Amortized Analysis via Coinduction

Harrison Grodin
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Robert Harper
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Amortized analysis is a program cost analysis technique for data structures in which the cost of operations is specified in aggregate, under the assumption of continued sequential use. Typically, amortized analyses are presented inductively, in terms of finite sequences of operations. We give an alternative coinductive formulation and prove that it is equivalent to the standard inductive definition. We describe a classic amortized data structure, the batched queue, and outline a coinductive proof of its amortized efficiency in **calf**, a dependent type theory for cost analysis.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Logic and verification; Software and its engineering → Functional languages; Theory of computation → Program reasoning; Theory of computation → Categorical semantics

Keywords and phrases amortized analysis, coinduction, data structure, mechanized proof

Digital Object Identifier 10.4230/LIPIcs.CALCO.2023.23

Category Early Ideas

Supplementary Material
Software (Source Code): `https://github.com/jonsterling/agda-calf` [18], archived at `swh:1:dir:7750187b111d75acca1980e9abffae2d63ffbe69`

Funding This material is based upon work supported by the United States Air Force Office of Scientific Research under grant number FA9550-21-0009 (Tristan Nguyen, program manager) and the National Science Foundation under grant number CCF-1901381. 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 or NSF.

Acknowledgements We are grateful to Yue Niu, Max New, and David Spivak for insightful discussions about this research.

1 Introduction

The **calf** framework is a dependent type theory that supports verification of both correctness conditions and cost bounds [19], based on call-by-push-value [17]. Amortized analysis is a cost analysis technique for data structures in which the operation costs are specified in aggregate, under the assumption of continued sequential use [23]. In this work, we demonstrate how amortized analysis can be understood as coalgebraic in **calf**.

In call-by-push-value, there are two sorts of types: value types A, B, C and computation types X, Y, Z. The type $\mathsf{F}A$ is a computation type classifying computations that result in a value of type A, and the type $\mathsf{U}X$ is a value type classifying suspended computations of type X. Computation types beyond $\mathsf{F}A$ will be essential for amortized analysis; in particular, we will make extensive use of products $X \times Y$, coproducts $\Sigma_{a:A} X(a)$, powers $A \to X$, and coinductive types $\nu X.\ Y(X)$ [2], all of which are computation types.

Semantically, we will interpret value types in **Set** and computation types in the category of $\mathbb{C}$-sets, where $\mathbb{C}$ is a monoid representing cost, as is standard for cost analysis of functional

programs [7, 8, 15, 6]. This is a simplification of **calf**, avoiding modalities. As in **calf**, we provide a primitive effect $\mathsf{step}^c(-)$ that incurs c units of abstract cost, interpreted using the $\mathbb{C}$-action. The $\mathbb{C}$-action associated to a computation type justifies equations describing how steps are incorporated into its elements:

$$\begin{aligned}
\mathsf{step}^c_{X\times Y}(\langle x, y\rangle) &= \langle \mathsf{step}^c_X(x), \mathsf{step}^c_Y(y)\rangle \\
\mathsf{step}^c_{\Sigma_{a:A} X(a)}(\langle a, x\rangle) &= \langle a, \mathsf{step}^c_X(x)\rangle \\
\mathsf{step}^c_{A\to X}(\lambda a.\ x) &= \lambda a.\ \mathsf{step}^c_X(x) \\
\mathsf{step}^c_{\nu X.\ Y(X)}(\mathsf{gen}(a.\ y;\ a_0)) &= \mathsf{gen}(a.\ \mathsf{step}^c_{Y(\nu X.\ Y(X))}(y);\ a_0)
\end{aligned}$$

In other words, cost at a product or power type is incurred pointwise, cost at a coproduct type is pushed into the given summand, and cost at a coinductive type is propagated forward. In this work, we will make use of the A-wide coproduct of a computation type X, also known as the *copower* of X by A [16, 9], which we write as $A \ltimes X \triangleq \Sigma_{-:A} X$. Note that $1 \ltimes X$ is isomorphic to X.

2 Cofree Comonads for Amortized Abstract Data Types

Throughout this paper, we will use queues as a running example of an abstract data type, although the development generalizes to other sequential-use abstract data types. Queues are an abstract type representing an ordered collection with a first-in-first-out data policy. Let value type E be the type of elements; the queue operations can be written as follows:

$$\begin{aligned}
\mathsf{enqueue}[e] &\sim 1 \\
\mathsf{dequeue} &\sim E + 1
\end{aligned}$$

This signature describes an operation $\mathsf{enqueue}[e]$ for each $e : E$ and an operation $\mathsf{dequeue}$.

In a type theory with one sort of type, a machine offering these operations is given via the following cofree comonad [12, 22, 21], interpreted in **Set**:

$$\mathsf{queue}(X) \triangleq \nu Q.\ (\mathsf{quit} : X) \times (\mathsf{enqueue} : E \to Q) \times (\mathsf{dequeue} : (E+1) \times Q)$$

Up to isomorphism, each operation corresponds to a product of its output type and Q, using a function for an E-wide product. In call-by-push-value, though, we must distinguish between a product of computation types and a copower of a value type and a computation type. Since the result type of an operation is a value type, such as $E + 1$ for the $\mathsf{dequeue}$ operation, we must use the latter. Thus, we may define the type of (amortized) queues as follows, interpreted in the category of $\mathbb{C}$-sets:

$$\mathsf{queue}(X) \triangleq \nu Q.\ (\mathsf{quit} : X) \times (\mathsf{enqueue} : E \to Q) \times (\mathsf{dequeue} : (E+1) \ltimes Q)$$

The type $\mathsf{queue}(X)$ can be understood as "object-oriented" [4, 13, 5], since the use of a queue involves a sequence of $\mathsf{enqueue}$ and $\mathsf{dequeue}$ projections terminated by a quit. Cost incurred at this type is propagated forward, accumulating at all future quit components (of type X) for end-of-use accounting.

3 Coinductive Amortized Analysis

Let $\mathbb{C} = (\mathbb{N}, +, 0)$. We define two queue implementations of type $\mathsf{queue}(X)$ and prove their amortized equivalence. Here, we let $X = \mathsf{F1}$, requiring that the queues terminate with an element of $\mathsf{F1}$ (i.e., simply a cost in $\mathbb{C}$).

Listing 1 Single-list specification implementation of a queue.

```
spec-queue : list E → queue (F unit)
quit    (spec-queue l) = ret triv
enqueue (spec-queue l) e = step 1 (spec-queue (l ++ [ e ]))
dequeue (spec-queue []) = ret (nothing , spec-queue [])
dequeue (spec-queue (e :: l)) = ret (just e , spec-queue l)
```

Listing 2 Amortized-efficient batched implementation of a queue.

```
batched-queue : list E → list E → queue (F unit)
quit    (batched-queue bl fl) = step (Φ (bl , fl)) (ret triv)
enqueue (batched-queue bl fl) e = batched-queue (e :: bl) fl
dequeue (batched-queue bl []) with reverse bl
... | [] = ret (nothing , batched-queue [] [])
... | e :: fl = step (length bl) (ret (just e , batched-queue [] fl))
dequeue (batched-queue bl (e :: fl)) =
  ret (just e , batched-queue bl fl)
```

▶ **Example 1** (Specification Queue)**.** One simple implementation of a queue, called `spec-queue`, is given in Listing 1 by coinduction using copattern matching [1], using a single list as the underlying representation type. The enqueue operation is annotated with one unit of cost; however, this is unrealistic, since a full traversal of the list is performed for each enqueue operation. We will treat this implementation as a client-facing specification, next defining a queue that actually implements this cost model. ⌟

▶ **Example 2** (Batched Queue)**.** Now, we define an amortized-efficient implementation which only incurs one large cost infrequently [10, 11, 3, 20]. This underlying representation type of the implementation is two lists: the "front list", fl, and the "back list", bl. Elements are enqueued to bl and dequeued from fl; if fl is empty when attempting to dequeue, the current bl is reversed and used in place of fl going forward. The **calf** implementation, called `batched-queue`, is shown in Listing 2. The quit case uses a *potential function* $\Phi(\mathsf{bl}, \mathsf{fl}) = \mathsf{length}(\mathsf{bl})$, as in the physicist's method of amortized analysis [23], accounting for elements enqueued on bl that were never moved to fl. ⌟

The amortized analysis is proved via a bisimulation; the theorem statement is analogous to the traditional amortized analysis, using the potential function to accumulate payment [23]. Every $\mathsf{enqueue}$ to `spec-queue` pushes one unit of cost forward, while `batched-queue` pushes $\mathsf{length}(\mathsf{bl})$ units of cost forward only on the occasional $\mathsf{dequeue}$, retroactively using its surplus potential from previous $\mathsf{enqueue}$ operations.

▶ **Theorem 3** (Amortized Analysis of Batched Queue)**.** *For all lists* bl *and* fl*,*

$$\texttt{batched-queue}\ \mathsf{bl}\ \mathsf{fl} = \mathsf{step}^{\Phi(\mathsf{bl},\mathsf{fl})}(\texttt{spec-queue}\ (\mathsf{fl} +\!\!+ \mathsf{reverse}\ \mathsf{bl})).$$

Proof. By routine coinduction, propagating cost forward over computation types. ◀

4 Relation to Inductive Amortized Analysis

Amortized analysis is typically framed algebraically, describing the cost incurred by a finite sequence of operations. In the preceding sections we observed that the analysis is naturally

Listing 3 Program evaluation at a queue.

```
eval : queue-program A → U (queue X) → A ⋉ X
eval (return a   ) q = a , Queue.quit q
eval (enqueue e p) q = eval p (Queue.enqueue q e)
eval (dequeue k  ) q =
  bind (k (proj₁ (Queue.dequeue q))) λ p →
  eval p (proj₂ (Queue.dequeue q))
```

viewed as *coalgebraic*. In fact these perspectives are equivalent. Define the free monad corresponding to the queue operation signature given in Section 2:

$$\mathsf{program}(A) \triangleq \mu P.\ (\mathsf{return} : A) + (\mathsf{enqueue} : E \times P) + (\mathsf{dequeue} : \mathsf{U}(E + 1 \to \mathsf{F}P))$$

An element of $\mathsf{program}(A)$ is a finite sequence of queue instructions terminated by returning a value of type A. We may evaluate a program on a queue, by induction on the program:

$$\mathsf{eval} : \mathsf{program}(A) \to \mathsf{U}(\mathsf{queue}(X)) \to A \ltimes X$$

This expresses the usual notion of running a sequence of operations on a data structure; the code is in Listing 3. Semantically, this definition corresponds to a morphism

$$\mathsf{program}(A) \ltimes \mathsf{queue}(X) \to A \ltimes X$$

resembling a monad-comonad interaction law [21, 14], here adjusted for call-by-push-value. Using eval, we may define an alternative notion of queue equivalence. Let $q_1, q_2 : \mathsf{queue}(X)$:

▶ **Definition 4** (Sequence-of-Operations Queue Equivalence). *Say $q_1 \approx q_2$ iff for all types A and programs $p : \mathsf{program}(A)$, it is the case that $\mathsf{eval}(p, q_1) = \mathsf{eval}(p, q_2)$.*

▶ **Theorem 5** (Amortizing Sequences of Operations). *It is the case that $q_1 = q_2$ iff $q_1 \approx q_2$.*

Proof. By routine ($\Rightarrow$) induction and ($\Leftarrow$) coinduction. ◀

Thus, coalgebraic amortized equivalence coincides with the traditional algebraic notion. Unsurprisingly, a proof that $q_1 \approx q_2$ shares the same core reasoning as a proof that $q_1 = q_2$; however, it requires the auxiliary definitions of $\mathsf{program}(A)$ and eval.

5 Conclusion

Here, we developed a computation type of amortized queues in **calf** as the cofree comonad of a functor based on the product, power, and copower computation type constructors, built to propagate cost forward for end-of-use accounting. We defined specification and amortized queue implementations and stated a theorem relating them via the physicist's method of amortized analysis. Finally, we observed that coinductive bisimulation coincides with traditional sequence-of-operations reasoning in amortized analysis. Our results for queues and two other simple amortized data structures are formalized in **calf**, which is embedded in Agda [18].

In future work, we hope to extend this approach to support abstract data types with binary and parallel operations, infinite sequences of operations, and situations in which an amortized implementation may be less costly than the specification. Additionally, we hope to better characterize the given constructions, accounting for the asymmetry present in call-by-push-value. As abstract data types are described via a comonad on the category of algebras for a monad, we also hope to connect to bialgebraic presentations of operational semantics [24].

References

1 Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. *ACM SIGPLAN Notices*, 48(1):27–38, January 2013. doi:10.1145/2480359.2429075.

2 Adriana Balan and Alexander Kurz. On Coalgebras over Algebras. *Electronic Notes in Theoretical Computer Science*, 264(2):47–62, August 2010. doi:10.1016/j.entcs.2010.07.013.

3 F. Warren Burton. An efficient functional implementation of FIFO queues. *Information Processing Letters*, 14(5):205–206, July 1982. doi:10.1016/0020-0190(82)90015-1.

4 William R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, Lecture Notes in Computer Science, pages 151–178, Berlin, Heidelberg, 1991. Springer. doi:10.1007/BFb0019443.

5 William R. Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 557–572, New York, NY, USA, October 2009. Association for Computing Machinery. doi:10.1145/1640089.1640133.

6 Joseph W. Cutler, Daniel R. Licata, and Norman Danner. Denotational recurrence extraction for amortized analysis. *Proceedings of the ACM on Programming Languages*, 4(ICFP):97:1–97:29, August 2020. doi:10.1145/3408979.

7 Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. *ACM SIGPLAN Notices*, 43(1):133–144, January 2008. doi:10.1145/1328897.1328457.

8 Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 140–151, New York, NY, USA, August 2015. Association for Computing Machinery. doi:10.1145/2784731.2784749.

9 Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. Enriching an Effect Calculus with Linear Types. In Erich Grädel and Reinhard Kahle, editors, *Computer Science Logic*, Lecture Notes in Computer Science, pages 240–254, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-642-04027-6_19.

10 David Gries. *The Science of Programming*. Springer New York, April 1989.

11 Robert Hood and Robert Melville. Real-time queue operations in pure LISP. *Information Processing Letters*, 13(2):50–54, November 1981. doi:10.1016/0020-0190(81)90030-2.

12 Bart Jacobs. Mongruences and cofree coalgebras. In V. S. Alagar and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science, pages 245–260, Berlin, Heidelberg, 1995. Springer. doi:10.1007/3-540-60043-4_57.

13 Bart Jacobs. Objects And Classes, Co-Algebraically. In Burkhard Freitag, Cliff B. Jones, Christian Lengauer, and Hans-Jörg Schek, editors, *Object Orientation with Parallelism and Persistence*, The Kluwer International Series in Engineering and Computer Science, pages 83–103. Springer US, Boston, MA, 1996. doi:10.1007/978-1-4613-1437-0_5.

14 Shin-ya Katsumata, Exequiel Rivas, and Tarmo Uustalu. Interaction Laws of Monads and Comonads. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20, pages 604–618, New York, NY, USA, July 2020. Association for Computing Machinery. doi:10.1145/3373718.3394808.

15 G. A. Kavvos, Edward Morehouse, Daniel R. Licata, and Norman Danner. Recurrence extraction for functional programs through call-by-push-value. *Proceedings of the ACM on Programming Languages*, 4(POPL):15:1–15:31, December 2019. doi:10.1145/3371083.

16 Gregory Maxwell Kelly. *Basic Concepts of Enriched Category Theory*. CUP Archive, February 1982.

17 Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Springer Netherlands, Dordrecht, 2003. URL: http://link.springer.com/10.1007/978-94-007-0954-6, doi:10.1007/978-94-007-0954-6.

18 Yue Niu, Jon Sterling, Harrison Grodin, and Robert Harper. **calf**: A Cost-Aware Logical Framework. URL: https://github.com/jonsterling/agda-calf.

19 Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. A cost-aware logical framework. *Proceedings of the ACM on Programming Languages*, 6(POPL):9:1–9:31, January 2022. doi:10.1145/3498670.

20 Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, Carnegie Mellon University, 1996. doi:10.1007/3-540-61628-4_5.

21 Gordon Plotkin and John Power. Tensors of Comodels and Models for Operational Semantics. *Electronic Notes in Theoretical Computer Science*, 218:295–311, October 2008. doi:10.1016/j.entcs.2008.10.018.

22 John Power and Olha Shkaravska. From Comodels to Coalgebras: State and Arrays. *Electronic Notes in Theoretical Computer Science*, 106:297–314, December 2004. doi:10.1016/j.entcs.2004.02.041.

23 Robert Endre Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, April 1985. doi:10.1137/0606031.

24 D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 280–291, June 1997. doi:10.1109/LICS.1997.614955.