Automated Expected Amortised Cost Analysis of Probabilistic Data Structures

Lorenz Leutgeb[1](✉), Georg Moser[2], and Florian Zuleger[3]

[1] Max Planck Institute for Informatics and Graduate School of Computer Science, Saarland Informatics Campus, Saarbrücken, Germany
`lorenz@mpi-inf.mpg.de`
[2] Department of Computer Science, Universität Innsbruck, Innsbruck, Austria
[3] Institute of Logic and Computation 192/4, Technische Universität Wien, Vienna, Austria

Abstract. In this paper, we present the first fully-automated *expected amortised cost analysis* of self-adjusting data structures, that is, of *randomised splay trees*, *randomised splay heaps* and *randomised meldable heaps*, which so far have only (semi-)manually been analysed in the literature. Our analysis is stated as a type-and-effect system for a first-order functional programming language with support for sampling over discrete distributions, non-deterministic choice and a ticking operator. The latter allows for the specification of fine-grained cost models. We state two soundness theorems based on two different—but strongly related—typing rules of ticking, which account differently for the cost of non-terminating computations. Finally we provide a prototype implementation able to fully automatically analyse the aforementioned case studies.

Keywords: amortised cost analysis · functional programming · probabilistic data structures · automation · constraint solving

1 Introduction

Probabilistic variants of well-known computational models such as automata, Turing machines or the λ-calculus have been studied since the early days of computer science (see [16,17,25] for early references). One of the main reasons for considering probabilistic models is that they often allow for the design of more efficient algorithms than their deterministic counterparts (see e.g. [6,23,25]). Another avenue for the design of efficient algorithms has been opened up by Sleator and Tarjan [34,36] with their introduction of the notion of *amortised complexity*. Here, the cost of a single data structure operation is not analysed in isolation but as part of a sequence of data structure operations. This allows for the design of algorithms where the cost of an expensive operation is averaged out over multiple operations and results in a good overall *worst-case cost*. Both methodologies—*probabilistic programming* and *amortised complexity*—can

S. Shoham and Y. Vizel (Eds.): CAV 2022, LNCS 13372, pp. 70–91, 2022.
https://doi.org/10.1007/978-3-031-13188-2_4

be combined for the design of even more efficient algorithms, as for example in *randomized splay trees* [1], where a rotation in the splaying operation is only performed with some probability (which improves the overall performance by skipping some rotations while still guaranteeing that enough rotations are performed).

In this paper, we present the first fully-automated *expected amortised cost analysis* of probabilistic data structures, that is, of *randomised splay trees*, *randomised splay heaps*, *randomised meldable heaps* and a *randomised analysis* of a *binary search tree*. These data structures have so far only (semi-)manually been analysed in the literature. Our analysis is based on a novel type-and-effect system, which constitutes a generalisation of the type system studied in [14,18] to the non-deterministic and probabilistic setting, as well as an extension of the type system introduced in [37] to sublinear bounds and non-determinism. We provide a prototype implementation that is able to fully automatically analyse the case studies mentioned above. We summarise here the main contributions of our article: (i) We consider a first-order functional programming language with support for *sampling over discrete distributions*, *non-deterministic choice* and a *ticking* operator, which allows for the specification of fine-grained cost models. (ii) We introduce compact *small-step* as well as *big-step* semantics for our programming language. These semantics are equivalent wrt. the obtained normal forms (i.e., the resulting probability distributions) but differ wrt. the cost assigned to non-terminating computations. (iii) Based on [14,18], we develop a novel type-and-effect system that strictly generalises the prior approaches from the literature. (iv) We state two soundness theorems (see Sect. 5.3) based on two different—but strongly related—typing rules of ticking. The two soundness theorems are stated wrt. the small-step resp. big-step semantics because these semantics precisely correspond to the respective ticking rule. The more restrictive ticking rule can be used to establish (positive) almost sure termination (AST), while the more permissive ticking rule supports the analysis of a larger set of programs (which can be very useful in case termination is not required or can be established by other means); in fact, the more permissive ticking rule is essential for the precise cost analysis of randomised splay trees. We note that the two ticking rules and corresponding soundness theorems do not depend on the details of the type-and-effect system, and we believe that they will be of independent interest (e.g., when adapting the framework of this paper to other benchmarks and cost functions). (v) Our prototype implementation ATLAS strictly extends the earlier version reported on in [18], and all our earlier evaluation results can be replicated (and sometimes improved).

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

1. *We automatically infer bounds on the expected amortised cost, which could previously only be obtained by sophisticated pen-and-paper proofs. In particular, we verify that the amortised costs of randomised variants of self-adjusting data structures improve upon their non-randomised variants.* In Table 1 we state the expected cost of the randomised data structures considered and their deterministic counterparts; the benchmarks are detailed in Sect. 2.

Table 1. Expected Amortised Cost of Randomised Data Structures. We also state the deterministic counterparts considered in [18] for comparison.

	probabilistic	deterministic [18]
	Splay Tree	
`insert`	$3/4 \log_2(\|t\|) + 3/4 \log_2(\|t\| + 1)$	$2 \log_2(\|t\|) + 3/2$
`delete`	$9/8 \log_2(\|t\|)$	$5/2 \log_2(\|t\|) + 3$
`splay`	$9/8 \log_2(\|t\|)$	$3/2 \log_2(\|t\|)$
	Splay Heap	
`insert`	$3/4 \log_2(\|h\|) + 3/4 \log_2(\|h\| + 1)$	$1/2 \log_2(\|h\|) + \log_2(\|h\| + 1) + 3/2$
`delete_min`	$3/4 \log_2(\|h\|)$	$\log_2(\|h\|)$
	Meldable Heap	
`insert`	$\log_2(\|h\|) + 1$	*not applicable*
`delete_min`	$2 \log_2(\|h\|)$	
`meld`	$\log_2(\|h_1\|) + \log_2(\|h_2\|)$	
	Coin Search Tree	
`insert`	$3/2 \log_2(\|t\|) + 1/2$	*not applicable*
`delete`	$3/2 \log_2(\|t\|) + 1$	
`delete_max`	$3/2 \log_2(\|t\|)$	

2. *We establish a novel approach to the expected cost analysis of data structures.* Our research has been greatly motivated by the detailed study of Albers et al. in [1] of the expected amortised costs of *randomised splaying*. While [1] requires a sophisticated pen-and-paper analysis, our approach allows us to fully-automatically compare the effect of different rotation probabilities on the expected cost (see Table 2 of Sect. 6).

Related Work. The generalisation of the model of computation and the study of the expected resource usage of *probabilistic* programs has recently received increased attention (see e.g. [2,4,5,7,10,11,15,21,22,24,27,37,38]). We focus on related work concerned with automations of expected cost analysis of deterministic or non-deterministic, probabilistic programs—imperative or functional. (A probabilistic program is called *non-deterministic*, if it additionally makes use of non-deterministic choice.)

In recent years the *automation* of expected cost analysis of probabilistic data structures or programs has gained momentum, cf. [2–5,22,24,27,37,38]. Notably, the Absynth prototype by [27], implement Kaminski's ert-calculus, cf. [15] for reasoning about expected costs. Avanzini et al. [5] generalise the ert-calculus to an expected cost transformer and introduce the tool eco-imp, which provides a modular and thus a more efficient and scalable alternative for non-deterministic, probabilistic programs. In comparison to these works, we base our analysis on a dedicated type system finetuned to express sublinear bounds; further our prototype implementation ATLAS derives bounds on the expected amortised costs. Neither is supported by Absynth or eco-imp.

Martingale based techniques have been implemented, e.g., by Peixin Wang et al. [38]. Related results have been reported by Moosbrugger et al. [24]. Meyer et al. [22] provide an extension of the KoAT tool, generalising the concept of alternating size and runtime analysis to probabilistic programs. Again, these innovative tools are not suited to the benchmarks considered in our work. With respect to probabilistic *functional* programs, Di Wang et al. [37] provided the only prior expected cost analysis of (deterministic) probabilistic programs; this work is most closely related to our contributions. Indeed, our typing rule (ite : coin) stems from [37] and the soundness proof wrt. the big-step semantics is conceptually similar. Nevertheless, our contributions strictly generalise their results. First, our core language is based on a simpler semantics, giving rise to cleaner formulations of our soundness theorems. Second, our type-and-effect provides two different typing rules for ticking, a fact we can capitalise on in additional strength of our prototype implementation. Finally, our amortised analysis allows for logarithmic potential functions.

A bulk of research concentrates on specific forms of *martingales* or *Lyapunov ranking functions*. All these works, however, are somewhat orthogonal to our contributions, as foremostly *termination* (i.e. AST or PAST) is studied, rather than computational complexity. Still these approaches can be partially suited to a variety of quantitative program properties, see [35] for an overview, but are incomparable in strength to the results established here.

Structure. In the next section, we provide a bird's eye view on our approach. Sections 3 and 4 detail the core probabilistic language employed, as well as its small- and big-step semantics. In Sect. 5 we introduce the novel type-and-effect system formally and state soundness of the system wrt. the respective semantics. In Sect. 6 we present evaluation results of our prototype implementation ATLAS. Finally, we conclude in Sect. 7. All proofs, part of the benchmarks and the source codes are given in [19].

2 Overview of Our Approach and Results

In this section, we first sketch our approach on an introductory example and then detail the benchmarks and results depicted in Table 1 in the Introduction.

2.1 Introductory Example

Consider the definition of the function `descend`, depicted in Fig. 1. The *expected* amortised complexity of `descend` is $\log_2(|t|)$, where $|t|$ denotes the size of a tree t (defined as the number of leaves of the tree).[1] Our analysis is set up in terms of template potential functions with unknown coefficients, which will be instantiated by our analysis. Following [14,18], our potential functions are composed of two types of resource functions, which can express *logarithmic* amortised cost: For a sequence of n trees $t_1, \dots, t_n$ and coefficients $a_i \in \mathbb{N}, b \in \mathbb{Z}$, with $\sum_{i=1}^{n} a_i + b \geqslant$

[1] An amortised analysis may always default to a wort-case analysis. In particular the analysis of `descend` in this section can be considered as a worst-case analysis. However, we use the example to illustrate the general setup of our amortised analysis.

```
descend t = match t with
  | leaf        → leaf
  | node l a r  → if coin 1/2   Denotes p = 1/2, the default which could be omitted.
    then let xl = (descend l)✓  in node xl a r  The symbol ✓ denotes a tick.
    else let xr = (descend r)✓  in node l a xr
```

Fig. 1. `descend` function

0, the resource function $p_{(a_1,\ldots,a_n,b)}(t_1,\ldots,t_n)\log_2(a_1\cdot|t_1|+\cdots+a_n\cdot|t_n|+b)$ denotes the logarithm of a linear combination of the sizes of the trees. The resource function $\mathsf{rk}(t)$, which is a variant of Schoenmakers' potential, cf. [28,31,32], is inductively defined as (i) $\mathsf{rk}(\texttt{leaf}) := 1$; (ii) $\mathsf{rk}(\texttt{node}\ l\ d\ r) := \mathsf{rk}(l) + \log_2(|l|) + \log_2(|r|) + \mathsf{rk}(r)$, where l, r are the left resp. right child of the tree `node` $l\ d\ r$, and d is some data element that is ignored by the resource function. (We note that $\mathsf{rk}(t)$ is not needed for the analysis of `descend` but is needed for more involved benchmarks, e.g. randomised splay trees.) With these resource functions at hand, our analysis introduces the coefficients q_*, $q_{(1,0)}$, $q_{(0,2)}$, q'_*, $q'_{(1,0)}$, $q'_{(0,2)}$ and employs the following *Ansatz*:[2]

$$q_*\cdot\mathsf{rk}(t)+q_{(1,0)}\cdot p_{(1,0)}(t)+q_{(0,2)}\cdot p_{(0,2)}(t)\geqslant c_{\texttt{descend}}(t)$$
$$+\,q'_*\,\mathsf{rk}(\texttt{descend}\ t)+q'_{(1,0)}\cdot p_{(1,0)}(\texttt{descend}\ t)+q'_{(0,2)}\cdot p_{(0,2)}(\texttt{descend}\ t)\ .$$

Here, $c_{\texttt{descend}}(t)$ denotes the expected cost of executing `descend` on tree t, where the cost is given by the ticks as indicated in the source code (each tick accounts for a recursive call). The result of our analysis will be an instantiation of the coefficients, returning $q_{(1,0)} = 1$ and zero for all other coefficients, which allows to directly read off the logarithmic bound $\log_2(|t|)$ of `descend`.

Our analysis is formulated as a *type-and-effect system*, introducing the above *template potential functions* for every subexpression of the program under analysis. The typing rules of our system give rise to a constraint system over the unknown coefficients that capture the relationship between the potential functions of the subexpressions of the program. Solving the constraint system then gives a valid instantiation of the potential function coefficients. Our type-and-effect system constitutes a generalisation of the type system studied in [14,18] to the non-deterministic and probabilistic setting, as well as an extension of the type system introduced in [37] to sublinear bounds and non-determinism.

In the following, we survey our type-and-effect system by means of example `descend`. A partial type derivation is given in Fig. 2. 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 [19]. We now discuss this type derivation step by step.

Let e denote the body of the function definition of `descend`, cf. Fig. 1. Our automated analysis infers an *annotated type* by verifying that the type

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

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\texttt{descend}\colon \mathsf{T}|Q \to \mathsf{T}|Q'}{l\colon \mathsf{T}|Q_5 \vdash \texttt{descend l}\colon \mathsf{T}|Q_6}\;(\mathsf{app})}{l\colon \mathsf{T}|Q_4 \vdash (\texttt{descend l})^{\checkmark}\colon \mathsf{T}|Q_6}\;(\mathsf{tick : now}) \qquad x_l\colon \mathsf{T}, r\colon \mathsf{T}|Q_7 \vdash \texttt{node}\ x_l\ a\ r\colon \mathsf{T}|Q'}{l\colon \mathsf{T}, r\colon \mathsf{T}|Q_3 \vdash \texttt{let}\ x_l = (\texttt{descend l})^{\checkmark}\ \texttt{in node}\ x_l\ a\ r\colon \mathsf{T}|Q'}\;(\mathsf{let})}{l\colon \mathsf{T}, r\colon \mathsf{T}|Q_2 \vdash \texttt{if coin 1/2 then}\ e_2\ \texttt{else}\ e_3\colon \mathsf{T}|Q'}\;(\mathsf{ite : coin})}{l\colon \mathsf{T}, r\colon \mathsf{T}|Q_1 \vdash \texttt{if coin 1/2 then}\ e_2\ \texttt{else}\ e_3\colon \mathsf{T}|Q'}\;(\mathsf{w})}{t\colon \mathsf{T}|Q \vdash \texttt{match}\ t\ \texttt{with|leaf}\ \to\ \texttt{leaf |node}\ l\ a\ r\ \to\ e_1\colon \mathsf{T}|Q'}\;(\mathsf{match})
$$

Fig. 2. Partial Type Derivation for Function `descend`

judgement $t\colon \mathsf{T}|Q \vdash e\colon \mathsf{T}|Q'$ is derivable. 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 `descend` and its results. Annotations fix the coefficients of the resource functions in the corresponding potential functions, e.g., (i) $\Phi(t\colon \mathsf{T}|Q) := q_* \cdot \mathsf{rk}(t) + q_{(1,0)} \cdot p_{(1,0)}(t) + q_{(0,2)} \cdot p_{(0,2)}(t)$ and (ii) $\Phi(e\colon \mathsf{T}|Q') := q'_* \cdot \mathsf{rk}(e) + q'_{(1,0)} \cdot p_{(1,0)}(e) + q'_{(0,2)} \cdot p_{(0,2)}(e)$.

By our soundness theorems (see Sect. 5.3), such a typing guarantees that the *expected* amortised cost of `descend` is bounded by the expectation (wrt. the value distribution in the limit) of the difference between $\Phi(t\colon \mathsf{T}|Q)$ and $\Phi(\texttt{descend}\ t\colon \mathsf{T}|Q')$. Because e is a `match` expression, the following rule is applied (we only state a restricted rule here, the general rule can be found in [19]):

$$
\dfrac{\varepsilon|\varnothing \vdash \texttt{leaf}\colon \mathsf{T}|Q' \qquad l\colon \mathsf{T}, r\colon \mathsf{T}|Q_1 \vdash e_1\colon \mathsf{T}|Q'}{t\colon \mathsf{T}|Q \vdash \texttt{match}\ t\ \texttt{with|leaf}\ \to\ \texttt{leaf |node}\ l\ a\ r\ \to\ e_1\colon \mathsf{T}|Q'}\;(\mathsf{match})
$$

Here e_1 denotes the subexpression of e that corresponds to the `node` case of `match`. Apart from the annotations Q, Q_1 and Q', the rule (match) constitutes a standard type rule for pattern matching. With regard to the annotations Q and Q_1, (match) ensures the correct distribution of potential by inducing the constraints

$$
q^1_1 = q^1_2 = q_* \qquad q^1_{(1,1,0)} = q_{(1,0)} \qquad q^1_{(1,0,0)} = q^1_{(0,1,0)} = q_* \qquad q^1_{(0,0,2)} = q_{(0,2)}\ ,
$$

where the constraints are immediately justified by recalling the definitions of the resource functions $p_{(a_1,\dots,a_n,b)}(t_1,\dots,t_n) := \log_2(a_1 \cdot |t_1| + \cdots + a_n \cdot |t_n| + b)$ and $\mathsf{rk}(t) = \mathsf{rk}(l) + \log_2(|l|) + \log_2(|r|) + \mathsf{rk}(r)$.

The next rule is a structural rule, representing a *weakening* step that rewrites the annotations of the variable context. The rule (w) allows a suitable adaptation of the coefficients based on the following inequality, which holds for any substitution σ of variables by values, $\Phi(\sigma; l\colon \mathsf{T}, r\colon \mathsf{T}|Q_1) \geqslant \Phi(\sigma; l\colon \mathsf{T}, r\colon \mathsf{T}|Q_2)$.

$$
\dfrac{l\colon \mathsf{T}, r\colon \mathsf{T}|Q_2 \vdash e_1\colon \mathsf{T}|Q'}{l\colon \mathsf{T}, r\colon \mathsf{T}|Q_1 \vdash e_1\colon \mathsf{T}|Q'}\;(\mathsf{w})
$$

In our prototype implementation this comparison is performed *symbolically*. We use a variant of Farkas' Lemma [19,33] in conjunction with simple

```
meld h1 h2 = match h1 with
  | leaf              → h2
  | node h1l h1x h1r  → match h2 with
    | node h2l h2x h2r  → if h1x > h2x
      then if coin
        then (node (meld h2l (node h1l h1x h1r))✓ h2x h2r)
        else (node h2l h2x (meld h2r (node h1l h1x h1r))✓ )
      else  Omitted for brevity,  symmetric to the the depicted  case.
```

Fig. 3. Partial `meld` function of Randomised Meldable Heaps

mathematical 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, Farkas' Lemma in conjunction with the latter fact gives rise to

$$q^1_{(0,0,2)} + 2f \geqslant q^2_{(0,0,2)} \qquad q^1_{(1,1,0)} - 2f \geqslant q^2_{(1,1,0)}$$
$$q^1_{(1,0,0)} + f \geqslant q^2_{(1,0,0)} \qquad q^1_{(0,1,0)} + f \geqslant q^2_{(0,1,0)} ,$$

for some fresh rational coefficient $f \geqslant 0$ introduced by Farkas' Lemma. After having generated the constraint system for `descend`, the solver is free to instantiate f as needed. In fact in order to discover the bound $\log_2(|t|)$ for `descend`, the solver will need to instantiate $f = 1/2$, corresponding to the inequality $\log_2(|l|+|r|) \geqslant 1/2 \log_2(|l|) + 1/2 \log_2(|r|) + 1$.

So far, the rules did not refer to sampling and are unchanged from their (non-probabilistic) counterpart introduced in [14,18]. The next rule, however, formalises a coin toss, biased with probability p. Our general rule (ite : coin) is depicted in Fig. 12 and is inspired by a similar rule for coin tosses that has been recently been proposed in the literature, cf. [37]. This rule specialises as follows to our introductory example:

$$\frac{\begin{array}{l} l\colon\mathsf{T}, r\colon\mathsf{T}|Q_4 \vdash e_3 \colon \mathsf{T}|Q' \\ l\colon\mathsf{T}, r\colon\mathsf{T}|Q_3 \vdash \mathtt{let}\ x_l = (\mathtt{descend}\ l)^{\checkmark}\ \mathtt{in}\ \mathtt{node}\ x_l\ a\ r \colon \mathsf{T}|Q' \end{array}}{l\colon\mathsf{T}, r\colon\mathsf{T}|Q_2 \vdash \mathtt{if}\ \mathtt{coin}\ \mathtt{1/2}\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \colon \mathsf{T}|Q'} \ (\mathsf{ite} : \mathsf{coin})$$

Here e_2 and e_3 respectively, denote the subexpressions of the conditional and in addition the crucial condition $Q_2 = 1/2 \cdot Q_3 + 1/2 \cdot Q_4$ holds. This condition, expressing that the corresponding annotations are subject to the probability of the coin toss, gives rise to the following constraints (among others)

$$q^2_{(0,0,2)} = 1/2 \cdot q^3_{(0,0,2)} + 1/2 \cdot q^4_{(0,0,2)} \qquad q^2_{(0,1,0)} = 1/2 \cdot q^3_{(0,1,0)} + 1/2 \cdot q^4_{(0,1,0)}$$
$$q^2_{(1,0,0)} = 1/2 \cdot q^3_{(1,0,0)} + 1/2 \cdot q^4_{(1,0,0)} .$$

In the following, we will only consider one alternative of the coin toss and proceed as in the partial type derivation depicted in Fig. 1 (ie. we state the `then`-branch and omit the symmetric `else`-branch). Thus next, we apply the rule for the `let` expression. This rule is the most involved typing rule in the system proposed in [14,18]. However, for our leading example it suffices to consider the following simplified variant:

```
splay a t = match t with
  | node cl c cr → match cl with
    | node bl b br → match (splay a bl)✓1/2 with   Recursive call costs 1/2.
      | node al a1 ar → if coin
    then (node al a1 (node ar b (node br c cr)))✓1/2   Rotation costs 1/2.
    else        node (node (node al a1 ar) b br) c cr    No rotation.
```

Fig. 4. Partial splay function of Randomised Splay Trees (zigzig-case)

$$\frac{l{:}\,\mathsf{T}|Q_4 \vdash (\mathtt{descend}\ l)^{\checkmark} : \mathsf{T}|Q_6 \qquad l{:}\,\mathsf{T}|Q_7 \vdash \mathtt{node}\ x_l\ a\ r : \mathsf{T}|Q'}{l{:}\,\mathsf{T}, r{:}\,\mathsf{T}|Q_3 \vdash \mathtt{let}\ x_l = (\mathtt{descend}\ l)^{\checkmark}\ \mathtt{in}\ \mathtt{node}\ x_l\ a\ r : \mathsf{T}|Q'}\ (\mathsf{let})$$

Focusing on the annotations, the rule (let : tree) suitably distributes potential assigned to the variable context, embodied in the annotation Q_3, to the recursive call within the let expression (via annotation Q_4) and the construction of the resulting tree (via annotation Q_7). The distribution of potential is facilitated by generating constraints that can roughly be stated as two “equalities”, that is, (i) “$Q_3 = Q_4 + D$”, and (ii) “$Q_7 = D + Q_6$”. Equality (i) states that the input potential is split into some potential Q_4 used for typing $(\mathtt{descend}\ l)^{\checkmark}$ and some remainder potential D (which however is not constructed explicitly and only serves as a placeholder for potential that will be passed on). Equality (ii) states that the potential Q_7 used for typing node x_l a r equals the remainder potential D plus the leftover potential Q_6 from the typing of $(\mathtt{descend}\ l)^{\checkmark}$. The (tick : now) rule then ensures that costs are properly accounted for by generating constraints for $Q_4 = Q_5 + 1$ (see Fig. 2). Finally, the type derivation ends by the application rule, denoted as (app), that verifies that the recursive call is well-typed wrt. the (annotated) signature of the function descend: $\mathsf{T}|Q \to \mathsf{T}|Q'$, ie. the rule enforces that $Q_5 = Q$ and $Q_6 = Q'$. We illustrate (a subset of) the constraints induced by (let), (tick : now) and (app):

$$\begin{array}{cccc}
q^3_{(1,0,0)} = q^4_{(1,0)} & q^3_{(0,1,0)} = q^7_{(0,1,0)} & q'_1 = q^6_1 & q^4_{(0,2)} = q^5_{(0,2)} + 1 \\
q^3_{(0,0,2)} = q^4_{(0,2)} & q^3_2 = q^7_2 & q'_{(1,0)} = q^6_{(1,0)} & q^4_{(1,0)} = q^5_{(1,0)} \\
q^3_1 = q^4_1 & q'_{(0,2)} = q^6_{(0,2)} & q^6_1 = q^7_1 & q^5_{(1,0)} = q_{(1,0)}\ ,
\end{array}$$

where (i) the constraints in the first three columns—involving the annotations Q_3, Q_4, Q_6 and Q_7—stem from the constraints of the rule (let : tree); (ii) the constraints in the last column—involving Q_4, Q_5, Q and Q'—stem from the constraints of the rule (tick : now) and (app). For example, $q^3_{(1,0,0)} = q^4_{(1,0)}$ and $q^3_{(0,1,0)} = q^7_{(0,1,0)}$ distributes the part of the logarithmic potential represented by Q_3 to Q_4 and Q_7; $q^6_1 = q^7_1$ expresses that the rank of the result of evaluating the recursive call can be employed in the construction of the resulting tree node x_l a r; $q^4_{(1,0)} = q^5_{(1,0)}$ and $q^4_{(0,2)} = q^5_{(0,2)} + 1$ relate the logarithmic resp. constant potential according to the tick rule, where the addition of one accounts for the cost embodied by the tick rule; $q^5_{(1,0)} = q_{(1,0)}$ stipulates that the potential at the recursive call site must match the function type.

Our prototype implementation ATLAS collects all these constraints and solves them fully automatically. Following [14,18], our implementation in fact searches

```
insert d t = match t with
  | leaf       → node leaf d leaf
  | node l a r → if coin 1/2         Assuming probability 1/2 for a < d.
    then node (insert d l)✓ a r
    else node l a (insert d r)✓
```

Fig. 5. `insert` function of a Binary Search Tree with randomized comparison

for a solution that minimises the resulting complexity bound. For the `descend` function, our implementation finds a solution that sets $q_{(1,0)}$ to 1, and all other coefficients to zero. Thus, the logarithmic bound $\log_2(|t|)$ follows.

2.2 Overview of Benchmarks and Results

Randomised Meldable Heaps. Gambin et al. [13] proposed meldable heaps as a simple priority-queue data structure that is guaranteed to have expected logarithmic cost for all operations. All operations can be implemented in terms of the `meld` function, which takes two heaps and returns a single heap as a result. The partial source code of `meld` is given in Fig. 3 (the full source code of all examples can be found in [19]). Our tool ATLAS fully-automatically infers the bound $\log_2(|h_1|) + \log_2(|h_2|)$ on the expected cost of `meld`.

Randomised Splay Trees. Albers et al. in [1] propose these splay trees as a variation of deterministic splay trees [34], which have better expected runtime complexity (the same computational complexity in the O-notation but with smaller constants). Related results have been obtained by Fürer [12]. The proposal is based on the observation that it is not necessary to rotate the tree in every (recursive) splaying operation but that it suffices to perform rotations with some fixed positive probability in order to reap the asymptotic benefits of self-adjusting search trees. The theoretical analysis of randomised splay trees [1] starts by refining the cost model of [34], which simply counts the number of rotations, into one that accounts for recursive calls with a cost of c and for rotations with a cost of d.

We present a snippet of a functional implementation of randomised splay trees in Fig. 4. We note that in this code snippet we have set $c = d = 1/2$; this choice is arbitrary; we have chosen these costs in order to be able to compare the resulting amortised costs to the deterministic setting of [18], where the combined cost of the recursive call and rotation is set to 1; we note that our analysis requires fixed costs c and d but these constants can be chosen by the user; for example one can set $c = 1$ and $d = 2.75$ corresponding to the costs observed during the experiments in [1]. Likewise the probability of the coin toss has been arbitrarily set to $p = 1/2$ but could be set differently by the user. (We remark that to the best of our knowledge no theoretical analysis has been conducted on how to chose the best value of p for given costs c and d.) Our prototype implementation is able to fully automatically infer an amortised complexity bound of $9/8 \log_2(|t|)$ for `splay` (with c, d and p fixed as above), which improves on the complexity

```
pre−condition: t is not a leaf
delete_max t = match t with
  | node l b r → match r with
    | leaf          → (l,b)
    | node rl c rr → match rr with
      | leaf → ((node l b rl),c)
      | rr    → let (t',max) = (delete_max rr)✓ in match t' with
        | node rrl1 x xa → (node (node (node l b rl) c rrl1) x xa,max)
```

Fig. 6. `delete_max` function of a Coin Search Tree with one rotation

bound of $3/2 \log_2(|t|)$ for the deterministic version of `splay` as reported in [18], confirming that randomisation indeed improves the expected runtime.

We remark on how the amortised complexity bound of $9/8 \log_2(|t|)$ for `splay` is computed by our analysis. Our tool ATLAS computes an annotated type for `splay` that corresponds to the inequality

$$3/4\,\mathsf{rk}(t) + 9/8 \log_2(|t|) + 3/4 \geqslant c_{\texttt{splay}}(t) + 3/4\,\mathsf{rk}(\texttt{splay}\ t) + 3/4\ .$$

By setting $\phi(t) := 3/4\,\mathsf{rk}(t) + 3/4$ as potential function in the sense of Tarjan and Sleator [34,36], the above inequality allows us to directly read out an upper bound on the amortised complexity $a_{\texttt{splay}}(t)$ of `splay` (we recall that the amortised complexity in the sense of Tarjan and Sleator is defined as the sum of the actual costs plus the output potential minus the input potential):

$$a_{\texttt{splay}}(t) = c_{\texttt{splay}}(t) + \phi(\texttt{splay}\ t) - \phi(t) \leqslant 9/8 \cdot \log_2(|t|)\ .$$

Probabilistic Analysis of Binary Search Trees. We present a probabilistic analysis of a deterministic binary search tree, which offers the usual `contains`, `insert`, and `delete` operations, where `delete` uses `delete_max` given in Fig. 6, as a subroutine (the source code of the missing operations is given in [19]). We assume that the elements inserted, deleted and searched for are equally distributed; hence, we conduct a probabilistic analysis by replacing every comparison with a coin toss of probability one half. We will refer to the resulting data structure as Coin Search Tree in our benchmarks. The source code of `insert` is given in Fig. 5.

Our tool ATLAS infers an logarithmic expected amortised cost for all operations, ie., for `insert` and `delete_max` we obtain (i) $1/2\,\mathsf{rk}(t) + 3/2 \log_2(|t|) + 3/2 \geqslant c_{\texttt{insert}}(t) + 1/2\,\mathsf{rk}(\texttt{insert}\ t) + 1$; and (ii) $1/2\,\mathsf{rk}(t) + 3/2 \log_2(|t|) + 1 \geqslant c_{\texttt{delete\_max}}(t) + 1/2\,\mathsf{rk}(\texttt{delete\_max}\ t) + 1$, from which we obtain an expected amortised cost of $3/2 \log_2(|t|) + 1/2$ and $3/2 \log_2(|t|)$ respectively.

3 Probabilistic Functional Language

Preliminaries. Let $\mathbb{R}_0^+$ denote the non-negative reals and $\mathbb{R}_0^{+\infty}$ their extension by ∞. We are only concerned with *discrete distributions* and drop "discrete" in the following. Let A be a countable set and let $\mathsf{D}(A)$ denote the set of *(sub)distributions* d over A, whose support $\mathsf{supp}(\mu) := \{a \in A \mid \mu(a) \neq 0\}$ is countable. Distributions are denoted by Greek letters. For $\mu \in \mathsf{D}(A)$, we may

$$
\begin{array}{llll}
\circ ::= \texttt{<} \mid \texttt{>} \mid \texttt{=} & & & \\
e ::= f\ x_1\ \ldots\ x_n & & & \mid e^{\checkmark\, a/b} \\
 & \mid \texttt{false} \mid \texttt{true} \mid e_1 \circ e_2 & & \mid \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
 & & & \mid \texttt{if nondet then}\ e_1\ \texttt{else}\ e_2 \\
 & & & \mid \texttt{if coin}\ a/b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
 & \mid \texttt{leaf} \mid \texttt{node}\ x_1\ x_2\ x_3 & & \mid \texttt{match}\ x\ \texttt{with}\ \texttt{|}\ \texttt{leaf} \to e_1\ \texttt{|}\ \texttt{node}\ x_1\ x_2\ x_3 \to e_2 \\
 & \mid \texttt{(}\ x_1\ \texttt{,}\ x_2\ \texttt{)} & & \mid \texttt{match}\ x\ \texttt{with}\ \texttt{|}\ \texttt{(}\ x_1\ \texttt{,}\ x_2\ \texttt{)} \to e \\
 & \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 & & \mid x
\end{array}
$$

Fig. 7. A Core Probabilistic (First-Order) Programming Language

write $\mu = \{a_i^{p_i}\}_{i \in I}$, assigning probabilities p_i to $a_i \in A$ for every $i \in I$, where I is a suitable chosen index set. We set $|\mu| := \sum_{i \in I} p_i$. If the support is finite, we simply write $\mu = \{a_1^{p_1}, \ldots, a_n^{p_n}\}$ The *expected value* of a function $f : A \to \mathbb{R}_0^+$ on $\mu \in \mathsf{D}(A)$ is defined as $\mathbb{E}_\mu(f) := \sum_{a \in \mathsf{supp}(\mu)} \mu(a) \cdot f(a)$. Further, we denote by $\sum_{i \in I} p_i \cdot \mu_i$ the *convex combination of distributions* μ_i, where $\sum_{i \in I} p_i \leqslant 1$. As by assumption $\sum_{i \in I} p_i \leqslant 1$, $\sum_{i \in I} p_i \cdot \mu_i$ is always a (sub-)distribution.

In the following, we also employ a slight extension of (discrete) distributions, dubbed *multidistributions* [4]. Multidistributions are countable *multisets* $\{a_i^{p_i}\}_{i \in I}$ over pairs $p_i : a_i$ of *probabilities* $0 < p_i \leqslant 1$ and *objects* $a_i \in A$ with $\sum_{i \in I} p_i \leqslant 1$. (For ease of presentation, we do not distinguish notationally between sets and multisets.) Multidistributions over objects A are denoted by $\mathsf{M}(A)$. For a multidistribution $\mu \in \mathsf{M}(A)$ the induced distribution $\overline{\mu} \in \mathsf{D}(A)$ is defined in the obvious way by summing up the probabilities of equal objects.

Syntax. In Fig. 7, we detail the syntax of our core probabilistic (first-order) programming language. With the exception of ticks, expressions are given in `let`-normal form to simplify the presentation of the operational semantics and the typing rules. In order to ease the readability, we make use of mild syntactic sugaring in the presentation of actual code (as we already did above).

To make the presentation more succinct, we assume only the following types: a set of *base types* $\mathcal{B}$ such as Booleans $\mathsf{Bool} = \{\texttt{true}, \texttt{false}\}$, integers Int, or rationals Rat, product types, and binary trees T, whose internal nodes are labelled with elements $b : \mathsf{B}$, where B denotes an arbitrary base type. *Values* are either of base types, trees or pairs of values. We use lower-case Greek letters (from the beginning of the alphabet) for the denotation of types. Elements $t : \mathsf{T}$ are defined by the following grammar which fixes notation. $t ::= \texttt{leaf} \mid \texttt{node}\ t_1\ b\ t_2$. The size of a tree is the number of leaves: $|\texttt{leaf}| := 1$, $|\texttt{node}\ t\ a\ u| := |t| + |u|$.

We skip the standard definition of integer constants $n \in \mathbb{Z}$ as well as variable declarations, cf. [29]. Furthermore, we omit binary operators with the exception of essential comparisons. As mentioned, to represent sampling we make use of a dedicated `if`-`then`-`else` expression, whose guard evaluates to `true` depending on a coin toss with fixed probability. Further, non-deterministic choice is similarly rendered via an `if`-`then`-`else` expression. Moreover, we make use of *ticking*, denoted by an operator $\cdot^{\checkmark\, a/b}$ to annotate costs, where a, b are optional

$$f\ x_1\sigma\ \ldots\ x_k\sigma \mapsto e\sigma \qquad \texttt{if true then } e_1 \texttt{ else } e_2 \mapsto e_1$$

$$\texttt{let } x \texttt{ = } w \texttt{ in } e_2 \mapsto e_2[x \mapsto w] \qquad \texttt{if false then } e_1 \texttt{ else } e_2 \mapsto e_2$$

$$\texttt{match leaf with|leaf->}e_1\texttt{|node } x_0\ x_1\ x_2\texttt{->}e_2 \mapsto e_1$$

$$\texttt{match node } t\ a\ u \texttt{ with|leaf->}e_1\texttt{|node } x_0\ x_1\ x_2\texttt{->}e_2 \mapsto e_2$$

$$\texttt{match (}t\texttt{,}u\texttt{) with |(}t\texttt{,}u\texttt{)->}e \mapsto e$$

$$\texttt{if coin } a/b \texttt{ then } e_1 \texttt{ else } e_2 \mapsto \{e_1^{a/b}, e_2^{1-a/b}\} \qquad \texttt{if nondet then } e_1 \texttt{ else } e_2 \mapsto e_1$$

$$e^{\checkmark\, a/b} \overset{a/b}{\mapsto} e \qquad \texttt{if nondet then } e_1 \texttt{ else } e_2 \mapsto e_2$$

Assuming $f\ x_1\ \ldots\ x_k \texttt{ = } e \in \mathsf{P}$, σ respects the signature of f, and w is a value.

Fig. 8. One-Step Reduction Rules

and default to one. Following Avanzini et al. [2], we represent ticking $\cdot^{\checkmark}$ as an operation, rather than in `let`-normal form, as in Wang et al. [37]. (This allows us to suit a big-step semantics that only accumulates the cost of terminating expressions.) The set of all expressions is denoted $\mathcal{E}$.

A *typing context* is a mapping from variables $\mathcal{V}$ to types. Type contexts are denoted by upper-case Greek letters, and the empty context is denoted ε. A program P consists of a signature $\mathcal{F}$ together with a set of function definitions of the form $f\ x_1\ \ldots\ x_n = e_f$, where the x_i are variables and e_f an expression. When considering some expression e that includes function calls we will always assume that these function calls are defined by some program P. A *substitution* or (*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 write $\mathsf{dom}(\sigma)$ to denote the domain of σ.

4 Operational Semantics

Small-Step Semantics. The small-step semantics is formalised as a (weighted) non-deterministic, probabilistic abstract reduction system [4,9] over $\mathsf{M}(\mathcal{E})$. In this way (expected) cost, non-determinism and probabilistic sampling are taken care of. Informally, a probabilistic abstract reduction system is a transition systems where reducts are chosen from a probability distribution. A reduction wrt. such a system is then given by a stochastic process [9], or equivalently, as a reduction relation over *multidistributions* [4], which arise naturally in the context of non-determinism (we refer the reader to [4] for an example that illustrates the advantage of multidistributions in the presence of non-determinism).

Following [5], we equip transitions with (positive) weights, amounting to the cost of the transition. Formally, a *(weighted) Probabilistic Abstract Reduction System* (PARS) on a countable set A is a ternary relation $\cdot \overset{\cdot}{\mapsto} \cdot \subseteq A \times \mathbb{R}_0^+ \times \mathsf{D}(A)$. For $a \in A$, a rule $a \overset{c}{\mapsto} \{b^{\mu(b)}\}_{b \in A}$ indicates that a reduces to b with probability $\mu(b)$ and cost $c \in \mathbb{R}_0^+$. Note that any right-hand-side of a PARS is supposed to be a *full* distribution, ie. the probabilities in μ sum up to 1. Given two objects a and b, $a \overset{c}{\mapsto} \{b^1\}$ will be written $a \overset{c}{\mapsto} b$ for brevity. An object $a \in A$ is called *terminal* if there is no rule $a \overset{c}{\mapsto} \mu$, denoted $a \not\mapsto$. We suit the one-step reduction relation $\mapsto$ given in Fig. 8 as a (non-deterministic) PARS over multidistributions.

$$\frac{e \overset{c}{\mapsto} \{e_i^{p_i}\}_{i \in I}}{\{\mathbb{C}[e^1]\} \xrightarrow{c} \{\mathbb{C}[e_i]^{p_i}\}_{i \in I}}\ (\text{Step}) \qquad \frac{\mu_i \xrightarrow{c_i} \nu_i \quad \sum_i p_i \leqslant 1}{\biguplus_i p_i \cdot \mu_i \xrightarrow{\sum_i p_i c_i} \biguplus_i p_i \cdot \nu_i}\ (\text{Conv})$$

$$\frac{v \not\mapsto}{\{v^1\} \xrightarrow{0} \{v^1\}}\ (\text{NF})$$

Fig. 9. Probabilistic Reduction Rules of Distributions of Expressions

As above, we sometimes identify Dirac distributions $\{e^1\}$ with e. *Evaluation contexts* are formed by `let` expressions, as in the following grammar: $\mathbb{C}$ ::= $\Box$ | `let` x `=` $\mathbb{C}$ `in` e. We denote with $\mathbb{C}[e]$ the result of substitution the empty context $\Box$ with expression e. Contexts are exploited to lift the one-step reduction to a ternary weighted reduction relation $\overset{\cdot}{\longrightarrow} \subseteq \mathsf{M}(\mathcal{E}) \times \mathbb{R}_0^{+\infty} \times \mathsf{M}(\mathcal{E})$, cf. Fig. 9. (In (Conv), $\biguplus$ refers to the usual notion of multiset union.)

The relation $\overset{\cdot}{\longrightarrow}$ constitutes the operational (small-step) semantics of our simple probabilistic function language. Thus $\mu \xrightarrow{c} \nu$ states that the submultidistribution of objects μ evolves to a submultidistribution of reducts ν in one step, with an expected cost of c. Note that since $\overset{\cdot}{\mapsto}$ is non-deterministic, so is the reduction relation $\overset{\cdot}{\longrightarrow}$. We now define the evaluation of an expression $e \in \mathcal{E}$ wrt. to the small-step relation $\overset{\cdot}{\longrightarrow}$: We set $e \xrightarrow{c}_\infty \mu$, if there is a (possibly infinite) sequence $\{e^1\} \xrightarrow{c_1} \mu_1 \xrightarrow{c_2} \mu_2 \xrightarrow{c_3} \ldots$ with $c = \sum_{n \geqslant} c_n$ and $\mu = \lim_{n \to \infty} \overline{\mu_n}\restriction_V$, where $\overline{\mu_n}\restriction_V$ denotes the restriction of the distribution $\overline{\mu_n}$ (induced by the multidistribution μ_n) to a (sub-)distribution over values. Note that the $\overline{\mu_n}\restriction_V$ form a CPO wrt. the pointwise ordering, cf. [39]. Hence, the fixed point $\mu = \lim_{n \to \infty} \overline{\mu_n}\restriction_V$ exists. We also write $e \longrightarrow_\infty \mu$ in case the cost of the evaluation is not important.

(Positive) Almost Sure Termination. A program P is *almost surely terminating* (*AST*) if for any substitution σ, and any evaluation $e\sigma \longrightarrow_\infty \mu$, we have that μ forms a full distribution. For the definition of positive almost sure termination we assume that every statement of P is enclosed in an ticking operation with cost one; we note that such a cost models the length of the computation. We say P is *positively almost surely terminating* (*PAST*), if for any substitution σ, and any evaluation $e\sigma \xrightarrow{c}_\infty \mu$, we have $c < \infty$. It is well known that PAST implies AST, cf. [9].

Big-Step Semantics. We now define the aforementioned big-step semantics. We first define approximate judgments $\sigma \mid\!\frac{c}{n}\; e \Rightarrow \mu$, see Fig. 10, which say that in derivation trees with depth up to n the expression e evaluates to a subdistribution μ over values with cost c. We now consider the cost c_n and subdistribution μ_n in $\sigma \mid\!\frac{c_n}{n}\; e \Rightarrow \mu_n$ for $n \to \infty$. Note that the subdistributions μ_n in $\sigma \mid\!\frac{c_n}{n}\; e \Rightarrow \mu_n$ form a CPO wrt. the pointwise ordering, cf. [39]. Hence, there exists a fixed point $\mu = \lim_{n \to \infty} \mu_n$. Moreover, we set $c = \lim_{n \to \infty} c_n$ (note that either c_n converges to some real $c \in \mathbb{R}_0^{+\infty}$ or we have $c = \infty$). We now define the big-step judgments $\sigma \mid\!\frac{c}{}\; e \Rightarrow \mu$ by setting $\mu = \lim_{n \to \infty} \mu_n$ and $c = \lim_{n \to \infty} c_n$ for $\sigma \mid\!\frac{c_n}{n}\; e \Rightarrow \mu_n$.

$$\frac{e \text{ is not a value}}{\sigma \,\Big|\tfrac{0}{0}\, e \Rightarrow \{\}} \qquad \frac{}{\sigma \,\Big|\tfrac{0}{0}\, \texttt{leaf} \Rightarrow \{\texttt{leaf}^1\}} \qquad \frac{x_1\sigma = t \quad x_2\sigma = b \quad x_3\sigma = u}{\sigma \,\Big|\tfrac{0}{0}\, \texttt{node}\ x_1\ x_2\ x_3 \Rightarrow \{(\texttt{node}\ t\ b\ u)^1\}}$$

$$\frac{x\sigma = v}{\sigma \,\Big|\tfrac{0}{0}\, x \Rightarrow \{v^1\}} \qquad \frac{x_1\sigma = t \quad x_2\sigma = u}{\sigma \,\Big|\tfrac{0}{0}\, \texttt{(}x_1\texttt{,}x_2\texttt{)} \Rightarrow \{\texttt{(}t\texttt{,}u\texttt{)}^1\}} \qquad \frac{f\ y_1\ \ldots\ y_k\ \texttt{=}\ e \in \mathsf{P} \quad \sigma' \,\Big|\tfrac{c}{n}\, e \Rightarrow \mu}{\sigma \,\Big|\tfrac{c}{n+1}\, f\ x_1\ \ldots\ x_k \Rightarrow \mu}$$

$$\frac{\sigma \,\Big|\tfrac{c_1}{n}\, e_1 \Rightarrow \nu \quad \text{for all } w \in \mathsf{supp}(\nu)\text{: } \sigma[x \mapsto w] \,\Big|\tfrac{c_w}{n}\, e_2 \Rightarrow \mu_w}{\sigma \,\Big|\tfrac{c_1 + \sum_{w \in \mathsf{supp}(\nu)} \nu(w) \cdot c_w}{n+1}\, \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Rightarrow \sum_{w \in \mathsf{supp}(\nu)} \nu(w) \cdot \mu_w}$$

$$\frac{x\sigma = \texttt{leaf} \quad \sigma \,\Big|\tfrac{c}{n}\, e_1 \Rightarrow \mu}{\begin{array}{lll} \sigma \,\Big|\tfrac{c}{n+1}\, \texttt{match}\ x\ \texttt{with}\ \texttt{|}\ \texttt{leaf}\ \texttt{->}\ e_1 & & \Rightarrow \mu \\ \qquad\qquad\qquad\qquad\quad \texttt{|}\ \texttt{node}\ x_0\ x_1\ x_2\ \texttt{->}\ e_2 & & \end{array}}$$

$$\frac{x\sigma = \texttt{node}\ t\ a\ u \quad \sigma'' \,\Big|\tfrac{c}{n}\, e_2 \Rightarrow \mu}{\begin{array}{lll} \sigma \,\Big|\tfrac{c}{n+1}\, \texttt{match}\ x\ \texttt{with}\ \texttt{|}\ \texttt{leaf}\ \texttt{->}\ e_1 & & \Rightarrow \mu \\ \qquad\qquad\qquad\qquad\quad \texttt{|}\ \texttt{node}\ x_0\ x_1\ x_2\ \texttt{->}\ e_2 & & \end{array}}$$

$$\frac{x\sigma = \texttt{true} \quad \sigma \,\Big|\tfrac{c}{n}\, e_1 \Rightarrow \mu}{\sigma \,\Big|\tfrac{c}{n+1}\, \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Rightarrow \mu} \qquad \frac{\sigma \,\Big|\tfrac{c}{n}\, e_1 \Rightarrow \mu}{\sigma \,\Big|\tfrac{c}{n+1}\, \texttt{if}\ \texttt{nondet}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Rightarrow \mu}$$

$$\frac{x\sigma = \texttt{false} \quad \sigma \,\Big|\tfrac{c}{n}\, e_2 \Rightarrow \mu}{\sigma \,\Big|\tfrac{c}{n+1}\, \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Rightarrow \mu} \qquad \frac{\sigma \,\Big|\tfrac{c}{n}\, e_2 \Rightarrow \mu}{\sigma \,\Big|\tfrac{c}{n+1}\, \texttt{if}\ \texttt{nondet}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Rightarrow \mu}$$

$$\frac{x\sigma = \texttt{(}t\texttt{,}u\texttt{)} \quad \sigma''' \,\Big|\tfrac{c}{n}\, e \Rightarrow \mu}{\sigma \,\Big|\tfrac{c}{n+1}\, \texttt{match}\ x\ \texttt{with}\ \texttt{|}\ \texttt{(}x_1\texttt{,}x_2\texttt{)}\ \texttt{->}\ e \Rightarrow \mu} \qquad \frac{\sigma \,\Big|\tfrac{c}{n}\, e \Rightarrow \mu}{\sigma \,\Big|\tfrac{c + |\mu| \cdot {}^a/_b}{n+1}\, e^{\checkmark\, a/b} \Rightarrow \mu}$$

$$\frac{\sigma \,\Big|\tfrac{c_1}{n}\, e_1 \Rightarrow \mu_1 \quad \sigma \,\Big|\tfrac{c_2}{n}\, e_2 \Rightarrow \mu_2 \quad p = {}^a/_b}{\sigma \,\Big|\tfrac{pc_1 + (1-p)c_2}{n+1}\, \texttt{if}\ \texttt{coin}\ a/b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Rightarrow p\mu_1 + (1-p)\mu_2}$$

Here $\sigma[x \mapsto w]$ denotes the update of the environment σ such that $\sigma[x \mapsto w](x) = w$ and the value of all other variables remains unchanged. For function application we set $\sigma' := \{y_1 \mapsto x_1\sigma, \ldots, y_k \mapsto x_k\sigma\}$. In the rules covering `match` we set $\sigma'' := \sigma \uplus \{x_0 \mapsto t, x_1 \mapsto a, x_2 \mapsto u\}$ and $\sigma''' := \sigma \uplus \{x_0 \mapsto t, x_2 \mapsto u\}$ for trees and tuples respectively.

Fig. 10. Big-Step Semantics.

We want to emphasise that the cost c in $\sigma \,\Big|\!\tfrac{c}{}\, e \Rightarrow \mu$ only counts the ticks on terminating computations.

Theorem 1 (Equivalence). *Let* P *be a program and* σ *a substitution. Then, (i)* $\sigma \,\Big|\!\tfrac{c}{}\, e \Rightarrow \mu$ *implies that* $e\sigma \xrightarrow{c'}_\infty \mu$ *for some* $c' \geq c$*, and (ii)* $e\sigma \xrightarrow{c}_\infty \mu$ *implies that* $\sigma \,\Big|\!\tfrac{c'}{}\, e \Rightarrow \mu$ *for some* $c' \leqslant c$*. Moreover, if* $e\sigma$ *almost-surely terminates, we can choose* $c = c'$ *in both cases.*

The provided operational big-step semantics generalises the (big-step) semantics given in [18]. Further, while partly motivated by big-step semantics introduced in [37], our big-step semantics is technically incomparable—due to a different representation of ticking—while providing additional expressivity.

$$\frac{\Gamma|Q \vdash e\colon \alpha|Q'}{\Gamma|Q + {}^a\!/_b \vdash e^{\checkmark a/b}\colon \alpha|Q'} \quad (\mathsf{tick : now})$$

$$\frac{\Gamma|Q \vdash e\colon \alpha|Q'}{\Gamma|Q \vdash e^{\checkmark a/b}\colon \alpha|Q' - {}^a\!/_b} \quad (\mathsf{tick : defer})$$

Fig. 11. Ticking Operator. Note that a, b are not variables but literal numbers.

5 Type-and-Effect System for Expected Cost Analysis

5.1 Resource Functions

In Sect. 2, we introduced a variant of Schoenmakers' potential function, denoted as $\mathsf{rk}(t)$, and the additional potential functions $p_{(a_1,\ldots,a_n,b)}(t_1,\ldots,t_n) = \log_2(a_1 \cdot |t_1| + \cdots + a_n \cdot |t_n| + b)$, denoting the $\log_2$ of a linear combination of tree sizes. We demand $\sum_{i=1}^{n} a_i + b \geqslant 0$ ($a_i \in \mathbb{N}, b \in \mathbb{Z}$) for well-definedness of the latter; $\log_2$ 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 simply* annotation *of length m is a sequence $Q = [q_1, \ldots, q_m] \cup \left[\left(q_{(a_1,\ldots,a_m,b)}\right) a_i, b \in \mathbb{N}\right]$, vanishing almost everywhere. The length of Q is denoted $|Q|$. The empty annotation, that is, the annotation where all coefficients are set to zero, is denoted as $\varnothing$. Let $t_1, \ldots, t_m$ be a sequence of trees. Then, the potential of $t_m, \ldots, t_n$ wrt. Q is given by*

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

In case of an annotation of length 1, we sometimes write q_* instead of q_1. We may also write $\Phi(v\colon \alpha|Q)$ for the potential of a value of type α annotated with Q. Both notations were already used above. Note that only values of tree type are assigned a potential. 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.

Example 1. Let t be a tree. To model its potential as $\log_2(|t|)$ in according to Definition 1, we simply set $q_{(1,0)} := 1$ and thus obtain $\Phi(t|Q) = \log_2(|t|)$, which describes the potential associated to the input tree t of our leading example `descend` above. □

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

$$\frac{\Gamma|Q_1 \vdash e_1 : \alpha|Q' \quad \Gamma|Q_2 \vdash e_2 : \alpha|Q' \quad p = {}^{a}/_{b} \quad Q = p \cdot Q_1 + (1-p) \cdot Q_2}{\Gamma|Q \vdash \texttt{if coin } a/b \texttt{ then } e_1 \texttt{ else } e_2 : \alpha|Q'} \ (\mathsf{ite : coin})$$

Fig. 12. Conditional expression that models tossing a coin.

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

$$\mathcal{F}(f) := \{\alpha_1 \times \cdots \times \alpha_n \,|Q \to \beta_1 \times \cdots \times \beta_k|\, Q' | m = |Q|, 1 = |Q'\,|\} .$$

We suppose f *takes* n *arguments of which* m *are trees;* $m \leqslant n$ *by definition. Similarly, the return type may be the product* $\beta_1 \times \cdots \times \beta_i$*. In this case, we demand that at most one* β_i *is a tree type.*[3]

Instead of $\alpha_1 \times \cdots \times \alpha_n \,|Q \to \beta_1 \times \cdots \times \beta_k|\, Q' \in \mathcal{F}(f)$, we sometimes succinctly write $f : \alpha|Q \to \beta|Q'$ where α, β denote the product types $\alpha_1 \times \cdots \times \alpha_n$, $\beta_1 \times \cdots \times \beta_k$, respectively. It is tacitly understood that the above syntactic restrictions on the length of the annotations Q, Q' are fulfilled. For every function f, we also consider its *cost-free* variant from which all ticks have been removed. We collect the cost-free signatures of all functions in the set $\mathcal{F}^{\text{cf}}$.

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

Let $Q = [q_*] \cup [(q_{(a,b)})_{a,b \in \mathbb{N}}]$ be an annotation and let K be a rational such that $q_{(0,2)} + K \geqslant 0$. Then, $Q' := Q + K$ is defined as follows: $Q' = [q_*] \cup [(q'_{(a,b)})_{a,b \in \mathbb{N}}]$, where $q'_{(0,2)} := q_{(0,2)} + K$ and for all $(a,b) \neq (0,2)$ $q'_{(a,b)} := q_{(a,b)}$. Recall that $q_{(0,2)}$ is the coefficient of function $p_{(0,2)}(t) = \log_2(0|t|+2) = 1$, so the annotation $Q+K$ increments or decrements cost from the potential induced by Q by $|K|$, respectively. Further, we define the multiplication of an annotation Q by a constant K, denoted as $K \cdot Q$ pointwise. Moreover, let $P = [p_*] \cup [(p_{(a,b)})_{a,b \in \mathbb{N}}]$ be another annotation. Then the addition $P + Q$ of annotations P, Q is similarly defined pointwise.

5.2 Typing Rules

The non-probabilistic part of the type system is given in [19]. In contrast to the type system employed in [14,18], the cost model is not fixed but controlled by the ticking operator. Hence, the corresponding application rule (app) has been adapted. Costing of evaluation is now handled by a dedicated *ticking* operator, cf. Fig. 11. In Fig. 12, we give the rule $(\mathsf{ite : coin})$ responsible for typing probabilistic conditionals.

[3] The restriction to at most one tree type in the resulting type is non-essential and could be lifted. However, as our benchmark functions do not require this extension, we have elided it for ease of presentation.

```
foo t = match t with
  | leaf        → leaf
  | node l a r → let l' = (foo l)✓ in let r' = (foo r)✓ in
    if nondet then l' else r'
```

Fig. 13. Function `foo` illustrates the difference between (tick : now) and (tick : defer).

We remark that the core type system, that is, the type system given by Fig. 12 together with the remaining rules [19], ignoring annotations, enjoys subject reduction and progress in the following sense, which is straightforward to verify.

Lemma 1. *Let e be such that $e\colon\alpha$ holds. Then: (i) If $e \overset{c}{\mapsto} \{e_i^{p_i}\}_{i\in I}$, then $e_i\colon\alpha$ holds for all $i \in I$. (ii) The expression e is in normal form wrt. $\overset{c}{\mapsto}$ iff e is a value.*

5.3 Soundness Theorems

A program P is called *well-typed* if for any definition $f\,(x_1,\ldots,x_n) = e \in \mathsf{P}$ and any annotated signature $f : \alpha_1 \times \cdots \times \alpha_n|Q \to \beta|Q'$, we have a corresponding typing $x_1\colon\alpha_1,\ldots,x_k\colon\alpha_k|Q \vdash e\colon\beta|Q'$. A program P is called *cost-free* well-typed, if the cost-free typing relation is used (which employs the cost-free signatures of all functions).

Theorem 2 (Soundness Theorem for (tick : now)). *Let* P *be well-typed. Suppose $\Gamma|Q \vdash e\colon\alpha|Q'$ and $e\sigma \xrightarrow{c}_{\infty} \mu$. Then $\Phi(\sigma;\Gamma|Q) \geqslant c + \mathbb{E}_\mu(\lambda v.\Phi(v|Q'))$. Further, if $\Gamma|Q \vdash^{cf} e\colon\alpha|Q'$, then $\Phi(\sigma;\Gamma|Q) \geqslant \mathbb{E}_\mu(\lambda v.\Phi(v|Q'))$.*

Corollary 1. *Let* P *be a well-typed program such that ticking accounts for all evaluation steps. Suppose $\Gamma|Q \vdash e\colon\alpha|Q'$. Then e is positive almost surely terminating (and thus in particular almost surely terminating).*

Theorem 3 (Soundness Theorem for (tick : defer)). *Let* P *be well-typed. Suppose $\Gamma|Q \vdash e\colon\alpha|Q'$ and $\sigma \overset{c}{\models} e \Rightarrow \mu$. Then, we have $\Phi(\sigma;\Gamma|Q) \geqslant c + \mathbb{E}_\mu(\lambda v.\Phi(v|Q'))$. Further, if $\Gamma|Q \vdash^{cf} e\colon\alpha|Q'$, then $\Phi(\sigma;\Gamma|Q) \geqslant \mathbb{E}_\mu(\lambda v.\Phi(v|Q'))$.*

We comment on the trade-offs between Theorems 2 and 3. As stated in Corollary 1 the benefit of Theorem 2 is that when every recursive call is accounted for by a tick, then a type derivation implies the termination of the program under analysis. The same does not hold for Theorem 3. However, Theorem 3 allows to type more programs than Theorem 2, which is due to the fact that (tick : defer) rule is more permissive than (tick : now). This proves very useful, in case termination is not required (or can be established by other means).

We exemplify this difference on the `foo` function, see Fig. 13. Theorem 3 supports the derivation of the type $\mathsf{rk}(t) + \log_2(|t|) + 1 \geqslant \mathsf{rk}(\texttt{foo}\ t) + 1$, while Theorem 2 does not. This is due to the fact that potential can be "borrowed" with Theorem 3. To wit, from the potential $\mathsf{rk}(t) + \log_2(|t|) + 1$ for `foo` one can derive the potential $\mathsf{rk}(l') + \mathsf{rk}(r')$ for the intermediate context after both let-expression (note there is no +1 in this context, because the +1 has been used to

Table 2. Coefficients q such $q \cdot \log_2(|t|)$ is a bound on the expected amortized complexity of `splay` depending on the probability p of a rotation and the cost c of a recursive call, where the cost of a rotation is $1 - c$. Coefficients are additionally presented in decimal representation to ease comparison.

p \ c	$1/2$		$1/3$		$2/3$	
$1/2$	$9/8$	1.125	1	1	$5/4$	1.25
$1/3$	1	1	$5/6$	$0.8\dot{3}$	$7/6$	$1.\dot{6}$
$2/3$	$55/36$	$1.52\dot{7}$	$77/54$	$1.4\overline{259}$	$44/27$	$1.\overline{629}$

pay for the ticks around the recursive calls). Afterwards one can restore the $+1$ by weakening $\mathsf{rk}(l') + \mathsf{rk}(r')$ to $\mathsf{rk}(\texttt{foo}\ t) + 1$ (using in addition that $\mathsf{rk}(t) \geqslant 1$ for all trees t). On the other hand, we cannot "borrow" with Theorem 2 because the rule ($\mathsf{tick} : \mathsf{now}$) forces to pay the $+1$ for the recursive call immediately (but there is not enough potential to pay for this). In the same way, the application of rule ($\mathsf{tick} : \mathsf{defer}$) and Theorem 3 is essential to establish the logarithmic amortised costs of randomised splay trees. (We note that the termination of `foo` as well as of `splay` is easy to establish by other means: it suffices to observe that recursive calls are on sub-trees of the input tree).

6 Implementation and Evaluation

Implementation. Our prototype ATLAS is an extension of the tool described in [18]. In particular, we rely on the preprocessing steps and the implementation of the weakening rule as reported in [18] (which makes use of Farkas' Lemma in conjunction with selected mathematical facts about the logarithm as mentioned above). We only use the fully-automated mode reported in [18]. We have adapted the generation of the constraint system to the rules presented in this paper. We rely on Z3 [26] for solving the generated constraints. We use the optimisation heuristics of [18] for steering the solver towards solutions that minimize the resulting expected amortised complexity of the function under analysis.

Evaluation. We present results for the benchmarks described in Sect. 2 (plus a randomised version of splay heaps, the source code can be found in [19]) in Table 1. Table 3 details the computation time for type checking our results. Note that type inference takes considerably longer (tens of hours). To the best of our knowledge this is the first time that an expected amortised cost could be inferred for these data structures.

By comparing the costs of the operations of randomised splay trees and heaps to the costs of their deterministic versions (see Table 1), one can see the randomised variants have equal or lower complexity in all cases (as noted in Table 2 we have set the costs of the recursive call and the rotation to $1/2$, such that in the deterministic case, which corresponds to a coin toss with $p = 1$, these

Table 3. Number of assertions, solving time for type checking, and maximum memory usage (in mebibytes) for the combined analysis of functions per-module. The number of functions and lines of code is given for comparison.

Module	Functions	Lines	Assertions	Time	Memory
`RandSplayTree`	4	129	195 339	33M27S	19424.44
`RandSplayHeap`	2	34	77 680	6M15S	14914.51
`RandMeldableHeap`	3	15	25 526	20S	4290.67
`CoinSearchTree`	3	24	14 045	4S	1798.59
`Tree`	1	5	151	<1S	45.23

costs will always add up to one). Clearly, setting the costs of the recursion to the same value as the cost of the rotation does not need to reflect the relation of the actual costs. A more accurate estimation of the relation of these two costs will likely require careful experimentation with data structure implementations, which we consider orthogonal to our work. Instead, we report that our analysis is readily adapted to different costs and different coin toss probabilities. We present an evaluation for different values of p, recursion cost c and rotation cost $1 - c$ in Table 2. In preparing Table 2 the template $q^* \cdot \mathsf{rk}(t) + q_{(1,0)} \cdot \log_2(|t|) + q_{(0,2)}$ was used for performance reasons. The memory usage according to Z3's "max memory" statistic was 7129MiB per instance. The total runtime was 1H45M, with an average of 11M39S and a median of 2M33S. Two instances took longer time (36M and 49M).

Deterministic Benchmarks. For comparison we have also evaluated our tool ATLAS on the benchmarks of [18]. All results could be reproduced by our implementation. In fact, for the function `SplayHeap.insert` it yields an improvement of $^1/_4 \log_2(|h|)$, ie. $^1/_2 \log_2(|h|) + \log_2(|h| + 1) + {^3/_2}$ compared to $^3/_4 \log_2(|h|) + \log_2(|h| + 1) + {^3/_2}$. We note that we are able to report better results because we have generalised the resource functions $p_{(a_1,\ldots,a_m,b)}(t_1, \ldots, t_m) := \log_2 (a_1 \cdot |t_1| + \cdots + a_m \cdot |t_m| + b)$ to also allow negative values for b (under the condition that $\sum_i a_i + b \geq 1$) and our generalised ($\mathsf{let} : \mathsf{tree}$) rule can take advantage of these generalized resource functions (see [19] for a statement of the rule and the proof of its soundness as part of the proof of Theorem 3).

7 Conclusion

In this paper, we present the first fully-automated *expected amortised cost analysis* of self-adjusting data structures, that is, of *randomised splay trees*, *randomised splay heaps* and *randomised meldable heaps*, which so far have only (semi-)manually been analysed in the literature.

In future work, we envision to extend our analysis to related probabilistic settings such as skip lists [30], randomised binary search trees [20] and *randomised treaps* [8]. We note that adaptation of the framework developed in this paper to new benchmarks will likely require to identify new potential functions and the extension of the type-effect-system with typing rules for these potential functions. Further, on more theoretical grounds we want to clarify the connection of the here proposed expected amortised cost analysis with Kaminski's ert-calculus, cf. [15], and study whether the expected cost transformer is conceivable as a potential function.

References

1. Albers, S., Karpinski, M.: Randomized splay trees: theoretical and experimental results. IPL **81**(4), 213–221 (2002). https://doi.org/10.1016/S0020-0190(01)00230-7
2. Avanzini, M., Barthe, G., Lago, U.D.: On continuation-passing transformations and expected cost analysis. PACMPL **5**(ICFP), 1–30 (2021). https://doi.org/10.1145/3473592
3. Avanzini, M., Lago, U.D., Ghyselen, A.: Type-based complexity analysis of probabilistic functional programs. In: Proceedings of 34th LICS, pp. 1–13. IEEE (2019). https://doi.org/10.1109/LICS.2019.8785725
4. Avanzini, M., Lago, U.D., Yamada, A.: On probabilistic term rewriting. Sci. Comput. Program. **185**, 102338 (2020). https://doi.org/10.1016/j.scico.2019.102338
5. Avanzini, M., Moser, G., Schaper, M.: A modular cost analysis for probabilistic programs. PACMPL **4**(OOPSLA), 172:1–172:30 (2020). https://doi.org/10.1145/3428240
6. Barthe, G., Katoen, J.P., Silva, A. (eds.): Foundations of Probabilistic Programming. Cambridge University Press, Cambridge (2020). https://doi.org/10.1017/9781108770750
7. Batz, K., Kaminski, B.L., Katoen, J., Matheja, C., Noll, T.: Quantitative separation logic: a logic for reasoning about probabilistic pointer programs. PACMPL **3**(POPL), 34:1–34:29 (2019). https://doi.org/10.1145/3290347
8. Blelloch, G.E., Reid-Miller, M.: Fast set operations using treaps. In: Proceedings of 10th SPAA, pp. 16–26 (1998). https://doi.org/10.1145/277651.277660
9. Bournez, O., Garnier, F.: Proving positive almost-sure termination. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 323–337. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32033-3_24
10. Chatterjee, K., Fu, H., Murhekar, A.: Automated recurrence analysis for almost-linear expected-runtime bounds. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 118–139. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_6
11. Eberl, M., Haslbeck, M.W., Nipkow, T.: Verified analysis of random binary tree structures. J. Autom. Reason. **64**(5), 879–910 (2020). https://doi.org/10.1007/s10817-020-09545-0
12. Fürer, M.: Randomized splay trees. In: Proceedings of 10th SODA, pp. 903–904 (1999). http://dl.acm.org/citation.cfm?id=314500.315079
13. Gambin, A., Malinowski, A.: Randomized meldable priority queues. In: Rovan, B. (ed.) SOFSEM 1998. LNCS, vol. 1521, pp. 344–349. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49477-4_26

14. Hofmann, M., Leutgeb, L., Moser, G., Obwaller, D., Zuleger, F.: Type-based analysis of logarithmic amortised complexity. MSCS (2021). https://doi.org/10.1017/S0960129521000232
15. Kaminski, B.L., Katoen, J., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected runtimes of randomized algorithms. JACM **65**(5), 30:1–30:68 (2018). https://doi.org/10.1145/3208102
16. Kozen, D.: Semantics of probabilistic programs. J. Comput. Syst. Sci. **22**(3), 328–350 (1981)
17. Kozen, D.: A probabilistic PDL. JCSC **30**(2), 162–178 (1985). https://doi.org/10.1016/0022-0000(85)90012-1
18. Leutgeb, L., Moser, G., Zuleger, F.: ATLAS: automated amortised complexity analysis of self-adjusting data structures. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 99–122. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_5
19. Leutgeb, L., Moser, G., Zuleger, F.: Automated expected amortised cost analysis of probabilistic data structures. arXiv:2206.03537 (2022)
20. Martínez, C., Roura, S.: Randomized binary search trees. JACM **45**(2), 288–323 (1998). https://doi.org/10.1145/274787.274812
21. McIver, A., Morgan, C., Kaminski, B.L., Katoen, J.: A new proof rule for almost-sure termination. PACMPL **2**(POPL), 33:1–33:28 (2018). https://doi.org/10.1145/3158121
22. Meyer, F., Hark, M., Giesl, J.: Inferring expected runtimes of probabilistic integer programs using expected sizes. In: TACAS 2021. LNCS, vol. 12651, pp. 250–269. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_14
23. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, Cambridge (2005). https://doi.org/10.1017/CBO9780511813603
24. Moosbrugger, M., Bartocci, E., Katoen, J.-P., Kovács, L.: Automated termination analysis of polynomial probabilistic programs. In: ESOP 2021. LNCS, vol. 12648, pp. 491–518. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72019-3_18
25. Motwani, R., Raghavan, P.: Randomized algorithms. In: Algorithms and Theory of Computation Handbook. Cambridge University Press (1999). https://doi.org/10.1201/9781420049503-c16
26. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
27. Ngo, V.C., Carbonneaux, Q., Hoffmann, J.: Bounded expectations: resource analysis for probabilistic programs. In: Proceedings of 39th PLDI, pp. 496–512 (2018). https://doi.org/10.1145/3192366.3192394
28. Nipkow, T., Brinkop, H.: Amortized complexity verified. JAR **62**(3), 367–391 (2019)
29. Pierce, B.: Types and Programming Languages. MIT Press, Cambridge (2002)
30. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. CACM **33**(6), 668–676 (1990). https://doi.org/10.1145/78973.78977
31. Schoenmakers, B.: A systematic analysis of splaying. IPL **45**(1), 41–50 (1993)
32. Schoenmakers, B.: Data structures and amortized complexity in a functional setting. Ph.D. thesis, Eindhoven University of Technology (1992)
33. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, Hoboken (1999)
34. Sleator, D., Tarjan, R.: Self-adjusting binary search trees. JACM **32**(3), 652–686 (1985)

35. Takisaka, T., Oyabu, Y., Urabe, N., Hasuo, I.: Ranking and repulsing supermartingales for reachability in probabilistic programs. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 476–493. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_28
36. Tarjan, R.: Amortized computational complexity. SIAM J. Alg. Disc. Meth **6**(2), 306–318 (1985)
37. Wang, D., Kahn, D.M., Hoffmann, J.: Raising expectations: automating expected cost analysis with types. PACMPL **4**(ICFP), 110:1–110:31 (2020). https://doi.org/10.1145/3408992
38. Wang, P., Fu, H., Goharshady, A.K., Chatterjee, K., Qin, X., Shi, W.: Cost analysis of nondeterministic probabilistic programs. In: Proceedings of 40th PLDI, pp. 204–220. ACM (2019)
39. Winskel, G.: The Formal Semantics of Programming Languages. FCS, MIT Press (1993). https://doi.org/10.7551/mitpress/3054.003.0004