Amortized Analysis of Splay Trees via a Lax Homomorphism

Lukas Kebuladze
Carnegie Mellon University

Carnegie Mellon University

On-Paper Amortized Analysis (Potential Method)

Amortized analysis, instead of analyzing individual operations like in standard asymptotic analysis, considers the total cost of a sequence operations as follows.

- Define *potential function* $\Phi : D \to \mathbb{R}$ where D is the set of data structure states.
- $\forall d \in D, \Phi(d)$ represents the *potential* (difference between real and imagined cost).
- Given $\delta_\circ : D \to D$ that implements an operation, then for any arbitrary $d \in D$, the *amortized cost* of $\delta_\circ$ on state d is defined as

$$\sigma_\$ = \delta_\$(d) + \Phi(\delta_\circ(d)) - \Phi(d) \tag{1}$$

where $\delta_\$: D \to \mathbb{R}$ is the true cost of the operation on state d. Letting $d^{(i)} = \delta_\circ^{(i)}(d)$ and $\sigma_\$^{(i)} = \delta_\$(d^{(i)}) + \Phi(d^{(i+1)}) - \Phi(d^{(i)})$, the following holds by telescoping sums:

$$\sum_{i=0}^{m-1} \delta_\$(d^{(i)}) = \Phi(d^{(0)}) - \Phi(d^{(m)}) + \sum_{i=0}^{m-1} \sigma_\$^{(i)} \tag{2}$$

Key idea: If $\Phi(d^{(0)}) - \Phi(d^{(m)}) \leq 0$, then $\sum_{i=0}^{m-1} \delta_\$(d^{(i)}) \leq \sum_{i=0}^{m-1} \sigma_\$^{(i)}$ according to Equation 2 which bounds the true cost of a sequence of m operations in terms of amortized cost.

Cost-Aware Logical Framework

What it is: The directed effectful cost-aware logical framework (**decalf**) is a *directed* dependent call-by-push-value language specialized for cost analysis [3].

- 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.

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

Notationally we write $A \rightharpoonup B \coloneqq \mathsf{U}(A \to \mathsf{F}(B))$ for the (effectful) Kleisli functions.

Accounting of cost: 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.

Program inequality at each type: $e \leq_X e'$ 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.

$$\mathsf{step}^0(e) = e$$
$$\mathsf{step}^{c_1}(\mathsf{step}^{c_2}(e)) = \mathsf{step}^{c_1+c_2}(e)$$
$$\mathsf{bind}(\mathsf{step}^c(e); f) = \mathsf{step}^c(\mathsf{bind}(e; f))$$

Figure 1. Monoid structure of cost and bind commutativity with cost

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

Figure 2. Monotonicity at **step** and **bind**

Splay Trees

A *splay tree* is a self-adjusting binary search tree (BST) developed by Sleator and Tarjan [4] where **every time a node is accessed, it is moved to the root of the tree**.

It moves the node to the root via the operation $\mathsf{splay} : \mathbb{N} \times \mathsf{tree} \rightharpoonup \mathsf{tree}$, which finds key $k : \mathbb{N}$ in tree $t : \mathsf{tree}$, 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.

Goal: show that for an n-node tree, **splay** has $O(\log n)$ amortized cost and m **splay** operations on an n-node tree has actual cost $O(m \log n + n \log n)$.

Understanding the Splay Operation

The efficiency of **splay** comes from the *splay-steps*, which are divided into six cases, three of which are symmetric. Three cases are depicted below (left) with node x being splayed and an example of **splay** operating on a tree (right).

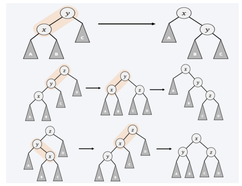

Figure 3. From top to bottom: zig, zig-zig, zig-zag

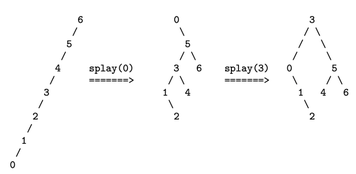

Figure 4. Tree becomes more balanced after splaying

How to Mechanize Amortized Analysis

Given some amortized data structure $\mathtt{D}$ with operation δ charging the real cost and specification data structure $\mathtt{S}$ with operation σ that charges the amortized cost of δ. In **decalf** embedded in Agda, do as follows.

1. Implement $(\mathtt{D}, \delta)$ and $(\mathtt{S}, \sigma)$ annotated with **step**, ascribing to some algebraic signature.
2. Define a lax homomorphism $\varphi : \mathtt{D} \rightharpoonup \mathtt{S}$ that charges potential function $\Phi : \mathtt{D} \to \mathbb{R}$.
3. Show that the single square commutes which proves the amortized cost bound of a single application of δ.
4. Show that the m-fold square commutes which proves the actual cost bound of m applications of δ.

Why Use a Lax Homomorphism?

This approach is prior work outlined in Grodin and Harper [1] and its application to this case is inspired by Grodin et al. [2]. The contribution here is applying this technique to splay trees.

Why Homomorphism? If we rearrange Equation 1 by moving $\Phi(d)$ to the other side, we get this:

$$\delta_\$(d) + \Phi(\delta_\circ(d)) = \Phi(d) + \sigma_\$$$

which is exactly the form of a homomorphism φ which charges cost Φ (i.e. a *single commuting square*). Since such a homomorphism also proves observational equivalence of states, *the homomorphism φ simultaneously defines the relation between the two states and the potential function.*

Once this is proved, *Equation 2 can be easily achieved by a m-fold composition of the single commuting square*

Why Lax? Complex amortized data structures, such as splay trees, have imprecise amortized analysis, since the amortized cost is an *upper bound* on the actual cost. In other words, Equation 1 needs to be turned into an inequality like so:

$$\delta_\$(d) + \Phi(\delta_\circ(d)) \leq \Phi(d) + \sigma_\$$$

which is achieved via a lax homomorphism.

Mechanization Definitions

First, the signature, amortizing implementation, and specification are defined.

$$\begin{aligned}&\textbf{record BST where}\\ &\quad \mathsf{T} : \mathsf{tp}^+\\ &\quad \mathsf{find} : \mathbb{N} \times \mathsf{T} \rightharpoonup \mathsf{T}\end{aligned}$$

Figure 5. Signature

$$\begin{aligned}&\mathbf{ST} \ : \ \mathbf{BST}\\ &\mathbf{ST}.\mathsf{T} = \mathsf{tree}\\ &\mathbf{ST}.\mathsf{find} = \mathsf{splay}\end{aligned}$$

Figure 6. Amortizing Impl

$$\begin{aligned}&\mathbf{LT} \ : \ \mathbf{BST}\\ &\mathbf{LT}.\mathsf{T} = \mathsf{list}\\ &\mathbf{LT}.\mathsf{find}\ (k, l) = \mathsf{step}^{3\lfloor \log_2 |l| \rfloor + 1}(\mathsf{ret}(l))\end{aligned}$$

Figure 7. Specification

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, making the signature and abstraction for static sets.

Mechanizing the Lax Homomorphism

Then, the lax homomorphism φ is defined, which charges the potential Φ and returns the in-order traversal (**inord**) representation of the tree.

$$\Phi : \mathbf{ST}.\mathsf{T} \to \mathbb{N}$$
$$\Phi(\mathsf{leaf}) = 0$$
$$\Phi(\mathsf{node}\ l\ x\ r) = \Phi(l) + \lfloor \log_2 |(\mathsf{node}\ l\ x\ r)| \rfloor + \Phi(r)$$

$$\varphi : \mathbf{ST}.\mathsf{T} \rightharpoonup \mathbf{LT}.\mathsf{T}$$
$$\varphi(t) = \mathsf{step}^{\Phi(t)}(\mathsf{ret}(\mathsf{inord}(t)))$$

Note that Φ is equivalent to the potential function used in the original analysis by Sleator and Tarjan [4].

Single Commuting Square

Through structural induction on the depth of the target key in the tree and reasoning about splay trees similar to the original amortized analysis by Sleator and Tarjan, the following is mechanized:

Generalized Access Lemma

For all keys k and splay trees t it follows that

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

This is equivalent to the following commutative diagram over the Kleisli category:

$$\begin{array}{ccc} \mathbf{ST}.\mathsf{T} & \xrightarrow{\mathbf{ST}.\mathsf{find}_k} & \mathbf{ST}.\mathsf{T} \\ {\scriptstyle\varphi}\downarrow & \swarrow & \downarrow{\scriptstyle\varphi} \\ \mathbf{LT}.\mathsf{T} & \xrightarrow{\mathbf{LT}.\mathsf{find}_k} & \mathbf{LT}.\mathsf{T} \end{array}$$

This lemma and corresponding diagram is the precise condition for φ to be a valid lax homomorphism.

Composed Commuting Square

To state the final theorem capturing the cost of a sequence of m **splay** operations, then $\mathsf{IsBounded}(e, c) \coloneqq e; \mathsf{ret}(\star) \leq_{\mathsf{F}(\mathbf{1})} \mathsf{step}^c(\mathsf{ret}(\star))$ is used to focus only on the cost to get the following which is proved in **decalf**.

Balance Theorem

Let $k_1, k_2, \ldots, k_m$ be arbitrary keys where m is the number of **find**s *performed. Let n be the number of nodes in the input splay tree t. Then,*

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

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

Similarly, this is equivalent to the following composed commutative diagram:

$$\begin{array}{ccccccccc} \mathbf{ST}.\mathsf{T} & \xrightarrow{\mathbf{ST}.\mathsf{find}_{k_1}} & \mathbf{ST}.\mathsf{T} & \longrightarrow \cdots \longrightarrow & \mathbf{ST}.\mathsf{T} & \xrightarrow{\mathbf{ST}.\mathsf{find}_{k_m}} & \mathbf{ST}.\mathsf{T} \\ {\scriptstyle\varphi}\downarrow & \swarrow & \downarrow{\scriptstyle\varphi} & & {\scriptstyle\varphi}\downarrow & \swarrow & \downarrow{\scriptstyle\varphi} \\ \mathbf{LT}.\mathsf{T} & \xrightarrow{\mathbf{LT}.\mathsf{find}_{k_1}} & \mathbf{LT}.\mathsf{T} & \longrightarrow \cdots \longrightarrow & \mathbf{LT}.\mathsf{T} & \xrightarrow{\mathbf{LT}.\mathsf{find}_{k_m}} & \mathbf{LT}.\mathsf{T} \end{array}$$

References

[1] Harrison Grodin and Robert Harper. Amortized Analysis via Coalgebra. *Electronic Notes in Theoretical Informatics and Computer Science*, Volume 4 - Proceedings of MFPS XL, December 2024.

[2] Harrison Grodin, Runming Li, and Robert Harper. Abstraction functions as types. *Proc. ACM Program. Lang.*, 10(POPL), January 2026.

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

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