathsf{T}, br:\mathsf{T}$. Here, $e_2 :=$ `splay a bl` and e_3 denotes the last `match` statement in e. The let-rule facilitates a splitting of the potential Q_3 for the evaluation of e_2 and e_3 according to the type contexts Δ and Θ. Abusing notation, the

distribution of potentials facilitated by the let-rule can be stated very roughly as two "equalities", that is, (i) "$Q_3 = Q{+}R{+}P$" and (ii) "$Q_4 = (Q'{-}1){+}R'{+}P$". (i) states that the potential Q_3 pays for evaluating the `let` expression e_2 (with and without costs, requiring the potential Q and R) and leaves the remainder potential P. (ii) states that the potential Q_4 is constituted of the remainder potential P and of the potentials left after evaluating e_2 (with and without costs, i.e. potentials $Q'-1$ and R'). E.g. Q_4 is given by the following constraints

$$\begin{array}{llll} q^4_1 = q^3_1 & q^4_3 = q'_* & q^4_{(1,0,0,0)} = q^3_{(1,0,0,0)} & q^4_{(1,1,1,0)} = {r'}_{(1,0)} \\ q^4_2 = q^3_3 & q^4_{(0,1,0,0)} = q^3_{(0,0,1,0)} & q^4_{(1,1,0,0)} = q^3_{(1,0,1,0)} & \end{array} ,$$

where the coefficients q^3 stem from the remainder potential of Q_3, the coefficient q'_* from $Q'-1$ and ${r'}_{(1,0)}$ from R'.

The most original part of this type rule is the second premise $\Delta|R \vdash^{\mathrm{cf}} \texttt{splay a bl}:\mathsf{T}|R'$. Here, $\vdash^{\mathrm{cf}}$ denotes the same kind of typing judgement as used in the overall typing derivation, but where all costs are set to zero (hence, the superscript *cost-free*). Let us assume $R = [r_{(1,0)}]$, $R' = [{r'}_{(1,0)}]$, and that ATLAS was able to establish that

$$\Phi(bl:\mathsf{T}|R) = \log_2(|bl|) \geqslant \log_2(|s|) = \Phi(s:\mathsf{T}|R') \;, \tag{4}$$

establishing the coefficients $r_{(1,0)} = 1$ and ${r'}_{(1,0)} = 1$. (We note that cost-free typing derivations as in Equation (4) constitute a *size analysis* that relates the sizes of input and output). Then, ATLAS infers from (4), taking advantage of the monotonicity of log, that

$$\log_2(|cr| + |bl| + |br|) \geqslant \log_2(|cr| + |br| + |s|) \;.$$

This inequality expresses that if the summand $\log_2(|cr|+|bl|+|br|)$ is included in the potential $\Phi(\Gamma|Q_3)$, then the summand $\log_2(|cr|+|br|+|s|)$ may be included in the potential $\Phi(cr:\mathsf{T}, br:\mathsf{T}, s:\mathsf{T}|Q_4)$. (The two logarithmic terms correspond to the coefficients $q^3_{(1,1,1,0)}$ and $q^4_{(1,1,1,0)}$ marked in red above.) Thus, the cost-free derivation allows the potential R to pass from Q_3, via R', to Q_4. This is crucial for being able to pay for the evaluation of e_3.

The let-rule has the three premises $\Delta|Q \vdash e_2:\mathsf{T}|Q'-1$, $\Delta|R \vdash^{\mathrm{cf}} e_2:\mathsf{T}|R'$ and $\Theta|Q_4 \vdash e_3:\mathsf{T}|Q'$. We focus here on the first premise and do not state the derivations for the other two premises (such derivations can be found in [27]). The judgement $\Delta|Q \vdash \texttt{splay a t}:\mathsf{T}|Q'-1$ can be derived by the rule for function application, which states a cost of 1 with regard to the type signature of `splay`, represented by decrementing the potential induced by the annotation Q'.

$$\frac{\texttt{splay}:\mathsf{T}|Q \to \mathsf{T}|Q'}{t:\mathsf{T}|Q \vdash \texttt{splay a t}:\mathsf{T}|Q'-1}\ (\mathsf{app})$$

The rule for function application is an axiom, and closes this branch of the typing derivation. This concludes the presentation of the partial type inference

given in Figure 2. Similarly to the above example of `splay`, estimates for the amortised costs of insertion and deletion on splay trees can be automatically inferred by our tool ATLAS. Further, our analysis handles similar self-adjusting data structures like *pairing heaps* and *splay heaps* (see Section 6.1).

3 Technical Foundation

In this short section, we provide a more detailed account of the formal system underlying our tool ATLAS. We state the soundness of the system in Theorem 1.

A *typing context* is a mapping from variables $\mathcal{V}$ to types; denoted by uppercase Greek letters. A program P is a set of typed 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 an *environment*) σ 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 employ a simple cost-sensitive big-step semantics based on eager evaluation, dressed up with cost assertions. The judgement $\sigma \overset{\ell}{\vdash} e \Rightarrow v$ means that under environment σ, expression e is evaluated to value v in exactly ℓ steps. Here only rule applications emit (unit) costs. For brevity, the formal definition of the semantics is omitted but can be found in [27].

In Section 2, we introduced a variant of Schoenmakers' potential function, denoted as $\mathsf{rk}(t)$, and the additional potential functions $p_{(a_1,\ldots,a_m,b)}(t_1, \ldots, t_m) := \log_2(a_1 \cdot |t_1| + \cdots + a_m \cdot |t_m| + b)$, denoting the $\log_2$ of a linear combination of tree sizes. $\log_2$ denotes the logarithm to the base 2; throughout the paper we stipulate $\log_2(0) := 0$ in order to avoid case distinctions. Note that the constant function 1 is representable: $1 = \lambda t.\, \log_2(0 \cdot |t| + 2) = p_{(0,2)}$. We are now ready to state the resource annotation of a sequence of trees:

Definition 1. *A* resource annotation *or simple* annotation *of length m is a sequence $Q = [q_1, \ldots, q_m] \cup [(q_{(a_m,\ldots,a_n,b)})_{a_i,b\in\mathbb{N}}]$, vanishing almost everywhere. Let $t_1, \ldots, t_m$ be a sequence of trees. Then, the potential of $t_1, \ldots, t_m$ wrt. Q is given by*

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

In case of an annotation of length 1, we sometimes write q_* instead of q_1, as we already did above.

Example 1. Let t be a tree, then its potential could be defined as follows: $\mathsf{rk}(t) + 3 \cdot \log_2(|t|) + 1$. Wrt. the above definition this potential becomes representable by setting $q_* := 1, q_{(1,0)} := 3, q_{(0,2)} := 1$. Thus, $\Phi(t|Q) = \mathsf{rk}(t) + 3 \cdot \log_2(|t|) + 1$. □

Let σ be a substitution, let Γ denote a typing context and let $x_1 : \mathsf{T}, \ldots, x_m : \mathsf{T}$ denote all tree types in Γ. A *resource annotation for Γ* or simply *annotation* is an annotation for the sequence of trees $x_1\sigma, \ldots, x_m\sigma$. We define the *potential* of the annotated context $\Gamma|Q$ wrt. a substitution σ as $\Phi(\sigma; \Gamma|Q) := \Phi(x_1\sigma, \ldots, x_m\sigma|Q)$.

Definition 2. *An* annotated signature $\mathcal{F}$ *maps functions* f *to sets of pairs of the annotation type for the arguments and the annotation type of the result:*

$$\mathcal{F}(f) := \{\alpha_1 \times \cdots \times \alpha_n | Q \to \beta | Q' \colon Q, Q' \text{ are annotations, } Q \text{ is of length } m\} \ .$$

We suppose f *takes* n *arguments of which* m *are trees;* $m \leqslant n$ *by definition.*

Instead of $\alpha_1 \times \cdots \times \alpha_n | Q \to \beta | Q' \in \mathcal{F}(f)$, we sometimes succinctly write $f \colon \alpha_1 \times \cdots \times \alpha_n | Q \to \beta | Q'$. The *cost-free* signature, denoted as $\mathcal{F}^{\mathrm{cf}}$, is similarly defined.

Example 2. Consider the function `splay` from above. Its signature is formally represented as $\mathsf{B} \times \mathsf{T} | Q \to \mathsf{T} | Q'$, where $Q := [q_*] \cup [(q_{(a,b)})_{a,b \in \mathbb{N}}]$ and $Q' := [q'_*] \cup [(q'_{(a,b)})_{a,b \in \mathbb{N}}]$. We leave it to the reader to specify the coefficients in Q, Q' so that the rule (app) as depicted in Section 2 can indeed by employed to type the recursive call of `splay`.

Let $Q = [q_*] \cup [(q_{(a,b)})_{a,b \in \mathbb{N}}]$ be an annotation such that $q_{(a,b)} > 0$. Then $Q' := Q - 1$ is defined as follows: $Q' = [q_*] \cup [(q'_{(a,b)})_{a,b \in \mathbb{N}}]$, where $q'_{(0,2)} := q_{(0,2)} - 1$ 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 $p_{(0,2)}(t) = \log_2(0|t|+2) = 1$, so the annotation $Q - 1$, decrements cost 1 from the potential induced by Q.

Type-and-Effect System. The typing system makes use of a *cost-free* semantics, which does not attribute any costs to the calculation. I.e. the rule (app) (Section 2) is changed so that no cost is emitted. The cost-free application rule is denoted as (app : cf). The cost-free typing judgement is written as $\Gamma | Q \vdash^{\mathrm{cf}} e \colon \alpha | Q'$. The judgement $\Gamma | Q \vdash e \colon \alpha | Q'$ is governed by a plethora of typing rules. We have illustrated several typing rules in Section 2 (the complete set of typing rules can be found in [27]).

A program P is called *well-typed* if for any rule $f(x_1, \ldots, x_k) = e \in \mathsf{P}$ and any annotated signature $f \colon \alpha_1 \times \cdots \times \alpha_k | Q \to \beta | Q'$, we have $x_1 \colon \alpha_1, \ldots, x_k \colon \alpha_k | Q \vdash e \colon \beta | Q'$. A program P is called *cost-free* well-typed, if the cost-free typing relation is employed.

Hofmann et al. establish the following soundness result:[6]

Theorem 1 (Soundness Theorem). *Let* P *be well-typed and let* σ *be an environment. Suppose* $\Gamma | Q \vdash e \colon \alpha | Q'$ *and* $\sigma \overset{\ell}{\vdash} e \Rightarrow v$*. Then* $\Phi(\sigma; \Gamma | Q) - \Phi(v | Q') \geqslant \ell$*. Further, if* $\Gamma | Q \vdash^{cf} e \colon \alpha | Q'$*, then* $\Phi(\sigma; \Gamma | Q) \geqslant \Phi(v | Q')$*.*

4 The Road to Automation, Continued

The above sketched type-and-effect system, originally proposed in [27], is only a first step towards full automation. Several challenges need to be overcome, which we detail in this section.

[6] Note that soundness assumes a terminating execution $\sigma \overset{\ell}{\vdash} e \Rightarrow v$ of P. We point out that our analysis does not guarantee the termination of P for all environments σ.

4.1 Type Checking

Comparison between logarithmic expressions, constitutes a first major challenge, as such a comparison cannot be directly encoded as a *linear* constraint problem. To achieve such *linearisation*, [27] makes use of the following: (i) a subtly and surprisingly effective variant of Schoenmakers potential (see Section 2); (ii) mathematical facts about the logarithm function—like Lemma 1 below—referred to as *expert knowledge*; and finally (iii) Farkas' Lemma for turning the universally-quantified premise of the weakening rule into an existentially-quantified statement that can be added to the constraint system—see Lemma 2. A simple mathematical fact that is employed by Hofmann et al.— following earlier pen-and-paper proofs in the literature [37, 38, 41]—states as follows:

Lemma 1. *Let* $x, y \geqslant 1$. *Then* $2 + \log_2(x) + \log_2(y) \leqslant 2\log_2(x + y)$.

We remark that our automated analysis shows that this lemma is not only crucial in the analysis of splaying, but also for the other data structures we have investigated. Further, Hofmann et al. state and prove the following variant of Farkas' Lemma, which lies at the heart of an effective transformation of comparison demands like (3) into a linear constraint problem. Note that $\vec{u}$ and $\vec{f}$ denote column vectors of suitable length.

Lemma 2 (Farkas' Lemma). *Suppose* $A\vec{x} \leqslant \vec{b}, \vec{x} \geqslant 0$ *is solvable. Then the following assertions are equivalent. (i)* $\forall \vec{x} \geqslant 0.\ A\vec{x} \leqslant \vec{b} \Rightarrow \vec{u}^T\vec{x} \leqslant \lambda$ *and (ii)* $\exists \vec{f} \geqslant 0.\ \vec{u}^T \leqslant \vec{f}^T A \wedge \vec{f}^T\vec{b} \leqslant \lambda$.

The lemma allows the assumption of *expert knowledge* through the assumption $A\vec{x} \leqslant \vec{b}$ for all $\vec{x} \geqslant 0$. E.g., thus formalised expert knowledge is a clear point of departure for additional information. E.g. Hofmann et al. [27] propose the following potential extensions: (i) additional mathematical facts on the log function; (ii) a dedicated size analysis ; (iii) incorporation of basic static analysis techniques. The incorporation of Farkas' Lemma with suitable expert knowledge is already essential for *type checking*, whenever the symbolic weakening rule (3) needs to be discharged.

ATLAS incorporates two facts into the expert knowledge: Lemma 2 and the monotonicity of the logarithm (see Section 5). We found these two facts to be sufficient for handling our benchmarks, i.e. expert knowledge of form (ii) and (iii) was not needed. (We note though that we have experimented with adding a dedicated size analysis (ii), which interestingly increased the solver performance, despite generating a large constraint system).

We indicate how ATLAS may be used to solve the constraints generated for the example in Section 2. We recall the crucial application of the *weakening* step between annotations Q_2 and Q_3. This weakening step can be automatically discharged using the monotonicity of logs and Lemma 1. (More precisely, ATLAS employs the mode `w{mono l2xy}` see, Section 5.) For example, ATLAS is able to

verify the validity of the following concrete constants:

$$Q_2\colon q_1^2 = q_2^2 = q_3^2 = 1 \qquad Q_3\colon q_1^3 = q_2^3 = q_3^3 = 1$$

$$q^2_{(0,0,0,2)} = 1 \quad q^2_{(0,1,1,0)} = 1 \qquad q^3_{(0,0,0,2)} = 2 \quad q^3_{(1,0,0,0)} = 1$$
$$q^2_{(0,0,1,0)} = 1 \quad q^2_{(1,0,0,0)} = 1 \qquad q^3_{(0,0,1,0)} = 1 \quad q^3_{(1,0,1,0)} = 1$$
$$q^2_{(0,1,0,0)} = 1 \quad q^2_{(1,1,1,0)} = 3 \qquad q^3_{(0,1,0,0)} = 3 \quad q^3_{(1,1,1,0)} = 1$$

4.2 Type Inference

We extend the type-and-effect system of [27] from type checking to type inference. Further, we automate the application of structural rules like *sharing* or *weakening*, which have so far required user guidance.

The two central contributions of this paper, as delineated in the introduction, are based on significant improvement over the state-of-the-art as described above. Concretely, they came about by a novel (i) *optimisation layer*; (ii) a careful control of the *structural rules*; (iii) the generalisation of user-defined *proof tactics* into an overall strategy of type inference; and (iv) provision of an automated amortised analysis in the sense of Sleator and Tarjan. In the sequel of the section, we will discuss these stepping stones towards full automation in more details.

Optimisation Layer. We add an optimisation layer to the set-up, in order to support *type inference*. This allows for the inference of (optimal) type annotations based on user-defined type annotations. For example, assume the user-provided type annotation $\mathsf{rk}(t)+3\log_2(|t|)+1 \to \mathsf{rk}(\texttt{splay}(t))$ can in principle be checked automatically. Then—instead of checking this annotation—ATLAS automatically *optimises* the signature, by minimising the deduced coefficients. (In Section 5 we discuss how this optimisation step is performed.) That is, ATLAS reports the following annotation

$$\texttt{splay}\colon {}^1\!/_2\,\mathsf{rk}(t) + {}^3\!/_2\log_2(|t|) \to {}^1\!/_2\,\mathsf{rk}(\texttt{splay}(t))\ ,$$

which yields the *optimal* amortised cost of splaying of ${}^3\!/_2\log_2(|t|)$. Optimality here means that no better bound has been obtained by earlier pen-and-paper verification methods (compare the discussion in Section 1).

Structural Rules. We observed that an unchecked application of the structural rules, that is of the *sharing* and the *weakening* rule, quickly leads to an explosion of the size of the constraint system and thus to de-facto unsolvable problems. To wit, an earlier version of our implementation ran continuously for *24/7* without being able to infer a type for the complete definition of the function `splay`.[7]

The type-and-effect system proposed by Hofmann et al. is in principle *linear*, that is, variables occur at most once in the function body. For example, this is employed in the definition of the let-rule, cf. Section 2. However, a *sharing* rule

[7] The code ran single-threaded on AMD® Ryzen 7 3800 @ 3.90 GHz.

```
(match (* t *) leaf
  (match (* cl *) ?
    (w{l2xy} (let:tree:cf (* s *)
      app (* splay_eq a bl *)
      (match leaf
        (let:tree:cf node (let:tree:cf node (w{mono} node))))))))
```

Fig. 3: Tactic that matches the zig-zig case of `splay` as shown in Fig. 1.

is admissible, that allows to treat multiple occurrences of variables. Occurrences of non-linear variables are suitably renamed apart and the carried potential is shared among the variants. (See [27] for the details.) The number of variables strongly influences the size of the constraint problem. Hence, eager application of the sharing rule proved infeasible. Instead, we restricted its application to individual program traces. For the considered benchmark examples, this removed the need for sharing altogether.

With respect to *weakening*, a careful application of the weakening rule proved necessary for performance reasons: First, we apply weakening only selectively. Second, when applying weakening, we employ different levels of *granularity*. We may only perform a simple coefficient comparison, or we may apply monotonicity or Lemma 1 or both in conjunction with Farkas' Lemma. We give the details in Section 5.

Proof Tactics. Hofmann et al. [27] already propose user-defined proof plans, so-called *tactics*, to improve the effectivity of type checking. In combination with our optimisation framework, tactics allow to significantly improve type annotations. To wit, ATLAS can be invoked with user-defined resource annotations for the function `splay`, representing its "standard" amortised complexity (e.g. copied from Okasaki's book [38]) and an easily definable tactic, cf. Figure 3. Then, ATLAS automatically derives the optimal bound reported above. Still, for full-automation tactics are clearly not sufficient. In order to obtain *type inference* in general, we developed a generalisation of all the tactics that proved useful on our benchmark and incorporated this proof search strategy into the type inference algorithm. Using this, the aforementioned (unsuccessful) week-long quest for a type inference of splaying can now be successfully answered (in an optimal form) in mere minutes.

We'd like to argue that ATLAS proof search strategy for full automation is free of bias towards the provided complexity analysis. As detailed in Section 5, the heuristics incorporates common design principles of the data structures analysed. Thus, we exploit recurring patterns in the input (destructuring of input trees, handling base/recursive cases, rotations) not in the solution. The situation is similar to the choice of the potential functions, which we expect to generalise to other data structures. Similarly, we expect generalisability of the current proof search strategy.

Automated Amortised Analysis In Section 2, we provided a high-level introduction into the potential method and remarked that Sleator and Tarjan's original

formulation is re-obtained, if the corresponding potential functions are defined such that $\phi(v) := a_f(v) + \psi(x)$, see page 5. We now discuss how we can extract amortised complexities in the sense of Sleator and Tarjan from our approach. Suppose, we are interested in an amortised analysis of splay heaps. Then, it suffices to equate the right-hand sides of the annotated signatures of the splay heap functions. That is, we set `del_min`: $\mathsf{T}|Q_1 \to \mathsf{T}|Q'$, `insert`: $\mathsf{B} \times \mathsf{T}|Q_2 \to \mathsf{T}|Q'$ and `partition`: $\mathsf{B} \times \mathsf{T}|Q_3 \to \mathsf{T}|Q'$ for some unknown resource annotations Q_1, Q_2, Q_3, Q'. Note that we use the same annotation Q' for all signatures. We can then obtain a potential function from the annotation Q' in the sense of Sleator and Tarjan and deduce $Q_i - Q'$ as an upper bound on the amortised complexity of the respective function. In Section 5, we discuss how to automatically optimise $Q_i - Q'$ in order to minimise the amortised complexity bound. This automated minimisation is the second major contribution of our work. Our results suggest a new approach for the complexity analysis of data structures. On the one hand, we obtain novel insights into the automated worst-case runtime complexity analysis of involved programs. On the other hand, we provide a proof-of-concept of a computer-aided analysis of amortised complexities of data-structures that so far have only been analysed manually.

5 Implementation

In this section, we present our tool ATLAS, which implements type inference for the type system presented in Sections 2 and 3. ATLAS operates in three phases:

1.) *Preprocessing*, ATLAS parses and normalises the input program;
2.) *Generation of the Constraint System*, ATLAS extracts constraints from the normalised program according to the typing rules (as sketched in Section 2);
3.) *Solving*, the derived constraint system is handed to an optimising constraint solver and the solver output is converted into a type annotation.

In terms of overall resource requirements, the bottleneck of the system is phase three. Preprocessing is both simple and fast. While the code implementing constraint generation might be complex, its execution is fast. All of the underlying complexity is shifted into the third phase. On modern machines with multiple gibibytes of main memory, ATLAS is constrained by the CPU, and not by the available memory. In the remainder of this section, we first detail these phases of ATLAS. We then go into more details of the second phase. Finally, we elaborate the optimisation function which is the key enabler of type inference.

5.1 The Three Phases of ATLAS

1.) Preprocessing. The parser used in the first phase is generated with ANTLR[8] and transformation of the syntax is implemented in Java. The preprocessing performs two tasks: (i) Transformation of the input program into *let-normal-form*, which is the form of program input required by our type system. (ii) The

[8] See antlr.org.

```
LNF[if a<a'
 then (l,a,(leaf,a',r))
 else ((l,a',leaf),a,r)]
```

```
let x1 = a<a' in if x1
 then LNF[(l,a,(leaf,a',r))]
 else LNF[((l,a',leaf),a,r)]
```

```
let x1 = a < a' in if x1
  then let x2 = leaf in let x3 = (x2,a',r) in (l,a,x3)
  else let x4 = leaf in let x5 = (l,a',x4) in (x5,a,r)
```

Fig. 4: Preprocessing: Let Normal Forms.

unsharing conversion creates explicit copies for variables that are used multiple times. Making multiple uses of a variables explicit is required by the let-rule of the type system. In order to satisfy the requirement of the let-rule, it is actually sufficient to track variable usage on the level of program paths. It turns out that in our benchmarks variables are only used multiple times in different branches of an if-statement, for which no unsharing conversion is needed. Hence, we do not discuss the unsharing conversion further in this paper and refer the interested reader to [27] for more details.

Let-normal-form conversion. The let-normal-form conversion is performed recursively and rewrites composed expressions into simple expressions, where each operator is only applied to a variable or a constant. This conversion is achieved by introducing additional let-constructs. We exemplify let-normal-form conversion on a code snippet in Figure 4.

2.) Generation of the Constraint System. After preprocessing, we apply the typing rules. Importantly, the application of all typing rules, except for the weakening rule, which we discuss in further detail below, is *syntax-directed*: This means that each node of the AST of the input program dictates which typing rule is to be applied. The weakening rule could in principle be applied at each AST node, giving the constraint solver more freedom to find a solution. This degree of freedom needs to be controlled by the tool designer. In addition, recall that the suggested implementation of the weakening rule (see Section 4.1) is to be parameterised by the expert knowledge, fed into the weakening rule. In our experiments we noticed that the weakening rule has to be applied sparingly in order to avoid an explosion of the resulting constraint system.

We summarise the degrees of freedom available to the tool designer, which can be specified as parameters to ATLAS on source level. 1.) The selected template potential functions, i.e. the family of indices $\vec{a}, b$ for which coefficients $q_{(\vec{a},b)}$ are generated (we assume not explicitly generated are set to zero). 2.) The number of annotated signatures (with costs and without costs) for each function. 3.) The policy for applying the (parameterised) weakening rule. We detail our choices for instantiating the above degrees of freedom in Section 5.2.

3.) Solving. For solving the generated constraint system, we rely on the Z3 SMT solver. We employ Z3's Java bindings, load Z3 as a shared library, and

exchange constraints for solutions. ATLAS forwards user-supplied configuration to Z3, which allows for flexible tuning of solver parameters. We also record Z3's statistics, most importantly memory usage. During the implementation of ATLAS, Z3's feature to extract unsatisfiable cores has proven valuable. It supplied us with many counterexamples, often directly pinpointing bugs in our implementation. The tool exports constraint systems in SMT-LIB format to the file system. This way, solutions could be cross-checked by re-computing them with other SMT solvers that support minimisation, such as OptiMathSAT [43].

5.2 Details on the Generation of the Constraint System

We now discuss our choices for the aforementioned degrees of freedom.

Potential function templates. Following [27], we create for each node in the AST of the considered input program, where n variables of tree-type are currently in context, the coefficients $q_1, \ldots, q_n$ for the rank functions and the coefficients $q_{(\vec{a},b)}$ for the logarithmic terms, where $\vec{a} \in \{0,1\}^n$ and $b \in \{0,2\}$. This choice turned out to be sufficient in our experiments.

Number of Function Signatures. We fix the number of annotations for each function $f : \alpha_1 \times \cdots \times \alpha_n | Q \to \beta | Q'$ to one regular and one cost-free signature. This was sufficient for our experiments.

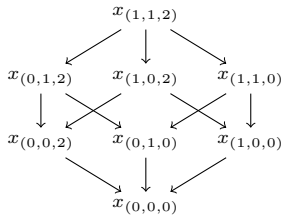

Fig. 5: Monotonicity Lattice for $|Q| = 2$.

Weakening. We need to discharge symbolic comparisons of form $\Phi(\Gamma|P) \leqslant \Phi(\Gamma|Q)$. As indicated in Section 4, we employ Farkas' Lemma to derive constraints for the weakening rule. For context $\Gamma = t_1, \ldots, t_n$, we introduce variables $x_{(\vec{a},b)}$ where $\vec{a} \in \{0,1\}^n, b \in \{0,2\}$, which represent the potential functions $p_{(\vec{a},b)} = \log_2(a_1|t_1| + \ldots + a_n|t_n| + b)$. Next, we explain how the monotonicity of $\log_2$ and Lemma 1 can be used to derive inequalities on the variables $x_{(\vec{a},b)}$, which can then be used to instantiate matrix A in Farkas' Lemma as stated in Section 4.

Monotonicity. We observe that $p_{(\vec{a},b)} = \log_2(a_1|t_1| + \ldots + a_n|t_n| + b) \leqslant \log_2(a'_1|t_1| + \ldots + a'_n|t_n| + b') = p_{(\vec{a}',b')}$, if $a_1 \leqslant a'_1, \ldots, a_n \leqslant a'_n$ and $b \leqslant b'$. This allows us to obtain the lattice shown in Figure 5. A path from $x_{(\vec{a}',b')}$ to $x_{(\vec{a},b)}$ signifies $x_{(\vec{a},b)} \leqslant x_{(\vec{a}',b')}$ resp. $x_{(\vec{a},b)} - x_{(\vec{a}',b')} \leqslant 0$, represented by a row with coefficients 1 and -1 in the corresponding columns of matrix A.

Mathematical facts, like Lemma 1. For an annotated context of length 2, Lemma 1 can be stated by the inequality $2x_{(0,0,2)} + x_{(0,1,0)} + x_{(1,0,0)} - 2x_{(1,1,0)} \leqslant 0$; we add a corresponding row with coefficients $2, 1, 1, -2$ to the matrix A. Likewise, for contexts of length > 2, we add, for each subset of 2 variables, a row with coefficients $2, 1, 1, -2$, setting the coefficients of all other variables to 0.

Sparse expert knowledge matrix. We observe for both kinds of constraints that matrix A is sparse. We exploit this in our implementation and only store non-zero coefficients.

Parametrisation of weakening. Each applications of the weakening rule is parameterised by the matrix A. In our tool, we instantiate A with either the

constraints for (i) monotonicity, shortly referenced as `w{mono}`; (ii) Lemma 1 (`w{l2xy}`); (iii) both (`w{mono l2xy}`); or (iv) none of the constraints (`w`). In the last case, Farkas' Lemma is not needed because weakening defaults to point-wise comparison of the coefficients $p_{(\vec{a},b)}$, which can be implemented more directly. Each time we apply weakening, we need to choose how to instantiate matrix A. Our experiments demonstrate that we need to apply monotonicity and Lemma 1 sparingly in order to avoid blowing up the constraint system.

Tactics and Automation. ATLAS supports manually applying the weakening rule—for this the user has to provide a tactic—and a fully-automated mode.

Naive Automation. Our first attempt to automation applied the weakening rule everywhere instantiated with the full amount of available expert knowledge. This approach did not scale.

Manual Mode via Tactics. A tactic is given as a text file that contains a tree of rule names corresponding to the AST nodes of the input program, into which the user can insert applications of the weakening rule, parameterised by the expert knowledge which should be applied. A simple tactic is depicted in Figure 3. Tactics are distributed with ATLAS, see [32]. The user can name subtrees for reference in the result of the analysis and include ML-style comments in the tactics text. We provide two special commands that allow the user to directly deal with a whole branch of the input program: The question mark (`?`) allows partial proofs; no constraints will be created for the part of the program thus marked. The underscore (`_`) switches to the naive automation of ATLAS and will apply the weakening rule with full expert knowledge everywhere. Both, `?` and `_`, were invaluable when developing and debugging the automated mode. We note that the manual mode still achieves solving times that are by a magnitude faster than the automated mode, which may be of interest to a user willing to hand-optimise solving times.

Automated Mode. For automation, we extracted common patterns from the tactics we developed manually: Weakening with mode `w{mono}` is applied before (var) and (leaf), `w{mono l2xy}` is applied only before (app). (We recall that the full set of rules employed by our analysis can be found in [27].) Further, for AST subtrees that construct trees, i.e. which only consist of (node), (var) and (leaf) rule applications, we apply `w{mono}` for each inner node, and `w{l2xy}` for each outermost node. For all other cases, no weakening is applied. This approach is sufficient to cover all benchmarks, with further improvements possible.

5.3 Optimisation

Given an annotated function $f\colon \alpha_1 \times \cdots \times \alpha_n | Q \to \beta | Q'$, we want to find values for the coefficients of the resource annotations Q and Q' that minimise $\Phi(\Gamma|Q) - \Phi(\Gamma|Q')$, since this difference is an upper bound on the amortised cost of f, cf. Section 4.2. However, as with weakening, we cannot directly express such a minimisation, and again resort to linearisation: We choose an optimisation function that directly maps from Q and Q' to $\mathbb{Q}$. Our optimisation function combines four measures, three of which involve a difference between coefficients

of Q and Q', and a fourth one that only involves coefficients from Q in order to minimise the absolute values of the discovered coefficients. We first present these measures for the special case of $|Q| = 1$.

The first measure $d_1(Q, Q') := q_* - q'_*$ reflects our goal of preserving the coefficient for rk; note that for $d_1(Q, Q') \neq 0$, the resulting complexity bound would be super-logarithmic. The second measure $d_2(Q, Q') := \sum_{(a,b)}(q_{(a,b)} - q'_{(a,b)}) \cdot w(a, b)$ reflects the goal of achieving logarithmic bounds that are as small as possible. Weights are defined to penalise more complex terms, and to exclude constants. (Recall that 1 is representable as $\log_2(0 + 2)$.) We set

$$w(a,b) := \begin{cases} 0, & \text{for } (a,b) = (0,2), \\ (a + (b+1)^2)^2, & \text{otherwise.} \end{cases}$$

The third measure $d_3(Q, Q') := q_{(0,2)} - q'_{(0,2)}$ reflects the goal of minimising constant cost. Lastly, we set $d_4(Q, Q') := \sum_{(a,b)} q_{(a,b)}$ in order to obtain small absolute numbers. The last measure does not influence bounds on the amortised cost, but leads to more beautiful solutions. These measures are then composed to the linear objective function $\min \sum_{i=1}^{4} d_i(Q, Q') \cdot w_i$. In our implementation, we set $w_i = [16127, 997, 97, 2]$; these weights are chosen (almost) arbitrary, we only noticed that w_1 must be sufficiently large to guarantee its priority. (We note that these weights were sufficient for our experiments; we refer to the literature for more principled ways of choosing the weights of an aggregated cost function [34].)

Multiple arguments. For $|Q| > 1$, we set $d_1 := \sum_{i=1}^{|Q|} q_i - q'_*$ and $d_2(Q, Q') := \sum_{(a,a,\ldots,b)}(q_{(a,a,\ldots,b)} - q'_{(a,b)}) \cdot w(a, b)$. The required changes for d_3 and d_4 are straight-forward. In our benchmarks, there is only one function (`merge` of pairing heaps) that requires this minimisation function.

6 Evaluation

We first describe the benchmark functions employed to evaluate ATLAS and then detail this experimental evaluation, already depicted in Table 1.

6.1 Automated Analysis of Splaying et al.

Splay Trees. Introduced by Sleator and Tarjan [47, 49], *splay trees* are self-adjusting binary search trees with strictly increasing in-order traversal, but without an explicit balancing condition. Based on splaying, searching is performed by splaying with the sought element and comparing to the root of the result. Similarly, insertion and deletion are based on splaying. Above we used the zig-zig case of splaying, depicted in Figure 1 as motivating code example. While the pen-and-paper analysis of this case is the most involved, type inference for this case alone did not directly yield the desired automation of the complete definition. Rather, full automation required substantial implementation effort, as detailed in Section 5. As already emphasised, it came as a surprise to us that our tool ATLAS is able match up and partly improve upon the sophisticated

Function	Proof / (w)	automated (naive)		automated (improved)		manual	
`ST.splay` (zig-zig)	Selective	n/a		7718	18S	2552	<1S
	All	11792	45S	9984	19S	2864	<1S
`ST.splay`	Selective	n/a		42095	8M1S	19111	12S
	All	68103 t/o	24H	54377	14M19S	23323	1M27S
`SH.partition`	Selective	n/a		33729	7M9S	15213	6S
	All	51995 t/o	24H	43549	15M2S	16829	10S
`PH.merge_pairs`	Selective	n/a		25860	1M3S	6414	<1S
	All	43515 t/o	24H	34918	13M41S	6558	<1S

(a) Comparison of the number of constraints generated and time taken for the type inference of the core operation of each benchmark plus the zig-zig case of `splay`.

Module	automated			manual		
	Assertions	Time	Memory	Assertions	Time	Memory
`ST`	54794	24M17S	3204	24677	43S	280
`SH`	37911	7M35S	1482	17877	12S	237
`PH`	29493	3M42S	760	7987	1S	29

(b) Number of assertions, solving time and maximum memory usage (in mebibytes) for the combined analysis of functions per-module.

Table 2: Experimental Results

optimisations performed by Schoenmakers [41, 42]. This seems to be evidence of the versatility of the employed potential functions. Further, we leverage the sophistication of our optimisation layer in conjunction with the current power of state-of-the-art constraint solvers, like Z3 [36].

Splay Heaps. To overcome deficiencies of splay trees when implemented functionally, Okasaki introduced *splay heaps*. Splay heaps are defined similarly to splay trees and their (manual) amortised cost analysis follows similar patterns as the one for splay trees. Due to the similarity in the definitions between splay heaps and splay trees, extension of our experimental results in this direction did not pose any problems. Notably, however, ATLAS improves the known complexity bounds on the amortised complexity for the functions studied. We also remark that typical assumptions made in pen-and-paper proofs are automatically discharged by our approach: Schoenmakers [41, 42] as well as Nipkow and Brinkop [37] make use of the (obvious) fact that the size of the resulting tree t' or heap h' equals the size of the input. As discussed, this information is captured by a cost-free derivation, cf. Section 2.

Pairing Heaps. These are another implementation of heaps, which are represented as binary trees, subject to the invariant that they are either `leaf`, or the right child is `leaf`, respectively. The left child is conceivable as list of pairing heaps. Schoenmakers and Nipkow et al. provide a (semi-)manual analysis of pairing heaps, that ATLAS can verify or even improve fully-automatically. We note that we analyse a single function `merge_pairs`, whereas [37] breaks down

the analysis and studies two functions `pass_1` and `pass_2` with `merge_pairs` = `pass_2` ∘ `pass_1`. All definitions can be found at [33].

6.2 Experimental Results

Our main results have already been stated in Table 1 of Section 1. Table 2a compares the differences between the "naive automation" and our actual automation ("automated mode"), see Section 5. Within the latter, we distinguish between a "selective" and a "full" mode. The "selective" mode is as described on page 18. The "full" mode employs weakening for the same rule applications as the "selective" mode, but always with option `w{mono l2xy}`. The same applies to the "full" manual mode. The naive automation does not support selection of expert knowledge. Thus the "selective" option is not available, denoted as "n/a". Timeouts are denoted by "t/o". As depicted in the table, the naive automation does not terminate within 24h for the core operations of the three considered data structures, whereas the improved automated mode produces optimised results within minutes. In Table 2b, we compare the (improved) automated mode with the manual mode, and report on the sizes of the resulting constraint system and on the resources required to produce the same results. Observe that even though our automated mode achieves reasonable solving times, there is still a significant gap between the manually crafted tactics and the automated mode, which invites future work.

7 Conclusion

In this paper we have for the first time been able to automatically conduct an amortised analysis for self-adjusting data structures. Our analysis is based on the "sum of logarithms" potential function and we have been able to automate reasoning about these potential functions by using Farkas' Lemma for the linear part of the calculations and adding necessary facts about the logarithm. Immediate future work is concerned with replacing the "sum of logarithms" potential function in order to analyse skew heaps and Fibonacci heaps [42]. In particular, the potential function for skew heaps, which counts "right heavy" nodes, is interesting, because it is also used as a building block by Iacono in his improved analysis of pairing heaps [29, 30]. Further, we envision to extend our analysis to related probabilistic settings such as priority queues [13] and skip lists [40].

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-form upper bounds in static cost analysis. JAR **46**(2) (2011)
2. Alonso-Blas, D.E., Genaim, S.: On the limits of the classical approach to cost analysis. In: SAS. pp. 405–421 (2012)
3. Avanzini, M., Lago, U.D., Moser, G.: Analysing the complexity of functional programs: higher-order meets first-order. In: ICFP. pp. 152–164. ACM (2015)

4. Avanzini, M., Moser, G.: A combination framework for complexity. IC **248**, 22–55 (2016)
5. Avanzini, M., Moser, G., Schaper, M.: TcT: Tyrolean Complexity Tool. In: TACAS. pp. 407–423. LNCS (2016)
6. Bauer, S., Jost, S., Hofmann, M.: Decidable inequalities over infinite trees. In: LPAR. vol. 57, pp. 111–130 (2018)
7. Brázdil, T., Chatterjee, K., Kucera, A., Novotný, P., Velan, D., Zuleger, F.: Efficient algorithms for asymptotic bounds on termination time in VASS. In: LICS. pp. 185–194 (2018)
8. Chatterjee, K., Fu, H., Goharshady, A.K.: Non-polynomial worst-case analysis of recursive programs. In: CAV. pp. 41–63 (2017)
9. Colcombet, T., Daviaud, L., Zuleger, F.: Size-change abstraction and max-plus automata. In: MFCS. pp. 208–219 (2014)
10. Fiedor, T., Holík, L., Rogalewicz, A., Sinn, M., Vojnar, T., Zuleger, F.: From shapes to amortized complexity. In: VMCAI. pp. 205–225 (2018)
11. Flores-Montoya, A.: Cost Analysis of Programs Based on the Refinement of Cost Relations. Ph.D. thesis, Darmstadt University of Technology, Germany (2017)
12. Fuhs, C., Kop, C., Nishida, N.: Verifying procedural programs via constrained rewriting induction. TOCL **18**(2), 14:1–14:50 (2017)
13. Gambin, A., Malinowski, A.: Randomized meldable priority queues. In: SOFSEM. pp. 344–349 (1998)
14. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Analyzing program termination and complexity automatically with AProVE. JAR **1**, 3–31 (2017)
15. Gulwani, S., Zuleger, F.: The reachability-bound problem. In: PLDI. pp. 292–304 (2010)
16. Harrison, J.: Verifying nonlinear real formulas via sums of squares. In: TPHOLs. pp. 102–118 (2007)
17. Hermenegildo, M., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J., Puebla, G.: An overview of ciao and its design philosophy. TPLP **12**(1-2), 219–252 (2012)
18. Hirokawa, N., Moser, G.: Automated complexity analysis based on the dependency pair method. In: IJCAR. pp. 364–380 (2008)
19. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. In: Proc. 38th POPL. pp. 357–370. ACM (2011)
20. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. TOPLAS **34**(3), 14 (2012)
21. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource aware ML. In: CAV. pp. 781–786 (2012)
22. Hoffmann, J., Das, A., Weng, S.C.: Towards automatic resource bound analysis for OCaml. In: POPL. pp. 359–373 (2017)
23. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: POPL. pp. 185–197 (2003)
24. Hofmann, M., Moser, G.: Amortised resource analysis and typed polynomial interpretations. In: Proc. of Joint 25th RTA and 12th TLCA. pp. 272–286 (2014)
25. Hofmann, M., Moser, G.: Multivariate amortised resource analysis for term rewrite systems. In: TLCA. pp. 241–256 (2015)
26. Hofmann, M., Moser, G.: Analysis of logarithmic amortised complexity (2018)

27. Hofmann, M., Leutgeb, L., Moser, G., Obwaller, D., Zuleger, F.: Type-based analysis of logarithmic amortised complexity. MSCS (2021), to appear, see `https://arxiv.org/abs/2101.12029`.
28. Hofmann, M., Rodriguez, D.: Automatic type inference for amortised heap-space analysis. In: ESOP. pp. 593–613 (2013)
29. Iacono, J.: Improved upper bounds for pairing heaps. In: SWAT. pp. 32–45 (2000)
30. Iacono, J., Yagnatinsky, M.V.: A linear potential function for pairing heaps. In: COCOA. pp. 489–504 (2016)
31. Jost, S., Vasconcelos, P., Florido, M., Hammond, K.: Type-based cost analysis for lazy functional languages. JAR **59**(1), 87–120 (2017)
32. Leutgeb, L.: ATLAS: Automated Amortised Complexity Analysis of Self-Adjusting Data Structures (2021). https://doi.org/10.5281/zenodo.4724917
33. Leutgeb, L.: ATLAS: Examples (2021). https://doi.org/10.5281/zenodo.4880499
34. Marques-Silva, J., Argelich, J., Graça, A., Lynce, I.: Boolean lexicographic optimization: algorithms & applications. Ann. Math. Artif. Intell. **62**(3-4), 317–343 (2011)
35. Moser, G., Schneckenreither, M.: Automated amortised resource analysis for term rewrite systems. Sci. Comput. Program. **185** (2020)
36. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS. pp. 337–340 (2008)
37. Nipkow, T., Brinkop, H.: Amortized complexity verified. JAR **62**(3), 367–391 (2019)
38. Okasaki, C.: Purely functional data structures. Cambridge University Press (1999)
39. Pani, T., Weissenbacher, G., Zuleger, F.: Rely-guarantee reasoning for automated bound analysis of lock-free algorithms. In: FMCAD. pp. 1–9 (2018)
40. Pugh, W.: Skip lists: A probabilistic alternative to balanced trees. Commun. ACM **33**(6), 668–676 (1990)
41. Schoenmakers, B.: A systematic analysis of splaying. IPL **45**(1), 41–50 (1993)
42. Schoenmakers, B.: Data Structures and Amortized Complexity in a Functional Setting. Ph.D. thesis, Eindhoven University of Technology (1992)
43. Sebastiani, R., Trentin, P.: Optimathsat: A tool for optimization modulo theories. In: CAV. pp. 447–454 (2015)
44. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. In: CAV. pp. 745–761 (2014)
45. Sinn, M., Zuleger, F., Veith, H.: Difference constraints: An adequate abstraction for complexity analysis of imperative programs. In: Kaivola, R., Wahl, T. (eds.) FMCAD. pp. 144–151. IEEE (2015)
46. Sinn, M., Zuleger, F., Veith, H.: Complexity and resource bound analysis of imperative programs using difference constraints. JAR **59**(1), 3–45 (2017)
47. Sleator, D., Tarjan, R.: Self-adjusting binary search trees. JACM **32**(3), 652–686 (1985)
48. Solar-Lezama, A.: The sketching approach to program synthesis. In: APLAS. pp. 4–13 (2009)
49. Tarjan, R.: Amortized computational complexity. SIAM J. Alg. Disc. Meth **6**(2), 306–318 (1985)
50. Wang, P., Wang, D., Chlipala, A.: TiML: A functional language for practical complexity analysis with invariants. Proc. ACM Program. Lang. **1**(OOPSLA) (2017)
51. Winkler, S., Moser, G.: Runtime complexity analysis of logically constrained rewriting. In: Proc. LOPSTR 2020 (2020)
52. Zuleger, F.: The polynomial complexity of vector addition systems with states. In: FOSSACS. pp. 622–641 (2020)