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

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 $FA$ is a computation type classifying computations that result in a value of type $A$, and the type $UX$ is a value type classifying suspended computations of type $X$. Computation types beyond $FA$ 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 $A \rightarrow X$, 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 C-sets, where C is a monoid representing cost, as is standard for cost analysis of functional
programs \[7, 8, 15, 6\]. This is a simplification of \texttt{calf}, avoiding modalities. As in \texttt{calf}, we provide a primitive effect \texttt{step} \(c\) that incurs \(c\) units of abstract cost, interpreted using the \(C\)-action. The \(C\)-action associated to a computation type justifies equations describing how steps are incorporated into its elements:

\[
\text{step} \cdot X \times Y \langle x, y \rangle = \langle \text{step} \cdot X (x), \text{step} \cdot Y (y) \rangle
\]

\[
\text{step} \cdot \Sigma_{a : A} X (a) = \langle a, \text{step} \cdot X (x) \rangle
\]

\[
\text{step} \cdot A \rightarrow X (\lambda x. x) = \lambda x. \text{step} \cdot X (x)
\]

\[
\text{step} \cdot \nu X. Y (X) (\text{gen}(a \cdot y; a_0)) = \text{gen}(a \cdot \text{step} \cdot Y (X) (\nu X. Y (X)) (y); a_0)
\]

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 \(A \odot X\). Note that \(1 \odot X\) 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:

\[
\text{enqueue}[e] \sim 1
\]

\[
\text{dequeue} \sim E + 1
\]

This signature describes an operation \texttt{enqueue}[e] for each \(e : E\) and an operation \texttt{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 \texttt{Set}:

\[
\text{queue}(X) \triangleq \nu Q. (\text{quit} : X) \times (\text{enqueue} : E \rightarrow Q) \times (\text{dequeue} : (E + 1) \times Q)
\]

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 \texttt{dequeue} operation, we must use the latter. Thus, we may define the type of (amortized) queues as follows, interpreted in the category of \(C\)-sets:

\[
\text{queue}(X) \triangleq \nu Q. (\text{quit} : X) \times (\text{enqueue} : E \rightarrow Q) \times (\text{dequeue} : (E + 1) \times Q)
\]

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

## 3 Coinductive Amortized Analysis

Let \(C = (\mathbb{N}, +, 0)\). We define two queue implementations of type \texttt{queue}(\(X\)) and prove their amortized equivalence. Here, we let \(X = \texttt{F1}\), requiring that the queues terminate with an element of \texttt{F1} (i.e., simply a cost in \(C\)).
Example 1 (Specification Queue). One simple implementation of a queue, called \textit{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”, \textit{fl}, and the “back list”, \textit{bl}. Elements are enqueued to \textit{bl} and dequeued from \textit{fl}; if \textit{fl} is empty when attempting to dequeue, the current \textit{bl} is reversed and used in place of \textit{fl} going forward. The \textit{calf} implementation, called \textit{batched-queue}, is shown in Listing 2. The quit case uses a potential function \( \Phi(bl, fl) = \text{length}(bl) \), as in the physicist’s method of amortized analysis [23], accounting for elements enqueued on \textit{bl} that were never moved to \textit{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 \textit{enqueue} to \textit{spec-queue} pushes one unit of cost forward, while \textit{batched-queue} pushes \text{length}(bl) units of cost forward only on the occasional \textit{dequeue}, retroactively using its surplus potential from previous \textit{enqueue} operations.

Theorem 3 (Amortized Analysis of Batched Queue). For all lists \textit{bl} and \textit{fl},

\[
\text{batched-queue } bl \text{ fl} = \text{step}^{\Phi(bl, fl)}(\text{spec-queue } (fl + \text{reverse } bl)).
\]

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
23:4 Amortized Analysis via Coinduction

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:

$$\text{program}(A) ≜ \mu P. (\text{return} : A) + (\text{enqueue} : E × P) + (\text{dequeue} : U(E + 1 → FP))$$

An element of \(\text{program}(A)\) 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:

$$\text{eval} : \text{program}(A) → U (\text{queue}(X)) → A ⋉ X$$

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

$$\text{program}(A) ⋉ \text{queue}(X) → A ⋉ X$$

resembling a monad-comonad interaction law [21, 14], here adjusted for call-by-push-value. Using \(\text{eval}\), we may define an alternative notion of queue equivalence. Let \(q₁, q₂ : \text{queue}(X)\):

▶ Definition 4 (Sequence-of-Operations Queue Equivalence). Say \(q₁ ≈ q₂\) iff for all types \(A\) and programs \(p : \text{program}(A)\), it is the case that \(\text{eval}(p, q₁) = \text{eval}(p, q₂)\).

▶ Theorem 5 (Amortizing Sequences of Operations). It is the case that \(q₁ = q₂\) iff \(q₁ ≈ q₂\).

Proof. By routine (⇒) induction and (⇐) coinduction.

Thus, coalgebraic amortized equivalence coincides with the traditional algebraic notion. Unsurprisingly, a proof that \(q₁ ≈ q₂\) shares the same core reasoning as a proof that \(q₁ = q₂\); however, it requires the auxiliary definitions of \(\text{program}(A)\) and \(\text{eval}\).

5 Conclusion

Here, we developed a computation type of amortized queues in \(\text{calf}\) 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 \(\text{calf}\), 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


Amortized Analysis via Coinduction


