Amortized Analysis of Splay Trees via a Lax Homomorphism

LUKAS KEBULADZE*, Carnegie Mellon University, USA

1 INTRODUCTION

The cost of data structures is an important aspect of programs computer scientists seek to verify. Although worst-case cost analysis of a data structure operation yields an accurate upper bound on the cost of each operation, it is often more important to consider the total cost of a sequence of operations, which is achieved via the technique of *amortized analysis*. This work demonstrates a mechanization of the amortized analysis of splay trees via a *lax homomorphism* in the directed, effectful cost-aware logical framework (**decalf**) [5].

Amortized analysis. Traditionally, amortized analysis is accomplished via the potential method [20], which defines a real-valued potential function $\Phi : \mathcal{S} \to \mathbb{R}$ mapping each data structure state $S \in \mathcal{S}$ to its *potential*, representing the difference between the realistic and imagined costs. The amortized cost can be thought of as the running total for how much cost will be incurred over a sequence of operations. Moreover, the following equations relate the actual cost c, the amortized cost ac, and Φ over a single operation and a sequence of operations. In other words, total actual cost is equal to total amortized cost plus change in potential.

$$ac = c + \Phi(S') - \Phi(S) \qquad \sum_{i=1}^{m} c_i = \sum_{i=1}^{m} (ac_i + \Phi(S_{i-1}) - \Phi(S_i)) = \Phi(S_0) - \Phi(S_m) + \sum_{i=1}^{m} ac_i$$

Cost-aware logical framework. The directed effectful cost-aware logical framework (**decalf**) [5, 16] is a *directed* dependent call-by-push-value language [9] specialized for cost analysis. Call-by-push-value (cbpv) distinguishes pure values and effectful computations, and employs a pair of adjoint functors $\mathsf{F} \dashv \mathsf{U}$ to mediate between them. Of particular importance in cbpv are bind, which sequences computations of F types, and ret, which returns values as effectless computations, which together satisfy the standard β and η equations. Notationally we write $A \rightharpoonup B \coloneqq \mathsf{U}(A \to \mathsf{F}(B))$ for the (effectful) Kleisli functions. Cost is reified in decalf as a computation $\mathsf{step}^c(e)$ that records c units of cost (over a cost monoid $\mathbb{C}$) before running the computation e. Moreover, the cost effect satisfies equations that respect the monoid structure and commute with bind.

$$\textsc{F-Intro}\ \frac{a : A}{\mathsf{ret}(a) : \mathsf{F}(A)} \qquad \textsc{F-Elim}\ \frac{e : \mathsf{F}(A) \qquad f : A \to X}{\mathsf{bind}(e; f) : X}$$

$$\mathsf{bind}(\mathsf{step}^c(e); f) = \mathsf{step}^c(\mathsf{bind}(e; f))$$
$$c \leq_{\mathbb{C}} c' \to \mathsf{step}^c(e) \leq_X \mathsf{step}^{c'}(e)$$
$$e \leq_{\mathsf{F}(A)} e' \to ((x : A) \to f(x) \leq_X f'(x)) \to \mathsf{bind}(e; f) \leq_X \mathsf{bind}(e'; f')$$

An innovation of **decalf** is the notion of *program inequality* at each type, in the sense of directed type theory [11, 18]. Roughly speaking, $e \leq_X e'$ at computation type X indicates that the cost of computation e is bounded by that of e' and they return the same result. This provides an integrated way to perform simultaneous cost and correctness verification. Program inequality is reflexive and transitive, and monotone at bind and step as shown in the inequalities above.

decalf has been used to study the cost and behavior of various algorithms and data structures such as sorting algorithms [5, 16], amortized data structures such as batched queues and dynamic arrays [3], red-black trees [10], and higher-order effectful programs [5].

*Undergraduate, ACM Member ID: 0155216, Email: lkebulad@andrew.cmu.edu, Advisor: Robert Harper

Splay trees. A *splay tree* is a self-adjusting binary search tree (BST) developed by Sleator and Tarjan [19]. The key operation, splay : $\mathbb{N} \times \mathsf{tree} \rightharpoonup \mathsf{tree}$, finds key $k : \mathbb{N}$ in tree t : tree, and then rotates k to the root of t via a sequence of single or double rotations, called *splay-steps*, while preserving in-order traversal. The implementation of splay is instrumented with step annotations, with the cost model as the number of splay-steps. Sleator and Tarjan [19] showed that the amortized cost of a splay is $O(\log n)$ yielding an actual cost of $O(m \log n + n \log n)$ for m splay operations on an n-node tree. These two cost bounds demonstrate why splay trees are efficient, since they achieve logarithmic time access regardless of the internal tree structure while having fast access to recently accessed nodes. This work formalizes these aspects of splay trees via a notion of lax homomorphism.

2 AMORTIZED ANALYSIS OF SPLAY TREES VIA A LAX HOMOMORPHISM

The signature for a BST can be viewed as an algebraic signature with carrier T, representing the abstract implementation type, and operation find. Thus, implementations of a BST are structures that ascribe to the BST signature. In this case, the two implementations are SplayTree (abbreviated as **ST**) and ListTree (abbreviated as **LT**). As previously stated in Section 1, the correctness specification for **ST** is in terms of the preservation of the in-order traversal. To represent this in terms of lists, fictitious cost is attached to the find operation for **LT** to represent the amortized cost of splay trees which is $3\lfloor \log_2 n \rfloor + 1$[1] where n is the number of tree nodes and the return value is just the identity to represent in-order traversal preservation. Thus, **ST** is an amortizing implementation of the specification implementation **LT**. The definitions for the BST signature, **ST**, and **LT**, are shown below.[2]

record BST where	**ST** : **BST**	**LT** : **BST**
$\mathsf{T} : \mathsf{tp}^+$	$\mathbf{ST}.\mathsf{T} = \mathsf{tree}$	$\mathbf{LT}.\mathsf{T} = \mathsf{list}$
$\mathsf{find} : \mathbb{N} \times \mathsf{T} \rightharpoonup \mathsf{T}$	$\mathbf{ST}.\mathsf{find} = \mathsf{splay}$	$\mathbf{LT}.\mathsf{find}\ (k, l) = \mathsf{step}^{3\lfloor \log_2 \lvert l \rvert \rfloor + 1}(\mathsf{ret}(l))$

For demonstration purposes, this signature is slightly simplified. The complete type of find is $\mathbb{N} \times \mathsf{T} \rightharpoonup \mathsf{bool} \times \mathsf{T}$ where the boolean indicates whether the key k is found in tree t. This complete type rules out trivial implementations and allows for stronger proofs of correctness. Additionally, the signature includes a constructor method, fromList, making the signature an abstraction for static sets. This algebraic view of data structures gives rise to a notion of lax homomorphism between implementations. In the context of the BST signature, a lax homomorphism from **ST** to **LT** is an effectful map $\varphi : \mathbf{ST}.\mathsf{T} \rightharpoonup \mathbf{LT}.\mathsf{T}$ that preserves operations find_k in the following sense: $\mathsf{bind}(\mathbf{ST}.\mathsf{find}_k(t); \varphi) \leq \mathsf{bind}(\varphi(t); \mathbf{LT}.\mathsf{find}_k)$, where $\leq$ is in terms of program inequality in **decalf** as described in Section 1. As observed by Grodin and Harper [3], such a lax homomorphism φ may be seen as a generalized form of the potential function in amortized analysis that verifies not only amortized cost bound but also behavioral correspondence between implementations.

The potential function equation from Section 1, $ac = c + \Phi(S') - \Phi(S)$, can be relaxed to an inequality $ac \geq c + \Phi(S') - \Phi(S)$, as done in Sleator and Tarjan [19] and then rearranged to form inequality $c + \Phi(S') \leq \Phi(S) + ac$. The term $c + \Phi(S')$ corresponds to $\mathsf{bind}(\mathbf{ST}.\mathsf{find}_k(t); \varphi)$ where bind plays the role of addition, c is the cost of $\mathbf{ST}.\mathsf{find}_k(t)$, and φ charges $\Phi(S')$ amount of potential. Likewise, $\Phi(S) + ac$ corresponds to $\mathsf{bind}(\varphi(t); \mathbf{LT}.\mathsf{find}_k)$ where ac is the fictitious amortized cost on $\mathbf{LT}.\mathsf{find}_k$. To set up the lax homomorphism for splay trees, Φ and φ are defined as follows, where φ charges the potential Φ and returns the in-order traversal (inord) representation of the tree.

$$\Phi : \mathbf{ST}.\mathsf{T} \to \mathbb{N} \qquad \varphi : \mathbf{ST}.\mathsf{T} \rightharpoonup \mathbf{LT}.\mathsf{T}$$
$$\Phi(\mathsf{leaf}) = 0 \qquad \varphi(t) = \mathsf{step}^{\Phi(t)}(\mathsf{ret}(\mathsf{inord}(t)))$$
$$\Phi(\mathsf{node}\ l\ x\ r) = \Phi(l) + \lfloor \log_2 \lvert(\mathsf{node}\ l\ x\ r)\rvert \rfloor + \Phi(r)$$

Note that Φ is the potential function originally proposed by Sleator and Tarjan [19]. Thus, the

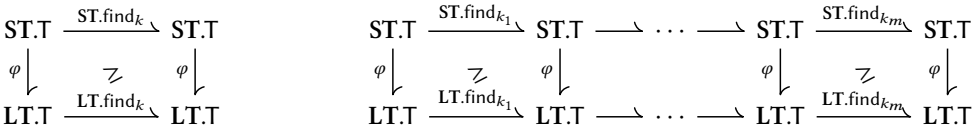
[1] This is a looser, asymptotically equivalent, amortized cost bound of Sleator and Tarjan [19, Lemma 1].
[2] For convenience, we write find_k for a family of operations indexed by key k.

following lemma describing the lax homomorphism in **decalf** verifies the behavior and the $O(\log n)$ amortized cost bound of the splay operation.

Lemma 2.1 (Generalized Access Lemma). *For all keys k and splay trees t it follows that*

$$\mathsf{bind}(\mathbf{ST}.\mathsf{find}_k(t); \varphi) \leq_{\mathsf{F}(\mathbf{LT}.\mathsf{T})} \mathsf{bind}(\varphi(t); \mathbf{LT}.\mathsf{find}_k).$$

Intuitively, this statement means that performing find on a splay tree costs less and behaves the same as performing find on a list, up to the potential function φ. Diagrammatically, Lemma 2.1 can be represented as the following inequality over the Kleisli category (left) and composed horizontally to represent a sequence of m find operations with arbitrary keys $k_1, k_2, ..., k_m$ (right).



When considering m applications of find_k, if only the cost is of interest, then a notion of cost bound, $\mathsf{IsBounded}(e, c) := e; \mathsf{ret}(\star) \leq_{\mathsf{F}(1)} \mathsf{step}^c(\mathsf{ret}(\star))$, can be used to represent the horizontally composed commutative diagram above as the following theorem:

Theorem 2.2 (Balance Theorem). *Let $k_1, k_2, ..., k_m$ be arbitrary keys where m is the number of* finds *performed. Let n be the number of nodes in the input splay tree t. Then,*

$$\mathsf{IsBounded}\ (\mathbf{ST}.\mathsf{find}^m_{k_i}(t))\ (m(3\lfloor \log_2 n \rfloor + 1) + n\lfloor \log_2 n \rfloor)$$

where $\mathbf{ST}.\mathsf{find}^m_{k_i}$ is the m-fold application of find *on keys $k_1, k_2, \cdots, k_m$.*

This theorem matches the balance theorem shown in Sleator and Tarjan [19, Theorem 1]. Lemma 2.1 and Theorem 2.2 are mechanized in **decalf** embedded in the Agda proof assistant [17] consisting of approximately 4000 lines of code (LOC) with Lemma 2.1 consisting of about 80% of the LOC. The remaining 20% goes towards Theorem 2.2, definitions, the implementation of splay trees (around 300 LOC), and other numeric reasoning lemmas.[3]

3 CONCLUSION AND RELATED WORK

This work contributes a fully mechanized amortized analysis of splay trees in **decalf** using the notion of a lax homomorphism. The result is a proof development that simultaneously verifies both functional behavior and amortized cost. Moreover, the lax homomorphism allows for modular verification, as downstream clients can analyze higher level data structures that are implemented via BSTs, such as ordered sets, by reasoning about lists in terms of behavior with the efficient cost bounds of splay trees.

Related Work. Another way to view the lax homomorphism φ is as a cost-aware abstraction function [6], a function that maps concrete implementations to abstract specifications, typical in verifying correctness of data structures [4, 13, 14]. Automated approaches to amortized analysis [7, 8, 21] typically cannot handle data structures as complex as splay trees, due to their limited ability to infer sophisticated potential functions. Cutler et al. [2] proposed a formal system for amortized analysis that can reason about splay trees, but their approach relies on on-paper proofs.

Other mechanizations of amortized data structures in proof assistants include verifying the union-find data structure via separation logic with time credits [1] and verifying splay trees in the Isabelle proof assistant [12, 15]. Compared with them, the lax homomorphism approach here is a unifying framework in dealing with simultaneous amortized cost and behavior verification.

[3]The proof development can be found at https://lk672.github.io/splay/Examples.Amortized.SplayTree.html.

ACKNOWLEDGMENTS

The author wishes to thank Harrison Grodin and Robert Harper for providing the opportunity for this work and especially Runming Li for providing continuous guidance and support throughout the process of producing this material. This material is based upon work supported by the United States Air Force Office of Scientific Research under grant number FA9550-21-0009 and FA9550-23-1-0434 (Tristan Nguyen, program manager). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR.

REFERENCES

[1] Arthur Charguéraud and François Pottier. 2019. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *Journal of Automated Reasoning* 62, 3 (2019), 331–365. doi:10.1007/s10817-017-9431-7

[2] Joseph W. Cutler, Daniel R. Licata, and Norman Danner. 2020. Denotational recurrence extraction for amortized analysis. *Proc. ACM Program. Lang.* 4, ICFP, Article 97 (Aug. 2020), 29 pages. doi:10.1145/3408979

[3] Harrison Grodin and Robert Harper. 2024. Amortized Analysis via Coalgebra. *Electronic Notes in Theoretical Informatics and Computer Science* Volume 4 - Proceedings of MFPS XL (Dec. 2024). doi:10.46298/entics.14797

[4] Harrison Grodin, Runming Li, and Robert Harper. 2026. Abstraction Functions as Types. *Proc. ACM Program. Lang.* 10, POPL, Article 31 (Jan. 2026), 28 pages. doi:10.1145/3776673

[5] Harrison Grodin, Yue Niu, Jonathan Sterling, and Robert Harper. 2024. Decalf: A Directed, Effectful Cost-Aware Logical Framework. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 10:273–10:301. doi:10.1145/3632852

[6] C. A. R. Hoare. 1972. Proof of Correctness of Data Representations. *Acta Informatica* 1, 4 (Dec. 1972), 271–281. doi:10.1007/BF00289507

[7] Jan Hoffmann and Martin Hofmann. 2010. Amortized Resource Analysis with Polynomial Potential. In *Programming Languages and Systems*, Andrew D. Gordon (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 287–306.

[8] Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. Association for Computing Machinery, New York, NY, USA, 185–197. doi:10.1145/604131.604148

[9] Paul Blain Levy. 2003. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Springer Netherlands, Dordrecht. doi:10.1007/978-94-007-0954-6

[10] Runming Li, Harrison Grodin, and Robert Harper. 2023. A Verified Cost Analysis of Joinable Red-Black Trees. arXiv:2309.11056 [cs] doi:10.48550/arXiv.2309.11056

[11] Daniel R. Licata and Robert Harper. 2011. 2-Dimensional Directed Type Theory. *Electronic Notes in Theoretical Computer Science* 276 (2011), 263–289. doi:10.1016/j.entcs.2011.09.026 Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII).

[12] Tobias Nipkow. 2015. Amortized Complexity Verified. In *Interactive Theorem Proving*, Christian Urban and Xingyuan Zhang (Eds.). Springer International Publishing, Cham, 310–324.

[13] Tobias Nipkow. 2016. Automatic Functional Correctness Proofs for Functional Search Trees. In *Interactive Theorem Proving*, Jasmin Christian Blanchette and Stephan Merz (Eds.). Springer International Publishing, Cham, 307–322. doi:10.1007/978-3-319-43144-4_19

[14] Tobias Nipkow (Ed.). 2025. *Functional Data Structures and Algorithms: A Proof Assistant Approach* (1 ed.). Vol. 67. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3731369

[15] Tobias Nipkow and Hauke Brinkop. 2019. Amortized Complexity Verified. *Journal of Automated Reasoning* 62, 3 (2019), 367–391. doi:10.1007/s10817-018-9459-3

[16] Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. 2022. A Cost-Aware Logical Framework. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 9:1–9:31. doi:10.1145/3498670

[17] Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI '09)*. Association for Computing Machinery, New York, NY, USA, 1–2. doi:10.1145/1481861.1481862

[18] Emily Riehl and Michael Shulman. 2023. A type theory for synthetic ∞-categories. arXiv:1705.07442 [math.CT] https://arxiv.org/abs/1705.07442

[19] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-adjusting binary search trees. *J. ACM* 32, 3 (July 1985), 652–686. doi:10.1145/3828.3835

[20] Robert Endre Tarjan. 1985. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods* 6, 2 (1985), 306–318. arXiv:https://doi.org/10.1137/0606031 doi:10.1137/0606031

[21] Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: a functional language for practical complexity analysis with invariants. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 79 (Oct. 2017), 26 pages. doi:10.1145/3133903