Electronic Notes in
Theoretical Informatics
And Computer Science

ENTICS
https://entics.episciences.org

Volume 4
Proceedings of
MFPS 2024

Amortized Analysis via Coalgebra

Harrison Grodin[1] Robert Harper[2]

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA

Abstract

Amortized analysis is a cost analysis technique for data structures in which cost is studied in aggregate: rather than considering the maximum cost of a single operation, one bounds the total cost encountered throughout a session. Traditionally, amortized analysis has been phrased inductively, quantifying over finite sequences of operations. Connecting to prior work on coalgebraic semantics for data structures, we develop the alternative perspective that amortized analysis is naturally viewed coalgebraically in a category of cost algebras, where a morphism of coalgebras serves as a first-class generalization of potential function suitable for integrating cost and behavior. Using this simple definition, we consider amortization of other sample effects, non-commutative printing and randomization. To support imprecise amortized upper bounds, we adapt our discussion to the bicategorical setting, where a potential function is a colax morphism of coalgebras. We support algebraic and coalgebraic operations simultaneously by using coalgebras for an endoprofunctor instead of an endofunctor, combining potential using a monoidal structure on the underlying category. Finally, we compose amortization arguments in the indexed category of coalgebras to implement one amortized data structure in terms of others.

Keywords: amortized analysis, cost analysis, call-by-push-value, data structures, abstract data types, writer monad, coalgebra, simulation, monoidal adjunctions, profunctors, indexed categories, bicategories, lax morphisms, colax morphisms

1 Introduction

In computer science, it is common to prove the cost of data structure operations, guaranteeing some exact or upper bound on the amount of abstract cost incurred. In simple cases, a tight bound can be proved about each operation in isolation. However, an upper bound can sometimes be too loose to be insightful, because any sequential use of the data structure can only reach the worst case infrequently. For example, an operation that uses \$8 of cost every eight invocations and no cost otherwise can be upper bounded by \$8, but this gives the grossly misleading perspective that a sequence of eight operations could cost up to \$64, even though the \$8 will only be charged once in the sequence. Instead, the cost of an operation should be considered in conjunction with the cost of the operations that came before or may come afterwards.

To address this problem, Tarjan [28] developed *amortized analysis*, a technique for bounding the total cost of a sequence of operations. Rather than claiming that the cost of the aforementioned operation is upper bounded by \$8, one can pretend that each invocation costs only \$1, averaging out the \$8 over the eight invocations. While this cost bound is not precisely true, it looks approximately true from the

[1] Email: hgrodin@cs.cmu.edu

[2] Email: rwh@cs.cmu.edu

Published December 15, 2024
10.46298/entics.14797

Proceedings Available Online at
https://doi.org/10.46298/entics.proceedings.mfps40

viewpoint of a client: a sequence of eight operations costs \$8, exactly as the \$1-per-operation abstraction suggests it should. We can visualize this reasoning using the following commutative diagram:

$$\begin{tikzcd}
\bullet \arrow[r, mapsto, "\$0"] \arrow[d, mapsto, "\$0"] & \bullet \arrow[r, mapsto, "\$0"] & \bullet \arrow[r, mapsto, "\$0"] & \bullet \arrow[r, mapsto, "\$0"] & \bullet \arrow[r, mapsto, "\$0"] & \bullet \arrow[r, mapsto, "\$0"] & \bullet \arrow[r, mapsto, "\$0"] & \bullet \arrow[r, mapsto, "\$8"] & \bullet \arrow[d, mapsto, "\$0"] \\
\bullet \arrow[r, mapsto, "\$1"] & \bullet \arrow[r, mapsto, "\$1"] & \bullet \arrow[r, mapsto, "\$1"] & \bullet \arrow[r, mapsto, "\$1"] & \bullet \arrow[r, mapsto, "\$1"] & \bullet \arrow[r, mapsto, "\$1"] & \bullet \arrow[r, mapsto, "\$1"] & \bullet \arrow[r, mapsto, "\$1"] & \bullet
\end{tikzcd}$$

Each mapping $\bullet \mapsto \bullet$ represents the cost of an operation. The arrows along the top represent the true cost, with the first seven operations taking no cost and the eighth operation taking \$8 of cost. The arrows along the bottom represent the imagined cost, with all operations taking \$1 cost. The lightened vertical arrows, annotated with \$0, connect the two perspectives, allowing us to state that both routes are equivalent. To compose this commutative cost diagram out of smaller squares for each individual operation, vertical arrows cannot always be annotated with \$0, as this would not satisfy the arithmetic constraints. Instead:

$$\begin{tikzcd}
\bullet \arrow[r, mapsto, "\$0"] \arrow[d, mapsto, "\$0"] & \bullet \arrow[r, mapsto, "\$0"] \arrow[d, mapsto, "\$1"] & \bullet \arrow[r, mapsto, "\$0"] \arrow[d, mapsto, "\$2"] & \bullet \arrow[r, mapsto, "\$0"] \arrow[d, mapsto, "\$3"] & \bullet \arrow[r, mapsto, "\$0"] \arrow[d, mapsto, "\$4"] & \bullet \arrow[r, mapsto, "\$0"] \arrow[d, mapsto, "\$5"] & \bullet \arrow[r, mapsto, "\$0"] \arrow[d, mapsto, "\$6"] & \bullet \arrow[r, mapsto, "\$8"] \arrow[d, mapsto, "\$7"] & \bullet \arrow[d, mapsto, "\$0"] \\
\bullet \arrow[r, mapsto, "\$1"] & \bullet \arrow[r, mapsto, "\$1"] & \bullet \arrow[r, mapsto, "\$1"] & \bullet \arrow[r, mapsto, "\$1"] & \bullet \arrow[r, mapsto, "\$1"] & \bullet \arrow[r, mapsto, "\$1"] & \bullet \arrow[r, mapsto, "\$1"] & \bullet \arrow[r, mapsto, "\$1"] & \bullet
\end{tikzcd}$$

Intuitively, the vertical arrows keep track of the difference between the realistic and imagined cost. At the beginning and end of this trace, both perspectives align and the difference is \$0, but at intermediate states, the imagined bottom perspective has incurred more cost than the realistic top perspective. This difference, called the *potential*, is essential in amortized analysis.

The technique of amortized analysis has been widely applied to data structures since, giving more practical bounds on ephemeral data structures. Since its inception, the amortized study of data structures has been phrased algebraically, reasoning about the total cost of a finite sequence of operations:

> In many uses of data structures, a *sequence of operations*, rather than just a single operation, is performed, and we are *interested in the total time of the sequence*, rather than in the times of the individual operations. [28]

This algebraic emphasis is in contrast to the common coalgebraic semantics taken when giving a semantics to sequential-use data structures, sometimes referred to as “objects” [15]. In this work, we take the perspective that amortized analysis is fundamentally coalgebraic, showing that the techniques used in amortized analysis are specialized instances of more general coalgebraic machinery, which elegantly connects to the theory of synthetic cost and behavior verification.

1.1 Amortized Analysis

In the original development of amortized analysis, Tarjan and Sleator [28, §2] describe the *physicist's method*, a general technique for tracking amortized cost in a sequence of operations. Let D be the set of states of a data structure. In this method, one defines a function $\Phi : D \to \mathbb{Z}$ that assigns to each state a *potential*, representing the difference (assumed to be nonnegative) between the realistic and imagined cost models. Then, if $\delta_\circ : D \to D$ implements an operation, the *amortized cost* of $\delta_\circ$ on a state d is defined as

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

where $\delta_\$(d)$ is the true cost of the operation on state d. It is common to iterate this amortization equation to reason about the total cost of a finite-length sequence of operations. Letting $d^{(i)} = \delta_\circ^{(i)}(d)$ and

$\sigma_\$^{(i)} = \delta_\$(d^{(i)}) + \Phi(d^{(i+1)}) - \Phi(d^{(i)})$, we have the following by telescoping sums:

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

Then, if $\Phi(d^{(0)}) \leq \Phi(d^{(n)})$, the true total cost $\sum_i \delta_\$(i)$ is bounded by the amortized cost $\sum_i \sigma_\$^{(i)}$. Given a suitable choice of Φ, each amortized cost $\sigma_\$^{(i)}$ can often be bounded by a simple term, such as a constant. For example, if $\sigma_\$^{(i)}$ is always a constant k, then the amortized cost of the sequence of operations is kn.

1.2 Coalgebraic Semantics of Data Structures

Coalgebras have been used to give a semantics for data structures implementing sequential-use abstract data types, in the style of object-oriented programming [15]. A *signature* (or interface) is represented by an endofunctor $\Sigma : \mathcal{C} \to \mathcal{C}$, where ΣA represents the operations provided by an implementation when A is the implementation type. For example, when $\mathcal{C} = \mathbf{Set}$, the signature

$$\Sigma A = (E \Rightarrow A) \times (1 + (E \times A))$$

describes data structures A that export two methods: one of type $E \Rightarrow A$ and one of type $1 + (E \times A)$. For example, this signature could be used to represent stacks or queues, where the methods are either push and pop (for stacks) or enqueue and dequeue (for queues). A Σ-*coalgebra* is a pair (D, δ) of a *carrier* object $D : \mathcal{C}$ and a *transition morphism* $\delta : D \to \Sigma D$. Such a coalgebra should be understood as an implementation of the signature Σ, consisting of a state type D and an implementation of the methods via δ. When Σ is a product, as above, δ can be specified via a collection of maps to each component, using the universal property of products. For example, here a coalgebra consists of two maps:

$$\delta_1 : D \to E \Rightarrow D \qquad\qquad \delta_2 : D \to 1 + (E \times D)$$

Given a state of type D, the maps offer each method for use. For example, interpreting the above signature for stacks, the methods will implement push and pop, respectively.

1.3 Abstract Cost Analysis via the Writer Monad

This work builds on the proposal of Grodin and Harper [9] to study amortized analysis coalgebraically in Calf, an effectful dependent type theory based on call-by-push-value [19] that supports the verification of both correctness conditions and cost bounds [21,10]. We recall the following syntax for programming with F-types, writing X, Y, Z for value types and A, B, C for computation types:

$$\frac{V : X}{\mathsf{ret}(V) : \mathsf{F}X} \qquad\qquad \frac{M : \mathsf{F}X \qquad x : X \vdash M' : A}{\mathsf{bind}\, x \leftarrow M \,\mathsf{in}\, M' : A}$$

We also use coproducts and monoidal products of computation types, whose syntaxes are presented in the Enriched Effect Calculus [5,6] and Linear/Non-Linear type theory [2], respectively. As in Calf, we include an effect primitive for instrumenting a computation M with c units of abstract cost, here notated $\mathsf{charge}\langle \$c \rangle(M)$, where $c : \mathbb{C}$ and $(\mathbb{C}, +, 0)$ is a monoid representing cost. For clarity, we notate that a value $c : \mathbb{C}$ is a cost as $\$c$. The cost effect increments an ongoing counter by c, respecting the monoid structure:

$$\mathsf{charge}\langle \$0 \rangle(M) = M \qquad\qquad \mathsf{charge}\langle \$c_1 \rangle(\mathsf{charge}\langle \$c_2 \rangle(M)) = \mathsf{charge}\langle \$(c_1 + c_2) \rangle(M)$$

Crucially for amortization, effects in call-by-push-value commute with other computations, which can be understood by thinking of computation types as “lazy” types of call-by-name:

$$\begin{aligned}
\mathsf{bind}\, x \leftarrow \mathsf{charge}\langle \$c\rangle(M)\ \mathsf{in}\ M' &= \mathsf{charge}\langle \$c\rangle(\mathsf{bind}\, x \leftarrow M\ \mathsf{in}\ M') && (M : \mathsf{F}X)\\
\mathsf{charge}\langle \$c\rangle(M)\ .\mathsf{proj}_i &= \mathsf{charge}\langle \$c\rangle(M\ .\mathsf{proj}_i) && (M : A \times B)\\
\mathsf{charge}\langle \$c\rangle(M)\ V &= \mathsf{charge}\langle \$c\rangle(M\ V) && (M : X \rightharpoonup A)\\
\mathsf{charge}\langle \$c\rangle(V, M) &= (V, \mathsf{charge}\langle \$c\rangle(M)) && ((V, M) : X \rtimes A)
\end{aligned}$$

One key observation of Calf is that cost analysis is inseparable from correctness verification: in general, the cost of a program may depend on its behavior. This attitude is further validated in the present work, since the amortized cost of a data structure operation may depend on aspects of its state.

Semantically, we will work in the Eilenberg–Moore category of a monad T on **Set**, written $\mathbf{Alg}(T)$. This category is complete and cocomplete, and it is powered and copowered over **Set**. We recall that for any monad $T : \mathbf{Set} \to \mathbf{Set}$, there is an adjunction $\mathsf{F} \dashv \mathsf{U} : \mathbf{Alg}(T) \to \mathbf{Set}$ whose induced monad is T. Often, we will let T be a writer monad $\mathbb{C} \times (-)$. An algebra for the writer monad is a set $\mathsf{U}A$ equipped with a coherent method for storing abstract cost within the set, $\mathsf{charge}_A : \mathbb{C} \times \mathsf{U}A \to \mathsf{U}A$. In the present work, this aspect of the category of algebras will be essential: since every object comes equipped with a method for absorbing cost, any cost incurred will by construction be amortized forward. Inspired by call-by-push-value [19], we will abbreviate morphisms $\mathsf{F}X \to A$ as simply $X \rightharpoonup A$, which we often implicitly understand as $X \to \mathsf{U}A$ using the adjunction. Such maps implicitly propagate cost from the input to the output, also essential for amortization since previously-incurred cost should never be lost. Also, when using the writer monad, we will notate the cost and behavior components of a map $\delta : X \rightharpoonup \mathsf{F}Y$ as $\delta_\$: X \to \mathbb{C}$ and $\delta_\circ : X \to Y$, respectively.

Synopsis

In Section 2, we introduce coalgebra morphisms and show that in the presence of cost, they generalize the potential functions of amortized analysis. In Section 3, we support inexact amortized upper bounds by generalizing from categories to bicategories and from coalgebra morphisms to colax coalgebra morphisms. In Section 4, we incorporate mixed-variance operations using endoprofunctors, exploiting the symmetric monoidal structure on the category of writer monad algebras present given a commutative cost model to sum the potential of all relevant instances. In Section 5, we observe that potential functions can be composed, and we view coalgebras as an indexed category in order to implement one amortized data structure in terms of another with a differing signature.

2 Coalgebra Morphisms as Generalized Potential Functions

For the purpose of cost analysis, we will typically consider data structures implemented as coalgebras over a signature endofunctor Σ on $\mathbf{Alg}(\mathbb{C} \times (-))$, the category of writer monad algebras for a monoid $(\mathbb{C}, +, 0)$. The carrier of the coalgebra in our examples will typically be of the form $\mathsf{F}\underline{D}$, the free cost algebra on a set $\underline{D}$. We start by giving two examples of data structure implementations, each with only a single method.

Example 2.1 For this example, we use $(\mathbb{C}, +, 0) \triangleq (\mathbb{Z}, +, 0)$, the additive monoid of integers. We use the signature functor $\Sigma = \mathrm{Id} : \mathbf{Alg}(\mathbb{C} \times (-)) \to \mathbf{Alg}(\mathbb{C} \times (-))$ to represent a single method: a transition morphism $\delta : D \to \mathrm{Id}(D)$ is the implementation of the method for chosen state type D. An Id-coalgebra consists of an object D of $\mathbf{Alg}(\mathbb{C} \times (-))$ and a transition morphism $\delta : D \to D$. Treating the method as allocation of one unit of space, we may define two Id-coalgebras: one simple, unrealistic model that allocates one cell per call, and one realistic model that allocates eight spaces every eight calls.

(i) The simple, unrealistic *specification* coalgebra has carrier $S = \mathsf{F}1$ and transition morphism as follows:

$$\begin{aligned}
&\sigma : 1 \rightharpoonup \mathsf{F}1\\
&\sigma\ * = \mathsf{charge}\langle \$1\rangle(\mathsf{ret}(*))
\end{aligned}$$

In this coalgebra, no state is maintained, and one allocation is performed per transition.

(ii) The realistic *data structure* coalgebra has carrier $D = \mathsf{F}(\mathsf{Fin}_8)$, tracking how many already-allocated cells are free. Its transition morphism is given by:

$$\begin{aligned}&\delta : \mathsf{Fin}_8 \rightharpoonup \mathsf{F}(\mathsf{Fin}_8)\\ &\delta\ \mathsf{zero} = \mathsf{charge}\langle \$8\rangle(\mathsf{ret}(7))\\ &\delta\ (\mathsf{suc}\ d) = \mathsf{ret}(d)\end{aligned}$$

If no space remains, \$8 of cost is charged, allocating eight cells; seven cells remain after the transition. If some space remains, the amount of space remaining is decreased without performing any allocations.

The coalgebra (D, δ) is an amortizing implementation of the specification coalgebra (S, σ). The coalgebras are not always exactly synchronized, as (starting from state $7 : \mathsf{Fin}_8$) the implementation incurs a large cost only after the specification "saves up" enough cost to match it. However, from the perspective of a client, the more complex implementation can be approximated via the simple specification. $\lrcorner$

To prove the relationship between the specification (S, σ) and the amortized implementation (D, δ), we write a morphism of coalgebras, generalizing the potential functions of amortized analysis.

2.1 Coalgebra Morphisms

To prove that (D, δ) is an amortizing implementation of (S, σ), one classically gives a potential function $\Phi : D \to \mathbb{C}$ satisfying the amortization condition discussed in Section 1. We now define what it means to be a morphism of coalgebras and show that the amortization condition falls out as a special case when working in the category of writer monad algebras.

Definition 2.2 Let (D, δ) and (S, σ) be Σ-coalgebras. A *morphism of* Σ*-coalgebras* from (D, δ) to (S, σ) consists of a morphism $\Phi : D \to S$ that preserves the Σ-coalgebra structure:

$$\begin{tikzcd} D \arrow[r, "\delta"] \arrow[d, "\Phi"] & \Sigma D \arrow[d, "\Sigma\Phi"] \\ S \arrow[r, "\sigma"] & \Sigma S \end{tikzcd}$$

In other words, Φ preserves observational equivalence: using only the Σ-coalgebra structure, one can make identical observations regardless of when Φ is used to transition from D to S. In this sense, (D, δ) is simulated by (S, σ), with the simulation mediated by Φ: up to the translation Φ, the coalgebra (S, σ) behaves just like (D, δ). We will refer to the commutativity of this diagram as the *generalized amortization condition* for reasons that will be developed shortly. $\lrcorner$

We write $\mathbf{Coalg}(\Sigma)$ for the category of Σ-coalgebras and coalgebra morphisms. Now, we show that the requirement that Φ preserve the coalgebra structure is exactly the required condition on a potential function in amortized analysis.

Example 2.3 Recall the Id-coalgebras from Example 2.1. To give a morphism from (D, δ) to (S, σ), we must provide a function $\Phi : \mathsf{F}(\mathsf{Fin}_8) \to \mathsf{F}1$, equivalently written $\Phi : \mathsf{Fin}_8 \rightharpoonup \mathsf{F}1$, that preserves the coalgebra structure, as specified above. Equationally, the structure preservation condition says that

$$\Phi \mathbin{;} \sigma = \delta \mathbin{;} \Sigma\Phi$$

as morphisms $\mathsf{Fin}_8 \rightharpoonup \mathsf{F}1$. Using the Eilenberg–Moore adjunction, a map $\mathsf{Fin}_8 \rightharpoonup \mathsf{F}1$ is equivalent to function $\mathsf{Fin}_8 \to \mathsf{U}(\mathsf{F}1)$. Since $\mathsf{U}(\mathsf{F}1) = \mathbb{C} \times 1 \cong \mathbb{C}$, it is equivalent to give a function $\Phi : \mathsf{Fin}_8 \to \mathbb{C}$ such that for all $d : \mathsf{Fin}_8$, it is the case that

$$\Phi(d) + \sigma_{\$}(*) = \delta_{\$}(d) + \Phi(\delta_{\circ}(d)).$$

Subtracting $\Phi(d)$ from both sides, using the commutativity of addition in $\mathbb{Z}$, and dropping the trivial argument $* : 1$, this condition is exactly the amortization condition

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

discussed in Section 1. In this case, such a function can be defined as $\Phi(d) = \$(7 - d)$. This morphism of coalgebras precisely witnesses the fact that (D, δ) is an amortizing implementation of (S, σ). ⌟

Here, (S, σ) serves as a client-facing amortized cost specification for (D, δ). While no meaningful information about the state of the computation is provided, the specification conveys an amortized cost that is accurate up to the potential function Φ. By including more information in the carrier of the specification coalgebra, we may support amortized costs that vary over time.

Example 2.4 Often, the cost of operations varies over time, whereas the previous example considers only constant specification costs. In simple traditional amortized analyses, one uses functions $\delta_\$: \mathbb{N} \to \mathbb{C}$ and $\sigma_\$: \mathbb{N} \to \mathbb{C}$ to assign distinct costs to each operation in a sequence, asking that

$$\sigma_\$(i) = \delta_\$(i) + \Phi(i + 1) - \Phi(i)$$

where $\Phi : \mathbb{N} \to \mathbb{C}$. Letting $S = D = \mathsf{F}\mathbb{N}$ and $\sigma_\circ(i) = \delta_\circ(i) = i + 1$, we recover the equivalent coalgebra morphism condition,

$$\Phi(i) + \sigma_\$(i) = \delta_\$(i) + \Phi(i + 1).$$

Here, both transition morphisms maintain the index of the current operation, making the assumption that some desired sequence of operations has been specified *a priori*. More generally, though, our coalgebraic perspective allows for arbitrary carriers, viewing the cost model of a data structure alongside its implementation rather than attempting to extract a cost-only function as a separable quantity. ⌟

Traditional accounts of amortized analysis make implicit use of commutativity and additive inverses in $\mathbb{C}$. However, the generalized amortization condition is sensible regardless of the cost model. Reasoning principles pertaining to amortization can be expressed at this new level of generality. For example, recall from Section 1.1 that a telescoping sum can be used to bound the cost of an n-length sequence of operations. Coalgebraically, this condition is simply the composition of n coalgebra morphism squares:

$$\begin{tikzcd}
D \arrow[r, "\delta"] \arrow[d, "\Phi"] & \Sigma D \arrow[r, "\Sigma\delta"] \arrow[d, "\Sigma\Phi"] & \Sigma^{(2)} D \arrow[r] \arrow[d, "\Sigma^{(2)}\Phi"] & \cdots \arrow[r, "\Sigma^{(n-1)}\delta"] & \Sigma^{(n)} D \arrow[d, "\Sigma^{(n)}\Phi"] \\
S \arrow[r, "\sigma"] & \Sigma S \arrow[r, "\Sigma\sigma"] & \Sigma^{(2)} S \arrow[r] & \cdots \arrow[r, "\Sigma^{(n-1)}\sigma"] & \Sigma^{(n)} S
\end{tikzcd}$$

Here, $\Sigma^{(n)}$ is the n-fold composition of Σ. Observe that when using the coalgebras of 2.1 and the potential function of 2.3, the commutative mappings presented in Section 1 are induced by this diagram. Henceforth, we break away from $\mathbb{Z}$ in favor of a cost model like $\mathbb{N}$ that does *not* admit additive inverses, making precise the assumption that potential must be nonnegative.

2.2 Classic Amortized Analyses

We will now describe more complex analyses using this framework. In general, for a data structure implementation (D, δ) and an amortized cost specification (S, σ), an amortized analysis will be a coalgebra morphism $\Phi : (D, \delta) \to (S, \sigma)$ serving as a behavior-relevant generalization of potential functions.

Example 2.5 Let $\Sigma A = E \rightharpoonup A$, the E-fold power, representing the signature with one method for reading one value of type E at a time. We can use this signature to implement a dynamically-resizing array, a classic first example of an amortized data structure, where the method represents pushing an element of type E to the end of the array. In this data structure, we represent a list of arbitrary length using an underlying array with a fixed length. When more data is added to the list than the current array can hold, a new array of twice the length is allocated, and the old data is copied over. In the cost model here, we charge one unit of cost for each write, array allocation, and array deallocation.

(i) The specification again uses carrier $S = \mathsf{F}1$. Its transition morphism $\sigma : 1 \rightharpoonup \mathsf{F}1$ is given by:

$$\begin{aligned}&\sigma : \mathsf{F}1\\&\sigma = \mathsf{charge}\langle \$3\rangle(\mathsf{ret}(*))\end{aligned}$$

In this coalgebra, no state is maintained, and we charge \$3 at each operation in order to save up for an eventual copy.

(ii) The data structure implementation uses carrier $D = \mathsf{F}\underline{D}$, where

$$\underline{D} = \sum_{n:\mathbb{N}} \mathsf{array}_E[2^n - 1, 2^{n+1} - 1)$$

stores a logarithm-size bound n and an array with length between $2^n - 1$ and $2^{n+1} - 1$. We give the transition morphism δ by cases:

$$\begin{aligned}&\delta : \underline{D} \rightharpoonup E \rightharpoonup \mathsf{F}\underline{D}\\&\delta\ (n, a)\ e = \begin{cases}\mathsf{charge}\langle \$(3 + \mathsf{length}(a))\rangle(\mathsf{ret}(\mathsf{suc}\ n, a +\!\!+ [e])) & \text{if } \mathsf{length}(a) + 1 = 2^{n+1} - 1\\ \mathsf{charge}\langle \$1\rangle(\mathsf{ret}(n, a +\!\!+ [e])) & \text{otherwise}\end{cases}\end{aligned}$$

We charge $\$(3 + \mathsf{length}(l))$ in the expensive case when we have to copy the data to a new array, accounting for the allocation, write, copy, and deallocation, and we charge \$1 for each write in the cheap case, since we have enough space and only need to perform a single write operation.

To give a morphism from (D, δ) to (S, σ), we must provide a function $\Phi : \underline{D} \to \mathbb{N}$ such that the necessary square commutes. We define $\Phi(n, a) = \$(2(\mathsf{length}(a) + 1) - 2^{n+1})$, which meets the necessary criterion, showing that (S, σ) is a reasonable specification for (D, δ). ⌟

Once again, the carrier of the specification coalgebra is trivial, since each push operation has a constant amortized cost. Sometimes, though, an operation can have a cost dependent on some aspects of the state, modeled by a nontrivial carrier of the specification coalgebra. Then, the generalized potential function reveals these aspects about the data structure state in addition to mediating the potential cost.

Example 2.6 Extending Example 2.5, we can add a method parameterized by an endofunction $E \Rightarrow E$ to update all values in an array. Let $\Sigma A = (E \Rightarrow A) \times ((E \Rightarrow E) \Rightarrow A)$. Then, for a carrier D, a transition morphism consists of a pair of maps $\delta_1 : D \to E \Rightarrow D$ and $\delta_2 : D \to ((E \Rightarrow E) \Rightarrow D)$. We let δ_1 be the same map as before, and we let $\delta_2(n, a)(f) = \mathsf{charge}\langle \$(\mathsf{length}(a))\rangle(\mathsf{ret}(n, \mathsf{map}\ f\ a))$. There is no way to amortize this large cost, since the update method may be called arbitrarily often. Thus, to match the cost of this update method in the specification coalgebra (S, σ), the carrier must reveal some data about the underlying array. Since only the length of the array matters here, we may choose $S = \mathsf{F}\mathbb{N}$, only tracking the length. Then, we define σ as follows, giving the projections separately via copattern matching [1]:

$$\begin{aligned}&\sigma : \mathbb{N} \rightharpoonup \Sigma(\mathsf{F}\mathbb{N})\\&\sigma\ .\mathsf{push}\ n\ e = \mathsf{charge}\langle \$3\rangle(\mathsf{ret}(\mathsf{suc}\ n))\\&\sigma\ .\mathsf{update}\ n\ f = \mathsf{charge}\langle \$n\rangle(\mathsf{ret}(n))\end{aligned}$$

The push method still costs \$3, but now it must remember the increase of the length of the array. The update method now can cost $\$n$ when the state is some n, preserving the length n. The coalgebra morphism $\Phi : \underline{D} \rightharpoonup \mathsf{F}\mathbb{N}$ is identical on cost, but the new nontrivial behavior component sends a bounded array (n, a) to its length, $\mathsf{length}(a)$, written $\Phi(n, a) = \mathsf{charge}\langle \$(2(\mathsf{length}(a) + 1) - 2^{n+1})\rangle(\mathsf{ret}(\mathsf{length}(a)))$. ⌟

In this example, the state-dependent cost only appears for the non-amortized operation. In general, though, this need not be the case; cost is allowed to depend on behavior. For example, the operations on a splay tree have an amortized cost logarithmic in the size of the tree [26]. We now consider queues, a classic example of an amortized data structures that amortize cost using two operations.

Example 2.7 A queue is an abstract data type in which elements are enqueued to one end of the queue and dequeued from the other end.[3] To express this method in the signature, we let

$$\Sigma A = (E \rightharpoonup A) \times (\mathsf{F}1 + E \rtimes A),$$

where $E \rtimes A$ is the E-fold copower. The first method, enqueue, accepts an element of type E to store and continues. The second method, dequeue, either terminates if the queue is empty or provides an element of type E before continuing. This signature is similar to that of Section 1.2, adapted to the category of cost algebras by replacing exponentials and products with powers and copowers where necessary.

(i) The specification uses carrier $S = \mathsf{F}(\mathsf{list}(E))$, storing a specification-level list of elements. Its transition morphism σ implements queues naively via a list of elements:

$$\begin{array}{l}
\sigma : \mathsf{list}(E) \rightharpoonup \Sigma(\mathsf{F}(\mathsf{list}(E))) \\
\sigma\ .\mathsf{enqueue}\ l\ e = \mathsf{charge}\langle \$1 \rangle(\mathsf{ret}(l \mathbin{+\!\!+} [e])) \\
\sigma\ .\mathsf{dequeue}\ [] = \mathsf{inj}_1(\mathsf{ret}(*)) \\
\sigma\ .\mathsf{dequeue}\ (e :: l) = \mathsf{inj}_2(e, \mathsf{ret}(l))
\end{array}$$

(ii) The data structure implementation uses carrier $D = \mathsf{F}(\mathsf{list}(E)^2)$, storing a pair of lists to form a "batched queue" [14,4,8,23]. The first list is treated as an "inbox", storing enqueued elements, and the second list is treated as an "outbox", producing elements to dequeue. Occasionally, when the outbox is empty, the inbox is reversed and placed in the outbox.

$$\begin{array}{l}
\delta : \mathsf{list}(E)^2 \rightharpoonup \Sigma(\mathsf{F}(\mathsf{list}(E)^2)) \\
\delta\ .\mathsf{enqueue}\ (l_i, l_o)\ e = \mathsf{ret}(e :: l_i, l_o) \\
\delta\ .\mathsf{dequeue}\ (l_i, []) = \begin{cases} \mathsf{inj}_1(\mathsf{ret}(*)) & \text{if } \mathsf{reverse}(l_i) = [] \\ \mathsf{charge}\langle \$(\mathsf{length}(l_i)) \rangle(\mathsf{inj}_2(e, \mathsf{ret}(l_o))) & \text{if } \mathsf{reverse}(l_i) = e :: l_o \end{cases} \\
\delta\ .\mathsf{dequeue}\ (l_i, e :: l_o) = \mathsf{inj}_2(e, \mathsf{ret}(l_i, l_o))
\end{array}$$

Here, our cost model charges $\$(\mathsf{length}(l))$ cost for a call to $\mathsf{reverse}(l)$.

The coalgebra morphism representing the amortized analysis is a first-class effectful program, integrating cost and behavior verification:

$$\Phi(l_i, l_o) = \mathsf{charge}\langle \$(\mathsf{length}(l_i)) \rangle(\mathsf{ret}(l_o \mathbin{+\!\!+} \mathsf{reverse}(l_i)))$$

In addition to the traditional cost-level potential function $\mathsf{length}(l_i)$, it includes a behavioral simulation $l_o \mathbin{+\!\!+} \mathsf{reverse}(l_i)$ converting the pair of lists to a single specification-level list. $\lrcorner$

2.3 Generalizations of Amortized Analysis

Viewing amortized analysis coalgebraically allows the underlying category to be swapped out, leading to compact and elegant presentations of novel variations of amortization.

Example 2.8 Traditionally, it is assumed that addition in the cost model is commutative and admits an inverse. In this form, though, no requirements are placed on the monoid whatsoever. For example, we may let our "costs" be strings, where the monoid operation is concatenation. Then, amortization represents buffering, a performance technique in which many strings are occasionally printed in aggregate to avoid repeating fixed costs associated with the writing of any data. Let $\Sigma A = \mathsf{String} \rightharpoonup A$, providing a single method for printing.

(i) The specification coalgebra uses state $S = \mathsf{F}1$. Its transition morphism $\sigma : 1 \rightharpoonup \mathsf{String} \rightharpoonup \mathsf{F}1$ simply prints the provided string:

$$\begin{array}{l}
\sigma : \mathsf{String} \rightharpoonup \mathsf{F}1 \\
\sigma\ s = \mathsf{charge}\langle \$s \rangle(\mathsf{ret}(*))
\end{array}$$

[3] This was the principal example of the predecessor to this work [9].

Here, "charging a string cost" should be understood as printing the string.

(ii) The implementation uses state $D = \mathsf{F}\underline{D}$, where

$$\underline{D} = \sum_{s:\mathsf{String}} \mathsf{length}(s) < n$$

for a fixed buffer size n. Its transition morphism prints strings in chunks of length n, saving any remaining characters in the state. Let chop_n split a string into a portion with length a multiple of n and a remainder with length less than n. Then, we define:

$\delta : \underline{D} \rightharpoonup \mathsf{String} \rightharpoonup \mathsf{F}\underline{D}$
$\delta\ s_0\ s = \mathsf{let}\ (s', s'_0) = \mathsf{chop}_n(s_0 +\!\!+\ s)\ \mathsf{in}\ \mathsf{charge}\langle \$s' \rangle(\mathsf{ret}(s'_0))$

The generalized potential function $\Phi : \underline{D} \to \mathsf{String}$ is simply the inclusion, "flushing" any data remaining in the buffer. For example, when $n = 8$ and printing "world" on a buffer containing "hello", the generalized amortization condition is the following:

$$\Phi(\text{``}\mathsf{hello}\text{''}) +\!\!+ \text{``}\mathsf{world}\text{''} = \text{``}\mathsf{hellowor}\text{''} +\!\!+ \Phi(\text{``}\mathsf{ld}\text{''})$$

In other words, we ask that flushing the original buffer with "hello" and then printing "world" is equivalent to printing "hellowor" via the buffered implementation and then flushing the remaining buffer "ld".

This shows that buffering can be understood as an amortized implementation of printing strings in real time. Using the state monad in place of the writer monad, this technique can be adapted to support buffering of arbitrary state. ⌟

From this perspective, we may move beyond the writer monad, amortizing other effects. For any strong monad T, it is also the case that $T(\mathbb{C} \times (-))$ forms a monad, where $\mathbb{C}$ is an arbitrary monoid for cost. We now consider working with coalgebras in the category of $T(\mathbb{C} \times (-))$-algebras for various choices of T.

Example 2.9 When $T = \mathcal{D}$ is the finitely-supported distribution monad, we can study randomized amortized analysis. For example, letting $\Sigma = \mathrm{Id}$, we can implement an alternating amortization technique that flips many coins occasionally, whereas the specification suggests that one coin is flipped per transition.

(i) The specification coalgebra uses state $S = \mathsf{F}1$. Its transition morphism $\sigma : 1 \rightharpoonup \mathsf{F}1$ is a simple Bernoulli distribution, flipping a coin and deciding whether to incur cost accordingly.

(ii) The implementation coalgebra uses state $D = \mathsf{F}(\mathsf{Fin}_k)$ for a fixed k indicating how often to sample. Its transition morphism $\delta : \mathsf{Fin}_k \rightharpoonup \mathsf{F}(\mathsf{Fin}_k)$ is like that of Example 2.1, sampling a k-binomial distribution when its counter is 0 and decrementing the counter otherwise.

The potential function $\Phi : \mathsf{Fin}_k \rightharpoonup \mathsf{F}1$ computes a distribution for each state $d : \mathsf{Fin}_k$. Similar to Example 2.3, we let the generalized potential function be the $(k - d - 1)$-binomial distribution, balancing the number of samples done by the specification and the implementation. Notice that here, though, Φ is not merely computing a number as a potential: it computes an entire distribution via an effectful program. ⌟

Example 2.10 Sometimes, one may consider expected amortized analysis of a randomized data structure, reasoning about the expected cost of a sequence of operations. While this notion is subtle to define explicitly, a reasonable definition of expected amortized analysis falls out of the coalgebraic perspective. If the cost model $\mathbb{C}$ is equipped with the structure of a convex space (*i.e.*, a $\mathcal{D}$-algebra structure), then we have a distributive law that computes the expected value:

$$\mathcal{D}(\mathbb{C} \times (-)) \to \mathbb{C} \times \mathcal{D}(-)$$

Therefore, $\mathbb{C} \times \mathcal{D}(-)$ forms a monad for reasoning about expected cost, and coalgebras over an endofunctor on $\mathbf{Alg}(\mathbb{C} \times \mathcal{D}(-))$ cleanly and precisely specify expected amortized analysis. ⌟

Although the remainder of the paper is compatible with other monads, we work only with the writer monad with numeric costs for simplicity of examples.

3 Lax Amortized Analysis in a Bicategory

In some examples, such as those considered thus far, amortized analysis is precise, where the specification exactly matches the amount of cost used by the implementation. However, it is common for the specified cost to be an upper bound, since some cases may be cheaper than specified due to internal factors that would be difficult to communicate via a concise specification. In the literature, the equation

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

defines the amortized cost $\sigma_{\$}$, which is then given an upper bound. Since here (S, σ) is a specification implementation, though, we treat σ as the upper bound itself, turning the above equation into the inequality

$$\sigma_{\$} \geq \delta_{\$}(d) + \Phi(\delta_{\circ}(d)) - \Phi(d).$$

To achieve this categorically, we augment our development slightly: rather than using coalgebras and morphisms for an endofunctor on a category, we consider *colax* coalgebras and morphisms for an endo-*2*-functor on a *bicategory*. The 2-cells of the bicategory will serve as inequalities for the purpose of cost analysis, drawing inspiration from Grodin *et al.* [10].

For a bicategory $\mathcal{C}$, let $\Sigma : \mathcal{C} \to \mathcal{C}$ be an endo-2-functor. Although the definition of Σ-coalgebra does not change, the definition of morphism between Σ-coalgebras is affected: rather than considering coalgebra morphisms where the given square commutes exactly, we only ask for the square to commute up to a 2-cell.

Definition 3.1 Let (D, δ) and (S, σ) be Σ-coalgebras. A *colax morphism of* Σ*-coalgebras* [3,17,18] from (D, δ) to (S, σ) consists of a morphism $\Phi : D \to S$ that colaxly preserves the Σ-coalgebra structure:

$$\begin{tikzcd}
D \arrow[r, "\delta"] \arrow[d, "\Phi"'] & \Sigma D \arrow[d, "\Sigma\Phi"] \arrow[dl, Rightarrow, shorten=4mm, "\varphi"'] \\
S \arrow[r, "\sigma"'] & \Sigma S
\end{tikzcd}$$

Here, φ is a 2-cell $\Phi \mathbin{;} \sigma \Leftarrow \delta \mathbin{;} \Sigma\Phi$, serving as a proof of inequality. We refer to φ as the *lax generalized amortization condition.* $\lrcorner$

For amortized analysis, we will typically find ourselves using a 2-poset, a bicategory whose 2-cells are mere propositions. In this case, the 2-cell of a morphism is simply a proof that $\Phi \mathbin{;} \sigma \geq \delta \mathbin{;} \Sigma\Phi$. Concretely, we will choose $\mathcal{C} = \mathbf{Poset}$. Following Grodin *et al.* [10], we will use discrete posets everywhere except for the cost component on the writer monad, which will be the natural numbers equipped with the usual increasing ordering, written ω.

Example 3.2 Let $(\mathsf{F}\underline{D}, \delta)$ and $(\mathsf{F}1, \sigma)$ be Id-coalgebras. Then, a colax morphism from $(\mathsf{F}\underline{D}, \delta)$ to $(\mathsf{F}1, \sigma)$ consists of a map $\Phi : \underline{D} \to \omega$ and a 2-cell

$$\Phi \mathbin{;} \sigma \geq \delta \mathbin{;} \Sigma\Phi$$

demonstrating that Φ satisfies the lax generalized amortization condition. In this case, the inequality condition exactly requires that

$$\Phi(d) + \sigma_{\$} \geq \delta_{\$}(d) + \Phi(\delta_{\circ}(d)),$$

which matches the traditional amortization condition

$$\sigma_{\$} \geq \delta_{\$}(d) + \Phi(\delta_{\circ}(d)) - \Phi(d)$$

when the cost model is commutative and has additive inverses. $\lrcorner$

Remark 3.3 Σ-coalgebras and their colax morphisms form a bicategory, where a 2-cell from (Φ, φ) to (Φ', φ') is a 2-cell $\Phi \Rightarrow \Phi'$ in $\mathcal{C}$ along with a coherence condition on φ and φ' [17]. In the restricted case of 2-posets, the coherence condition is trivialized. Then, a 2-cell $\Phi \leq \Phi'$ justifies that Φ expresses the amortization argument with at most as much overhead as Φ'. For instance, in Example 3.2, if Φ is a colax morphism, then so is $\Phi'(d) = \Phi(d) + \$c$ for any $c : \omega$, using the commutativity of addition in ω. Then, $\Phi \leq \Phi'$, since Φ' unnecessarily adds $\$c$ cost.

The advantages and generalizations from the 1-categorical development translate immediately to the 2-categorical setting. For example, the specification carrier can be altered to expose some state, and the monad can be varied to amortize other effects.

Example 3.4 Extending Example 2.5, we can treat a dynamically-resizing array as a stack by adding a method to pop the most-recently-added element. Consider the same signature Σ as in Example 2.7; we provide coalgebra implementations that represent stacks instead of queues:

(i) In order to implement a Σ-coalgebra, we must not only keep track of the length of the stack, but all the elements stored in the stack, since the pop method produces an element of type E. The specification uses state type $S = \mathsf{F}(\mathsf{list}(E))$, storing a specification-level list of elements. Its transition morphism σ is defined as follows:

$\sigma : \mathsf{list}(E) \rightharpoonup \Sigma(\mathsf{list}(E))$
$\sigma\ .\mathsf{push}\ l\ e = \mathsf{charge}\langle \$3 \rangle(\mathsf{ret}(e :: l))$
$\sigma\ .\mathsf{pop}\ [] = \mathsf{inj}_1(\mathsf{ret}(*))$
$\sigma\ .\mathsf{pop}\ (e :: l) = \mathsf{charge}\langle \$2 \rangle(\mathsf{inj}_2(e, \mathsf{ret}(l)))$

The push method behaves as in Example 2.5, and the new pop method charges \$2 for a pop on a nonempty array, terminating if the array is empty.

(ii) The data structure implementation uses a similar state type,

$$\underline{D} = \sum_{n:\mathbb{N}} \mathsf{array}_E[2^n - 1, 2^{n+2} - 1).$$

The only difference is a looser bound on the range of the array length. The push method stays the same, and the new pop method is similar, where init and last get the initial segment and last element of an array, respectively.

$\delta : \underline{D} \to E \to \mathsf{F}D$
$\delta\ .\mathsf{push}\ (n, a)\ e \mid (\mathsf{length}(a) + 1 \equiv 2^{n+2} - 1) = \mathsf{charge}\langle \$(3 + \mathsf{length}(a)) \rangle(\mathsf{ret}(\mathsf{suc}\ n, a \mathbin{+\!\!+} [e]))$
$\delta\ .\mathsf{push}\ (n, a)\ e \mid \text{otherwise} = \mathsf{charge}\langle \$1 \rangle(\mathsf{ret}(n, a \mathbin{+\!\!+} [e]))$
$\delta\ .\mathsf{pop}\ (0, []) = \mathsf{inj}_1(\mathsf{ret}(*))$
$\delta\ .\mathsf{pop}\ (\mathsf{suc}\ n, a) \mid (\mathsf{length}(a) \equiv 2^n - 1) = \mathsf{charge}\langle \$(2 + (\mathsf{length}(a) - 1)) \rangle(\mathsf{inj}_2(\mathsf{last}(a), \mathsf{ret}(n, \mathsf{init}(a))))$
$\delta\ .\mathsf{pop}\ (\mathsf{suc}\ n, a) \mid \text{otherwise} = \mathsf{charge}\langle \$1 \rangle(\mathsf{inj}_2(\mathsf{last}(a), \mathsf{ret}(\mathsf{suc}\ n, \mathsf{init}(a))))$

Let $c(x, y) = \max(2 \cdot (x - y), y - x)$. The program $\Phi(n, a) = \mathsf{charge}\langle \$(c(\mathsf{length}(a), 2^{n+1} - 1)) \rangle(\mathsf{ret}(\mathsf{toList}(a)))$ is a coalgebra morphism representing the amortized analysis, containing the traditional cost-level potential function alongside a behavioral simulation that converts the size-bounded array to a specification-level list. Intuitively, an array is in a lower potential state the closer the number of elements it stores is to the middle of the bounds, since a resize is necessarily many operations away; this is mathematically justified by the potential function. When moving away from this midpoint, the potential function meets the amortization condition. However, when moving towards this midpoint, the potential is decreasing; in this case, we do not need all the cost provided by the specification, and the implementation only meets the specification laxly. Thus, the morphism Φ is not a strict coalgebra morphism, but it is a colax coalgebra morphism, guaranteeing that the amortized implementation is at least as efficient as the specification suggests. $\lrcorner$

Example 3.5 By choosing a non-free carrier, we can "lazily" avoid some computation if its results are never needed while amortizing the cost if the results are eventually needed. A unary counter can be represented by the signature

$$\Sigma A = A \times (\mathbb{N} \rtimes A) \times \mathsf{F}1,$$

where the operations increment the counter, compute the total, and terminate the counter, respectively. As our cost model, we charge \$1 per successor.

(i) The specification uses carrier $S = \mathsf{F}\mathbb{N}$, storing a single natural number.

$$\begin{aligned}
&\sigma : \mathsf{F}\mathbb{N} \to \Sigma(\mathsf{F}\mathbb{N}) \\
&\sigma \ .\mathsf{increment}\ s = \mathsf{bind}\, n \leftarrow s\, \mathsf{in}\, \mathsf{charge}\langle \$1 \rangle(\mathsf{ret}(\mathsf{suc}\ n)) \\
&\sigma \ .\mathsf{total}\ s = \mathsf{bind}\, n \leftarrow s\, \mathsf{in}\, (n, \mathsf{ret}(n)) \\
&\sigma \ .\mathsf{terminate}\ s = \mathsf{bind}\, n \leftarrow s\, \mathsf{in}\, \mathsf{ret}(*)
\end{aligned}$$

Note that we explicitly write $\mathsf{F}\mathbb{N} \to \Sigma(\mathsf{F}\mathbb{N})$ instead of our usual shorthand $\mathbb{N} \rightharpoonup \Sigma(\mathsf{F}\mathbb{N})$ to emphasize similarity with the following implementation.

(ii) The data structure implementation has carrier D as the following comma object, a subobject of the lazy pair $\mathsf{F}\mathbb{N} \times \mathsf{F}1$ where the cost of the $\mathsf{F}1$ component is at most the cost of the $\mathsf{F}\mathbb{N}$ component:

$$\begin{tikzcd}
D \arrow[r] \arrow[d] \arrow[dr, phantom, "\leq"] & \mathsf{F}\mathbb{N} \arrow[d, "\mathsf{F}*"] \\
\mathsf{F}1 \arrow[r, equal] & \mathsf{F}1
\end{tikzcd}$$

The implementation stores this lazy pair of $\mathsf{F}\mathbb{N}$ and $\mathsf{F}1$ computations to "hedge its bets": in case the final total is never used, the computation of type $\mathsf{F}1$ will waste the cost from the unused successors.

$$\begin{aligned}
&\delta : D \to \Sigma(D) \\
&\delta \ .\mathsf{increment}\ d = ((\mathsf{bind}\, n \leftarrow d\ .\mathsf{proj}_1\, \mathsf{in}\, \mathsf{charge}\langle \$1 \rangle(\mathsf{ret}(\mathsf{suc}\ n))), (d\ .\mathsf{proj}_2)) \\
&\delta \ .\mathsf{total}\ d = \mathsf{bind}\, n \leftarrow d\ .\mathsf{proj}_1\, \mathsf{in}\, (n, (\mathsf{ret}(n), \mathsf{ret}(*))) \\
&\delta \ .\mathsf{terminate}\ d = d\ .\mathsf{proj}_2
\end{aligned}$$

The increment operation keeps the $\mathsf{F}\mathbb{N}$ and $\mathsf{F}1$ projections independent, accumulating one cost in the $\mathsf{F}\mathbb{N}$ component while leaving the $\mathsf{F}1$ component as-is. However, to implement the totaling operation, the first projection must be used to compute the eager $\mathbb{N}$ before the lazy "hedging" can take place. By the algebra structure on a product, all cost stored in the $\mathsf{F}\mathbb{N}$ component will flow to *both* components going forward: since we needed the total after all, even the amortized cost of the second component of type $\mathsf{F}1$ must be increased. In the termination operation, though, the opposite situation occurs: we don't need the first component after all, so by taking the second projection, we may save on cost incurred by increments since the last total was computed.

The first projection $\Phi : D \to \mathsf{F}\mathbb{N}$ is a colax morphism from (D, δ) to (S, σ), where the lax amortization condition in the termination case is justified by the cost inequality guaranteed by the comma object. $\lrcorner$

In the remainder of the paper, we will work with 1-categories for simplicity, although the constructions readily generalize to the 2-categorical setting.

4 Splitting and Combining Potential

In more complex amortized analyses, the behavior of an amortized data structure can branch, breaking the data structure into multiple parts or combining multiple instances. As discussed by Okasaki [23, §5.3], in this scenario, the amortization condition should consider the sum of the potentials of the input and output states. Informally, we say

$$\sigma_\$ = \delta_\$(\mathit{Input}) + \sum_{d' \in \delta_\circ(\mathit{Input})} \Phi(d') - \sum_{d \in \mathit{Input}} \Phi(d),$$

where *Input* is a set of input states and $\delta_\circ(\mathit{Input})$ is the corresponding set of output states. In the case that there is a single input and a single output, this condition is equivalent to the usual condition.

To make sense this in our presentation, we consider two additional structures. First, we represent multiple data structure states in parallel via a monoidal product on $\mathcal{C}$, recovering and formalizing the idea of summing the potentials of states for the typical case of free algebra carriers. Then, we generalize signatures from endofunctors to endo*profunctors* to allow for multiple parallel inputs.

4.1 Parallel States with Additive Potential

To represent multiple simultaneous output states, we require that $\mathcal{C}$ come equipped with a symmetric monoidal structure $(\top, \otimes)$. Then, for example, a method that splits the amortized data structure into two parts will be represented by the signature by $\Sigma A = A \otimes A$. Such a symmetric monoidal structure $(\top, \otimes)$ exists in the category of algebras $\mathcal{C} = \mathbf{Alg}(T)$ when the monad T is strong and commutative. The writer monad $\mathbb{C} \times (-)$ is commutative exactly when the monoid on $\mathbb{C}$ is commutative; this is often a reasonable assumption in the setting of cost analysis. When T is commutative, the adjunction $\mathsf{F} \dashv \mathsf{U} : \mathbf{Alg}(T) \to \mathbf{Set}$ is also lax monoidal, which implies that F is a strong monoidal functor:

$$\top \cong \mathsf{F}1$$
$$\mathsf{F}X \otimes \mathsf{F}Y \cong \mathsf{F}(X \times Y)$$

The forward direction of the first isomorphism adds together the potential stored in the components. Thus, we can support the branching generalization of amortized analysis via the monoidal product.

Example 4.1 Let $\Sigma A = A \otimes A$, representing a single method that splits a data structure into components. Suppose $(\mathsf{F}\underline{D}, \delta)$ and $(\mathsf{F}1, \sigma)$ are Σ-coalgebras. To prove that the former is an amortized implementation of a latter, we give a potential function $\Phi : \underline{D} \to \mathbb{C}$ such that the following square commutes:

$$\begin{tikzcd}
\mathsf{F}\underline{D} \arrow[r, "\delta"] \arrow[d, "\Phi"] & \mathsf{F}\underline{D} \otimes \mathsf{F}\underline{D} \arrow[d, "\Phi\otimes\Phi"] \\
\mathsf{F}1 \arrow[r, "\sigma"] & \mathsf{F}1 \otimes \mathsf{F}1
\end{tikzcd}$$

Since $\mathsf{F}1 \otimes \mathsf{F}1 \cong \mathsf{F}1$, the behavioral component of both paths are trivial. The condition on costs can be stated as follows, using the fact that the map $\mathsf{F}1 \otimes \mathsf{F}1 \to \mathsf{F}1$ adds costs:

$$\Phi(d) + \sigma_{\$} = \delta_{\$}(d) + (\Phi(\delta_{\circ}(d)_1) + \Phi(\delta_{\circ}(d)_2))$$

Equivalently, we may write:

$$\Phi(d) + \sigma_{\$} = \delta_{\$}(d) + \sum_{i \in \{1,2\}} \Phi(\delta_{\circ}(d)_i)$$

Returning to the informal amortization condition, this makes precise the notion of having multiple states output states, using the monoidal product to capture the addition of potentials. $\lrcorner$

4.2 Algebraic and Coalgebraic Operations via Profunctors

Some data structures support "algebraic" operations that, for example, may combine multiple instances. Such situations can be described by the use of Σ-*algebras* rather than coalgebras and their (colax) morphisms. However, in general, an operation may have many inputs and outputs. For example, an operation may take two instances of a data structure and produce two updated instances, $(A \otimes A) \to (A \otimes A)$. As defined, a coalgebra for an endofunctor Σ consists of a carrier D and a transition morphism $\delta : D \to \Sigma D$ that takes a single state and provides possibilities for transition. To support such "binary methods" (as in object-oriented programming [29]) with multiple inputs while still retaining coalgebraic operations with multiple outputs, we generalize to coalgebras for an endoprofunctor Σ, which will provide the possibility for multiple input and output states by supporting both covariant and contravariant uses of the carrier.

When $(\mathcal{C}, \top, \otimes)$ is additionally equipped with a closure $\multimap : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathcal{C}$, one may naively attempt to use to support a method with two inputs (and two outputs) using signature $\Sigma A = A \multimap (A \otimes A)$, so that a Σ-coalgebra $D \to \Sigma D$ is equivalent to a map $D \otimes D \to D \otimes D$. However, this definition of Σ is not functorial: A is used in a contravariant position. To address this, we generalize from endofunctors to endoprofunctors, analogous to generalizing from functions to relations or from coinductive types to existential types. A profunctor $\mathcal{C} \nrightarrow \mathcal{D}$ is a functor $\mathcal{D}^{\mathrm{op}} \times \mathcal{C} \to \mathbf{Set}$. To represent an arbitrary signature for an abstract data type, we will use an endoprofunctor $\Sigma : \mathcal{C} \nrightarrow \mathcal{C}$ in place of an endofunctor, taking in both a covariant and a contravariant copies of what will ultimately be A. For example, if we let

$$\Sigma(A^-, A^+) = (A^- \otimes A^-) \multimap (A^+ \otimes A^+),$$

we describe a signature that takes in a pair of states and produces a new pair of states. We now recall the definition of a coalgebra for an endoprofunctor, using the bicategorical generalization coalgebra.

Definition 4.2 Let Σ be an endoprofunctor. A Σ-coalgebra is a pair (D, δ) of a carrier object $D : 1 \nrightarrow \mathcal{C}$ and a transition morphism $\delta : D \to \Sigma \circ D$ [22]. ⌟

Note that profunctors $1 \nrightarrow \mathcal{C}$ are equivalent to presheaves $\widehat{\mathcal{C}}$. For cost analysis, we will continue to use $\mathcal{C} \triangleq \mathbf{Alg}(\mathbb{C} \times (-))$. Coalgebras for endoprofunctors encompass coalgebras for endofunctors.

Theorem 4.3 *Let $\Sigma : \mathcal{C} \to \mathcal{C}$ be an arbitrary endofunctor, and define $\widetilde{\Sigma} : \mathcal{C} \nrightarrow \mathcal{C}$ by:*

$$\widetilde{\Sigma}(A^-, A^+) = A^- \multimap \Sigma A^+$$

Then, a Σ-coalgebra with carrier A is equivalent to a $\widetilde{\Sigma}$-coalgebra with carrier $\text{よ} A$.

Proof. By the Yoneda lemma, maps $\delta : \text{よ} D \to \widetilde{\Sigma} \circ \text{よ} D$ are equivalent to elements of $\widetilde{\Sigma}(D, D)$. By definition, we have $\widetilde{\Sigma}(D, D) = D \multimap \Sigma D$, precisely the definition of a Σ-coalgebra transition morphism. □

While coalgebras over endofunctors are definable as coalgebras over endoprofunctors, this new environment allows more flexibility in the contravariant position. Morphisms of coalgebras over an endoprofunctor generalize morphisms of coalgebras over an endofunctor, as well.

Definition 4.4 Let (D, δ) and (S, σ) be Σ-coalgebras, where Σ is an endoprofunctor. A morphism of Σ-coalgebras from (D, δ) to (S, σ) consists of a morphism $\Phi : D \to S$ that preserves the Σ-coalgebra structure, as before:

$$\begin{tikzcd}
D \arrow[r, "\delta"] \arrow[d, "\Phi"] & \Sigma \circ D \arrow[d, "\Sigma \circ \Phi"] \\
S \arrow[r, "\sigma"] & \Sigma \circ S
\end{tikzcd}$$

Note that D and S are presheaves in $\widehat{\mathcal{C}}$ and Φ is a morphism of presheaves. ⌟

When $D = \text{よ} D_0$ and $S = \text{よ} S_0$, the situation becomes simpler. Since the Yoneda embedding is fully faithful, it is equivalent to give a map $\Phi : D_0 \to S_0$. By the Yoneda lemma, the structure preservation requirement can be simplified to the following:

$$\begin{tikzcd}
1 \arrow[r, "\delta"] \arrow[d, "\sigma"] & \Sigma(D_0, D_0) \arrow[d, "{\Sigma(D_0, \Phi)}"] \\
\Sigma(S_0, S_0) \arrow[r, "{\Sigma(\Phi, S_0)}"] & \Sigma(D_0, S_0)
\end{tikzcd}$$

Viewing the covariant and contravariant positions as outputs and inputs, respectively, this formalizes of the amortization condition with multiple inputs and outputs. We demonstrate this via a concrete example.

Example 4.5 Define endoprofunctor $\Sigma(A^-, A^+) = (A^- \otimes A^-) \multimap (A^+ \otimes A^+)$, representing a method that takes two states of a data structure and provides two new states. Suppose $(\text{よ}(\mathsf{F}\underline{D}), \delta : \Sigma(\mathsf{F}\underline{D}, \mathsf{F}\underline{D}))$ and $(\text{よ}(\mathsf{F}1), \sigma : \Sigma(\mathsf{F}1, \mathsf{F}1))$ are Σ-coalgebras. Then, the above condition simplifies to:

$$(\Phi(d_1) + \Phi(d_2)) + \sigma_\$ = \delta_\$(d) + (\Phi(\delta_\circ(d)_1) + \Phi(\delta_\circ(d)_2))$$

where $d = (d_1, d_2) : \underline{D} \times \underline{D}$. Equivalently, we may write:

$$\sum_{i \in \{1,2\}} \Phi(d_i) + \sigma_\$ = \delta_\$(d) + \sum_{i \in \{1,2\}} \Phi(\delta_\circ(d)_i)$$

Up to our usual movement of the starting potential across the equation, this is precisely the generalized amortization equation given above, using the structure Σ to make precise the informal *Input* notion. ⌟

Using a monoidal product, we allowed multiple states to exist simultaneously, adding any potential they contain. Then, using profunctors, we generalized the notion of signature to support arbitrary contravariant data, allowing arbitrarily many inputs and outputs to a method.

5 Composition of Amortized Data Structures

Thus far, we have considered coalgebra morphisms in isolation, each showing that one data structure implementation matches a specification up to amortization. Using the fact that Σ-coalgebras and coalgebra morphisms form a category, we may compose potential functions to support different levels of amortized abstraction.

Example 5.1 Recall Examples 2.1 and 2.3, where the specification (S, σ) purports to incur \$1 of cost every operation while really the implementation (D, δ) incurs \$8 of cost every eight operations. Alternatively, we may view (D, δ) as the specification, which can be implemented by an even more amortized scheme. For example, we may define a coalgebra (D', δ') that incurs \$16 of cost every sixteen operations, which is an amortized implementation of (D, δ) via a new potential function $\Phi' : (D', \delta') \to (D, \delta)$. Of course, we may then compose these potential functions,

$$(D', \delta') \xrightarrow{\Phi'} (D, \delta) \xrightarrow{\Phi} (S, \sigma),$$

showing that (D', δ') is an implementation of the original specification (S, σ) after all. ⌟

More commonly, we may wish to compose coalgebras with different signatures, using one amortized data structure to implement another. For example, we may wish to use a pair of stacks, with an amortized implementation in terms of arrays in Example 3.4, to implement a queue, as given in Example 2.7. The coalgebras and morphisms for pairs of stacks and queues exist in different categories, $\mathbf{Coalg}(\Sigma_\mathrm{s} \times \Sigma_\mathrm{s})$ and $\mathbf{Coalg}(\Sigma_\mathrm{q})$, where both Σ_s and Σ_q are (coincidentally!) both the signature from Example 3.4 extended with a method $\mathbb{N} \rtimes A$ for computing the number of elements stored in the data structure.

Rather than viewing each category of coalgebras $\mathbf{Coalg}(\Sigma)$ in isolation for a given signature Σ, we may think of $\mathbf{Coalg}(-) : \mathbf{Fun}(\mathbf{Alg}(T), \mathbf{Alg}(T)) \to \mathbf{Cat}$ as an indexed category whose (covariant) Grothendieck construction $\int^\Sigma \mathbf{Coalg}(\Sigma)$ is the category with objects $(\Sigma_D, (D, \delta))$, where (D, δ) is a Σ_D-coalgebra. In other words, an object consists of a signature Σ_D along with an implementation of that signature. A morphism from $(\Sigma_D, (D, \delta))$ to $(\Sigma_S, (S, \sigma))$ consists of a natural transformation $\varphi : \Sigma_D \to \Sigma_S$ implementing the operations of Σ_S in terms of the operations of Σ_D, along with a Σ_S-coalgebra morphism from $(D, D \xrightarrow{\delta} \Sigma_D D \xrightarrow{\varphi D} \Sigma_S D)$ to (S, σ) performing an amortized analysis on the φ-translated implementation.

Example 5.2 Let Σ_s be the signature functor from Example 3.4 for stacks, and let $\Sigma_\mathrm{c} A = A \times (\mathsf{F}1 + A)$ be the signature of for a counter with successor and predecessor methods. Let D be the carrier used to implement stacks. The amortized analysis of stacks induces a morphism

$$(\Sigma_\mathrm{s}, (D, \delta)) \to (\Sigma_\mathrm{s}, (\mathsf{F}(\mathsf{list}(E)), \sigma))$$

whose translation component $\Sigma_\mathrm{s} \to \Sigma_\mathrm{s}$ is the identity. Separately, let $(\mathsf{F}\mathbb{N}, \sigma_\mathrm{counter})$ be a Σ_c-coalgebra specification similar to the Σ_s coalgebra $(\mathsf{F}(\mathsf{list}(E)), \sigma)$, keeping the same costs but only storing the number of elements stored rather than the elements themselves. We can give a morphism to $(\mathsf{F}\mathbb{N}, \sigma_{\text{counter}})$

$$(\Sigma_{\text{s}}, (\mathsf{F}(\mathsf{list}(E)), \sigma)) \to (\Sigma_{\text{c}}, (\mathsf{F}\mathbb{N}, \sigma_{\text{counter}}))$$

that implements counters in terms of stacks of a sentinel value $e_0 : E$. Composing these morphisms, we get a map

$$(\Sigma_{\text{s}}, (D, \delta)) \to (\Sigma_{\text{c}}, (\mathsf{F}\mathbb{N}, \sigma_{\text{counter}}))$$

that implements natural number counters in terms of a dynamically resizing array, mixing signatures. ⌟

To implement an amortized queue as a pair of amortized stacks, we must first be able to pair two coalgebras. Fortunately, when restricted to signature functors equipped with a tensorial strength (which includes all signatures considered in this work), the indexed category $\mathbf{Coalg}(-)$ is lax monoidal.

Theorem 5.3 *Let $\mathcal{C}$ be a symmetric monoidal category. Then, $\mathbf{Coalg}(-) : \mathbf{Strong}(\mathcal{C}, \mathcal{C}) \to \mathbf{Cat}$ is a lax monoidal pseudofunctor, lifting the symmetric monoidal structure $(\top, \otimes)$ to coalgebras.*

Proof. For coalgebras $(D_1, \delta_1) : \mathbf{Coalg}(\Sigma_1)$ and $(D_2, \delta_2) : \mathbf{Coalg}(\Sigma_2)$, we define $(D_1, \delta_1) \otimes (D_2, \delta_2)$ to have carrier $D \triangleq D_1 \otimes D_2$ and transition morphism

$$D \xrightarrow{\Delta} D \times D \xrightarrow{(\delta_1 \otimes D_2) \times (D_1 \otimes \delta_2)} (\Sigma_1 D_1 \otimes D_2) \times (D_1 \otimes \Sigma_2 D_2) \xrightarrow{t_1 \times t_2} \Sigma_1 D \times \Sigma_2 D$$

where t_1 and t_2 are strengths for Σ_1 and Σ_2, respectively. The unit is $(\top, \top \xrightarrow{*} 1) : \mathbf{Coalg}(1)$. □

If $(S_{\text{s}}, \sigma_{\text{s}}) : \mathbf{Coalg}(\Sigma_{\text{s}})$ is the specification coalgebra for a stack, then $(S_{\text{s}}, \sigma_{\text{s}}) \otimes (S_{\text{s}}, \sigma_{\text{s}}) : \mathbf{Coalg}(\Sigma_{\text{s}} \times \Sigma_{\text{s}})$ is the compound specification for a pair of stacks. We may then hope to give a morphism

$$(\Sigma_{\text{s}} \times \Sigma_{\text{s}}, (S_{\text{s}}, \sigma_{\text{s}}) \otimes (S_{\text{s}}, \sigma_{\text{s}})) \to (\Sigma_{\text{q}}, (S_{\text{q}}, \sigma_{\text{q}}))$$

implementing amortized queues in terms of a pair of stacks. The translation morphism $\Sigma_{\text{s}} \times \Sigma_{\text{s}} \to \Sigma_{\text{q}}$ would have to describe each queue operation in terms of a stack operation. However, implementing a queue operation may require more than one stack operation, popping all the elements of the “inbox” stack in the case the “outbox” stack is empty. To allow each queue operation to perform arbitrarily many stack operations, we instead treat $\mathbf{Coalg}(-)$ as category indexed in the coKleisli category of the cofree comonad comonad, $(-)^\omega : \mathbf{Fun}(\mathbf{Alg}(T), \mathbf{Alg}(T)) \to \mathbf{Fun}(\mathbf{Alg}(T), \mathbf{Alg}(T))$. This approach is the formal dual of recent work by Grodin and Spivak [11] about algebraic effect handlers; in this sense, a translation morphism can be viewed as a “coalgebraic coeffect cohandler”.

Definition 5.4 The pseudofunctorial action of the indexed category $\mathbf{Coalg}(-) : \mathbf{coKl}((-)^\omega) \to \mathbf{Cat}$ is given by post-composition of coalgebra maps $\delta : D \to \Sigma_D^\omega D$ with the coKleisli extension of a translation morphism $\varphi : \Sigma_D^\omega \to \Sigma_S$, written $\varphi^\dagger : \Sigma_D^\omega \to \Sigma_S^\omega$.

$$\begin{aligned}
&\mathbf{Coalg}_0(\Sigma) = \mathbf{Coalg}^\omega(\Sigma^\omega) \\
&\mathbf{Coalg}_1(\varphi)_0(D, D \xrightarrow{\delta} \Sigma_D^\omega D) = (D, D \xrightarrow{\delta} \Sigma_D^\omega D \xrightarrow{\varphi^\dagger D} \Sigma_S^\omega D) \\
&\mathbf{Coalg}_1(\varphi)_1((D, \delta) \xrightarrow{\Phi} (S, \sigma)) = (D, (\delta \mathbin{;} \varphi^\dagger D)) \xrightarrow{\Phi} (S, (\sigma \mathbin{;} \varphi^\dagger S))
\end{aligned}$$

We write $\mathbf{Coalg}^\omega(\Sigma^\omega)$ for the category of comonad coalgebras over the cofree comonad Σ^ω to simplify composition. However, this category is equivalent to $\mathbf{Coalg}(\Sigma)$, so the object part is the same as when indexed over the category of endofunctors. ⌟

In the fibered category $\int^{\Sigma:\mathbf{coKl}((-)^\omega)} \mathbf{Coalg}(\Sigma)$, the objects are the same as before, but the signature translation of a morphism from $(\Sigma_D, (D, \delta))$ to $(\Sigma_S, (S, \sigma))$ now uses the cofree comonad on Σ_D in its domain, $\varphi : \Sigma_D^\omega \to \Sigma_S$, translating the operations of Σ_S to finitely many Σ_D operations.

Example 5.5 In this category, we may implement a morphism

$$(\Sigma_{\mathrm{s}} \times \Sigma_{\mathrm{s}}, (S_{\mathrm{s}}, \sigma_{\mathrm{s}}) \otimes (S_{\mathrm{s}}, \sigma_{\mathrm{s}})) \to (\Sigma_{\mathrm{q}}, (S_{\mathrm{q}}, \sigma_{\mathrm{q}})),$$

using amortized stacks to implement amortized queues. The translation morphism $\Sigma_{\mathrm{s}} \times \Sigma_{\mathrm{s}} \to \Sigma_{\mathrm{q}}$ is analogous to the behavior of the implementation coalgebra in Example 2.7 but generic in stack implementation, and the coalgebra morphism is precisely as in Example 2.7. Pre-composing this with the morphism from Example 3.4, we implement the queue specification via a pair of stacks up to amortization, which in turn are implemented as arrays up to amortization. ⌟

The techniques considered here generalize to bicategories (Section 3) and mixed-variance signatures given by profunctors (Section 4).

6 Conclusion

In this work, we observed that the condition imposed on a potential function in amortized analysis is generalized by the commutativity condition for a coalgebra in the category of writer monad algebras. Using this perspective, we gave simple, clear accounts of examples, and we generalized amortized analysis to other effectful settings. We expanded this definition to include branching amortization given a commutative cost model using profunctors, and we investigated the composition of amortized data structures via the indexed category of coalgebras. Finally, we observed that this development makes sense in a bicategory, where colax coalgebra morphisms give a more relaxed and practically useful definition of amortization.

The key ideas and some examples presented have been mechanized inside Calf [21,10] in the Agda proof assistant. Basic definitions, such as functor, coalgebra, and (colax) coalgebra morphism, are formulated explicitly, with the colax commutativity condition expressed using the program inequality of Grodin *et al*. [10]. The analyses themselves are structured analogous to the on-paper reasoning presented here, with the added formality required for mechanized proof.

6.1 Related Work

This development principally expands upon the early work of Grodin and Harper [9], which first proposed to use coalgebraic techniques to reason about amortized analysis. Amortized analysis was first characterized by Tarjan [28], providing a technique for describing the cost of data structure operations that takes into account the evolving state of the data structure and considers cost in aggregate, in contrast with algorithmic worst-case upper bounds on a per-operation basis. Representing abstract data types as coalgebras and simulations as coalgebra morphisms has been well studied; see, *e.g.*, Jacobs [15,16] for an overview. Formalization of cost analysis using a writer monad has been studied extensively; see Niu *et al*. [21] for a comprehensive review of the literature. Mechanizations of amortized cost analysis has also been pursued, but in the traditional algebraic form [24,20,21]. Type theories for automatic cost inference, such as AARA [13,12], are based on the potential functions of the physicist's method of amortized analysis.

6.2 Future Work

While the cost algebra carriers of most coalgebras considered are free (*i.e.*, of the form $\mathsf{F}X$), in Example 3.5 we use a product of free algebras to lazily avoid some computation. Although this example is somewhat contrived and the laziness does not affect the asymptotic complexity of operations, we leave it as future work to determine if such methods could be used to save cost in a more realistic data structure. Similar observations connecting amortization to laziness have been made by Okasaki [23, §6]; however, his work uses memoized suspensions to amortize in a call-by-value setting, whereas here we use linearity at the level of computation types in call-by-push-value to address this issue. Future work may make this connection between amortization, laziness, and linearity more precise.

When using endoprofunctors as signatures, though, the carriers of all coalgebras considered are representable (*i.e.*, of the form $よ A$); we hope to investigate in future work whether non-representable carriers would be of use for amortized analysis.

In the pioneering works on amortized analysis, Sleator and Tarjan [28,25] describe the "move-to-front" paging algorithm and prove that it is competitive relative to *any* other algorithm. This argument, while also using amortization, does not immediately appear to fit into our coalgebraic framework. The potential is computed by comparing the move-to-front algorithm to another algorithm; we conjecture that the potential is behaving like a generalized bisimulation relation rather than a morphism. While we view amortized analysis as a directed simulation here (*i.e.*, a morphism of coalgebras), we suspect that a double categorical approach may be valuable for understanding a symmetric generalization more akin to bisimulation. Cospans of coalgebras have been used to generalize bisimulations [27], which can serve as vertical morphisms to add to the bicategory of colax coalgebras. Additionally, Goncharov *et al.* [7] propose a double category of (co)lax coalgebras of a lax-double functor, generalizing from a fundamentally double categorical perspective.

In the present work, we consider coalgebras in a category of algebras. This construction appears more symmetrically in the development of mathematical operational semantics, there called a bialgebra, representing a transition system in the style of operational semantics. We leave it to future investigation to understand the connections between amortized analysis and operational semantics.

In our formalization within Calf, the notion of coalgebra morphism is defined explicitly, in contrast with the notion of cost algebra that is built in as a primitive. We hope to incorporate coalgebras and coalgebra morphisms more fundamentally, allowing amortized analysis to be more seamless within the language rather than embedded.

Acknowledgements

The authors wish to thank Max New, Yue Niu, and David Spivak for insightful conversations that directly impacted the direction of this work and Zachary Battleman, Runming Li, Parth Shastri, and Jonathan Sterling for fruitful adjacent collaboration that contributed broader inspiration. Additionally, the authors thank the anonymous thank the anonymous reviewers for their thoughtful comments. 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.

References

[1] Abel, A., B. Pientka, D. Thibodeau and A. Setzer, *Copatterns: Programming infinite structures by observations*, ACM SIGPLAN Notices **48**, pages 27–38 (2013), ISSN 0362-1340.
https://doi.org/10.1145/2480359.2429075

[2] Benton, P. N., *A mixed linear and non-linear logic: Proofs, terms and models*, in: L. Pacholski and J. Tiuryn, editors, *Computer Science Logic*, Lecture Notes in Computer Science, pages 121–135, Springer, Berlin, Heidelberg (1995), ISBN 978-3-540-49404-1.
https://doi.org/10.1007/BFb0022251

[3] Blackwell, R., G. M. Kelly and A. J. Power, *Two-dimensional monad theory*, Journal of Pure and Applied Algebra **59**, pages 1–41 (1989), ISSN 0022-4049.
https://doi.org/10.1016/0022-4049(89)90160-6

[4] Burton, F. W., *An efficient functional implementation of FIFO queues*, Information Processing Letters **14**, pages 205–206 (1982), ISSN 0020-0190.
https://doi.org/10.1016/0020-0190(82)90015-1

[5] Egger, J., R. E. Møgelberg and A. Simpson, *Enriching an Effect Calculus with Linear Types*, in: E. Grädel and R. Kahle, editors, *Computer Science Logic*, Lecture Notes in Computer Science, pages 240–254, Springer, Berlin, Heidelberg (2009), ISBN 978-3-642-04027-6.
https://doi.org/10.1007/978-3-642-04027-6_19

[6] Egger, J., R. E. Møgelberg and A. Simpson, *The enriched effect calculus: Syntax and semantics*, Journal of Logic and Computation **24**, pages 615–654 (2014), ISSN 1465-363X.
https://doi.org/10.1093/logcom/exs025

[7] Goncharov, S., D. Hofmann, P. Nora, L. Schröder and P. Wild, *Kantorovich Functors and Characteristic Logics for Behavioural Distances* (2022). 2202.07069.
http://arxiv.org/abs/2202.07069

[8] Gries, D., *The Science of Programming*, Springer New York (1989), ISBN 978-0-387-96480-5.

[9] Grodin, H. and R. Harper, *Amortized Analysis via Coinduction (Early Ideas)*, in: P. Baldan and V. de Paiva, editors, *10th Conference on Algebra and Coalgebra in Computer Science (CALCO 2023)*, volume 270 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:6, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023), ISBN 978-3-95977-287-7, ISSN 1868-8969.
https://doi.org/10.4230/LIPIcs.CALCO.2023.23

[10] Grodin, H., Y. Niu, J. Sterling and R. Harper, *Decalf: A Directed, Effectful Cost-Aware Logical Framework*, Proceedings of the ACM on Programming Languages **8**, pages 10:273–10:301 (2024).
https://doi.org/10.1145/3632852

[11] Grodin, H. and D. Spivak, *Poly-morphic effect handlers* (2024).
https://topos.site/blog/2024/01/poly-morphic-effect-handlers/

[12] Hoffmann, J. and S. Jost, *Two decades of automatic amortized resource analysis*, Mathematical Structures in Computer Science **32**, pages 729–759 (2022), ISSN 0960-1295, 1469-8072.
https://doi.org/10.1017/S0960129521000487

[13] Hofmann, M. and S. Jost, *Static prediction of heap space usage for first-order functional programs*, ACM SIGPLAN Notices **38**, pages 185–197 (2003), ISSN 0362-1340.
https://doi.org/10.1145/640128.604148

[14] Hood, R. and R. Melville, *Real-time queue operations in pure LISP*, Information Processing Letters **13**, pages 50–54 (1981), ISSN 0020-0190.
https://doi.org/10.1016/0020-0190(81)90030-2

[15] Jacobs, B., *Objects And Classes, Co-Algebraically*, in: B. Freitag, C. B. Jones, C. Lengauer and H.-J. 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), ISBN 978-1-4613-1437-0.
https://doi.org/10.1007/978-1-4613-1437-0_5

[16] Jacobs, B., *Introduction to Coalgebra*, Cambridge University Press (2017), ISBN 978-1-107-17789-5.

[17] Lack, S., *Limits for Lax Morphisms*, Applied Categorical Structures **13**, pages 189–203 (2005), ISSN 1572-9095.
https://doi.org/10.1007/s10485-005-2958-5

[18] Lack, S. and M. Shulman, *Enhanced 2-categories and limits for lax morphisms*, Advances in Mathematics **229**, pages 294–356 (2012), ISSN 0001-8708.
https://doi.org/10.1016/j.aim.2011.08.014

[19] Levy, P. B., *Call-By-Push-Value: A Functional/Imperative Synthesis*, Springer Netherlands, Dordrecht (2003), ISBN 978-94-010-3752-5 978-94-007-0954-6.
https://doi.org/10.1007/978-94-007-0954-6

[20] Nipkow, T. and H. Brinkop, *Amortized Complexity Verified*, Journal of Automated Reasoning **62**, pages 367–391 (2019), ISSN 1573-0670.
https://doi.org/10.1007/s10817-018-9459-3

[21] Niu, Y., J. Sterling, H. Grodin and R. Harper, *A Cost-Aware Logical Framework*, Proceedings of the ACM on Programming Languages **6**, pages 9:1–9:31 (2022).
https://doi.org/10.1145/3498670

[22] nLab authors, *algebra for a profunctor*, https://ncatlab.org/nlab/show/algebra+for+a+profunctor (2024). Revision 11.

[23] Okasaki, C., *Purely Functional Data Structures*, Cambridge University Press (1999), ISBN 978-0-521-66350-2.

[24] Plotkin, G. and J. Power, *Tensors of Comodels and Models for Operational Semantics*, Electronic Notes in Theoretical Computer Science **218**, pages 295–311 (2008), ISSN 1571-0661.
https://doi.org/10.1016/j.entcs.2008.10.018

[25] Sleator, D. D. and R. E. Tarjan, *Amortized efficiency of list update and paging rules*, Communications of the ACM **28**, pages 202–208 (1985), ISSN 0001-0782.
https://doi.org/10.1145/2786.2793

[26] Sleator, D. D. and R. E. Tarjan, *Self-adjusting binary search trees*, Journal of the ACM **32**, pages 652–686 (1985), ISSN 0004-5411.
https://doi.org/10.1145/3828.3835

[27] Staton, S., *Relating coalgebraic notions of bisimulation*, Logical Methods in Computer Science **Volume 7, Issue 1**, page 670 (2011), ISSN 1860-5974. 1101.4223.
https://doi.org/10.2168/LMCS-7(1:13)2011

[28] Tarjan, R. E., *Amortized Computational Complexity*, SIAM Journal on Algebraic Discrete Methods **6**, pages 306–318 (1985), ISSN 0196-5212.
https://doi.org/10.1137/0606031

[29] Tews, H., *Coalgebras for Binary Methods: Properties of Bisimulations and Invariants*, RAIRO - Theoretical Informatics and Applications **35**, pages 83–111 (2001), ISSN 0988-3754, 1290-385X.
https://doi.org/10.1051/ita:2001110