Amortized Analysis via Coinduction

Harrison Grodin, j.w.w. Robert Harper

June 19, 2023

Carnegie Mellon University

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.

Table of contents

Goal

Understand *amortized analysis* in *call-by-push-value*/**calf**, using *coinduction*.

Call-By-Push-Value and calf

Type Polarity

In call-by-push-value, types are separated into two sorts:

Type Polarity

In call-by-push-value, types are separated into two sorts:

Positive/Value Types

$$A, B, C ::=$$

$$0 \quad A + B$$

$$1 \quad A \times B$$

$$\mu(A.\ B(A))$$

Type Polarity

In call-by-push-value, types are separated into two sorts:

Positive/Value Types

$$A, B, C ::= \quad 0 \quad A + B \quad 1 \quad A \times B \quad \mu(A.\ B(A))$$

Interpreted in **Set**.

Type Polarity

In call-by-push-value, types are separated into two sorts:

Positive/Value Types

$$A, B, C ::= 0 \quad A + B \quad 1 \quad A \times B \quad \mu(A.\ B(A))$$

Negative/Computation Types

$$X, Y, Z ::= 1 \quad X \times Y \quad A \to X \quad \nu(X.\ Y(X))$$

Interpreted in **Set**.

Type Polarity

In call-by-push-value, types are separated into two sorts:

Positive/Value Types

$$A, B, C ::= 0 \quad A + B \quad 1 \quad A \times B \quad \mu(A.\ B(A))$$

Interpreted in **Set**.

Negative/Computation Types

$$X, Y, Z ::= 1 \quad X \times Y \quad A \to X \quad \nu(X.\ Y(X))$$

Interpreted in $\mathbf{Set}^{\mathsf{T}}$, for monad T.

Type Polarity

In call-by-push-value, types are separated into two sorts:

Positive/Value Types

$$A, B, C ::= \mathsf{U}X \mid 0 \mid A + B \mid 1 \mid A \times B \mid \mu(A.\ B(A))$$

Interpreted in **Set**.

Negative/Computation Types

$$X, Y, Z ::= 1 \mid X \times Y \mid A \to X \mid \nu(X.\ Y(X))$$

Interpreted in $\mathbf{Set}^{\mathsf{T}}$, for monad T.

Type Polarity

In call-by-push-value, types are separated into two sorts:

Positive/Value Types

$$A, B, C ::= \mathsf{U}X \mid 0 \mid A + B \mid 1 \mid A \times B \mid \mu(A.\ B(A))$$

Interpreted in **Set**.

Negative/Computation Types

$$X, Y, Z ::= \mathsf{F}A \mid 1 \mid X \times Y \mid A \to X \mid \nu(X.\ Y(X))$$

Interpreted in $\mathbf{Set}^{\mathsf{T}}$, for monad T.

Semantics of Computation Types

In $\mathbf{Set}^{\mathsf{T}}$, an object X has a set $\mathsf{U}X$ and a map $\alpha_X : \mathsf{T}(\mathsf{U}X) \to \mathsf{U}X$.

Semantics of Computation Types

In $\mathbf{Set}^{\mathsf{T}}$, an object X has a set $\mathsf{U}X$ and a map $\alpha_X : \mathsf{T}(\mathsf{U}X) \to \mathsf{U}X$.

Definition (Free Algebra)

$$\begin{aligned} \mathsf{U}(\mathsf{F}A) &= \mathsf{T}A \\ \alpha_{\mathsf{F}A} &= \mathsf{TT}A \xrightarrow{\mu} \mathsf{T}A \end{aligned}$$

Semantics of Computation Types

In $\mathbf{Set}^{\mathsf{T}}$, an object X has a set $\mathsf{U}X$ and a map $\alpha_X : \mathsf{T}(\mathsf{U}X) \to \mathsf{U}X$.

Definition (Free Algebra)

$$\mathsf{U}(\mathsf{F}A) = \mathsf{T}A$$
$$\alpha_{\mathsf{F}A} = \mathsf{T}\mathsf{T}A \xrightarrow{\mu} \mathsf{T}A$$

Definition (Product Algebra)

$$\mathsf{U}(X \times Y) = \mathsf{U}X \times \mathsf{U}Y$$
$$\alpha_{X \times Y} = \mathsf{T}(\mathsf{U}X \times \mathsf{U}Y) \to \mathsf{T}(\mathsf{U}X) \times \mathsf{T}(\mathsf{U}Y) \xrightarrow{\alpha_X \times \alpha_Y} \mathsf{U}X \times \mathsf{U}Y$$

Semantics of Computation Types

In $\mathbf{Set}^{\mathsf{T}}$, an object X has a set $\mathsf{U}X$ and a map $\alpha_X : \mathsf{T}(\mathsf{U}X) \to \mathsf{U}X$.

Definition (Free Algebra)

$$\mathsf{U}(\mathsf{F}A) = \mathsf{T}A$$
$$\alpha_{\mathsf{F}A} = \mathsf{TT}A \xrightarrow{\mu} \mathsf{T}A$$

Definition (Product Algebra)

$$\mathsf{U}(X \times Y) = \mathsf{U}X \times \mathsf{U}Y$$
$$\alpha_{X \times Y} = \mathsf{T}(\mathsf{U}X \times \mathsf{U}Y) \to \mathsf{T}(\mathsf{U}X) \times \mathsf{T}(\mathsf{U}Y) \xrightarrow{\alpha_X \times \alpha_Y} \mathsf{U}X \times \mathsf{U}Y$$

Key Idea

Effects “flow over” computation types (accumulating at F types).

Cost as an Effect

In **calf** (based on CBPV), costs are annotated via an effect:

$$\frac{\Gamma \vdash e : X}{\Gamma \vdash \mathsf{step}^{c}_{X}(e) : X}$$

Cost as an Effect

In **calf** (based on CBPV), costs are annotated via an effect:

$$\frac{\Gamma \vdash e : X}{\Gamma \vdash \mathsf{step}^{c}_{X}(e) : X}$$

Here, monad $\mathsf{T} = \mathbb{N} \times (-)$.

Cost as an Effect

In **calf** (based on CBPV), costs are annotated via an effect:

$$\frac{\Gamma \vdash e : X}{\Gamma \vdash \mathsf{step}^{c}_{X}(e) : X}$$

Here, monad $\mathsf{T} = \mathbb{N} \times (-)$.

Example (Summing a List)

Cost model: 1 cost per addition.

$$\begin{aligned}
&\mathtt{sum} : \mathsf{list}(\mathbb{N}) \to \mathsf{F}(\mathbb{N})\\
&\mathtt{sum}\ [] =\\
&\mathtt{sum}\ (x :: l) =
\end{aligned}$$

Cost as an Effect

In **calf** (based on CBPV), costs are annotated via an effect:

$$\frac{\Gamma \vdash e : X}{\Gamma \vdash \mathsf{step}^{c}_{X}(e) : X}$$

Here, monad $\mathsf{T} = \mathbb{N} \times (-)$.

Example (Summing a List)

Cost model: 1 cost per addition.

$$\begin{aligned}
&\mathtt{sum} : \mathsf{list}(\mathbb{N}) \to \mathsf{F}(\mathbb{N}) \\
&\mathtt{sum}\ [] = \mathsf{ret}(0) \\
&\mathtt{sum}\ (x :: l) =
\end{aligned}$$

Cost as an Effect

In **calf** (based on CBPV), costs are annotated via an effect:

$$\frac{\Gamma \vdash e : X}{\Gamma \vdash \mathsf{step}^c_X(e) : X}$$

Here, monad $\mathsf{T} = \mathbb{N} \times (-)$.

Example (Summing a List)

Cost model: 1 cost per addition.

$$\begin{aligned}
&\mathtt{sum} : \mathsf{list}(\mathbb{N}) \to \mathsf{F}(\mathbb{N}) \\
&\mathtt{sum}\ [] = \mathsf{ret}(0) \\
&\mathtt{sum}\ (x :: l) = n \leftarrow \mathtt{sum}\ l;
\end{aligned}$$

Cost as an Effect

In **calf** (based on CBPV), costs are annotated via an effect:

$$\frac{\Gamma \vdash e : X}{\Gamma \vdash \mathsf{step}^{c}_{X}(e) : X}$$

Here, monad $\mathsf{T} = \mathbb{N} \times (-)$.

Example (Summing a List)

Cost model: 1 cost per addition.

$$\begin{aligned}
&\mathtt{sum} : \mathsf{list}(\mathbb{N}) \to \mathsf{F}(\mathbb{N}) \\
&\mathtt{sum}\ [] = \mathsf{ret}(0) \\
&\mathtt{sum}\ (x :: l) = n \leftarrow \mathtt{sum}\ l;\ \mathsf{step}^{1}(x + n)
\end{aligned}$$

Mixed Product

In **calf** (CBPV with writer monad), we have a “mixed product”:

$$A \ltimes X$$

Mixed Product

In **calf** (CBPV with writer monad), we have a "mixed product":

$$A \ltimes X$$

Definition (Mixed Product Algebra)

$$\mathsf{U}(A \ltimes X) = A \times \mathsf{U}X$$

$$\alpha_{A \ltimes X} = \mathbb{N} \times (A \times \mathsf{U}X) \cong A \times (\mathbb{N} \times \mathsf{U}X) \xrightarrow{\mathsf{id}_A \times \alpha_X} A \times \mathsf{U}X$$

Mixed Product

In **calf** (CBPV with writer monad), we have a "mixed product":

$$A \ltimes X$$

Definition (Mixed Product Algebra)

$$\mathrm{U}(A \ltimes X) = A \times \mathrm{U}X$$

$$\alpha_{A \ltimes X} = \mathbb{N} \times (A \times \mathrm{U}X) \cong A \times (\mathbb{N} \times \mathrm{U}X) \xrightarrow{\mathrm{id}_A \times \alpha_X} A \times \mathrm{U}X$$

Lemma

$$1 \ltimes X \cong X$$

Abstract Data Types, Coinductively

Abstract Data Types, Coinductively

Consider an operation signature:

$$\mathbf{op_1} \rightsquigarrow A_1$$

$$\vdots$$

$$\mathbf{op_n} \rightsquigarrow A_n$$

Abstract Data Types, Coinductively

Consider an operation signature:

$$\mathbf{op_1} \rightsquigarrow A_1$$
$$\vdots$$
$$\mathbf{op_n} \rightsquigarrow A_n$$

Work with *cofree comonad*:

$$DX \triangleq \nu(Z.\ (\mathbf{quit} : X) \times (\mathbf{op_1} : A_1 \ltimes Z) \times \cdots \times (\mathbf{op_n} : A_n \ltimes Z))$$

Abstract Data Types, Coinductively

Consider an operation signature:

$$\mathbf{op_1} \rightsquigarrow A_1$$
$$\vdots$$
$$\mathbf{op_n} \rightsquigarrow A_n$$

Work with *cofree comonad*:

$$\begin{aligned} DX &\triangleq \nu(Z.\ (\mathbf{quit} : X) \times (\mathbf{op_1} : A_1 \ltimes Z) \times \cdots \times (\mathbf{op_n} : A_n \ltimes Z)) \\ &\cong (\mathbf{quit} : X) \times (\mathbf{op_1} : A_1 \ltimes DX) \times \cdots \times (\mathbf{op_n} : A_n \ltimes DX) \end{aligned}$$

Abstract Data Types, Coinductively

Consider an operation signature:

$$\mathbf{op_1} \rightsquigarrow A_1$$

$$\vdots$$

$$\mathbf{op_n} \rightsquigarrow A_n$$

Work with *cofree comonad*:

$$DX \triangleq \nu(Z.\ (\mathbf{quit} : X) \times (\mathbf{op_1} : A_1 \ltimes Z) \times \cdots \times (\mathbf{op_n} : A_n \ltimes Z))$$
$$\cong (\mathbf{quit} : X) \times (\mathbf{op_1} : A_1 \ltimes DX) \times \cdots \times (\mathbf{op_n} : A_n \ltimes DX)$$

Here, always let $X = \mathsf{F}1 \cong (\mathbb{N},\ + : \mathbb{N} \times \mathbb{N} \to \mathbb{N})$.

$$D \cong (\mathbf{quit} : \mathsf{F}1) \times (\mathbf{op_1} : A_1 \ltimes D) \times \cdots \times (\mathbf{op_n} : A_n \ltimes D)$$

Abstract Data Types, Coinductively

Example (Queue)

$$\textbf{enqueue}[k : K] \rightsquigarrow 1$$
$$\textbf{dequeue} \rightsquigarrow K + 1$$

Abstract Data Types, Coinductively

Example (Queue)

$$\textbf{enqueue}[k : K] \rightsquigarrow 1$$
$$\textbf{dequeue} \rightsquigarrow K + 1$$

$$Q \cong (\textbf{quit} : \mathsf{F}1) \times (\textbf{enqueue} : K \to Q) \times (\textbf{dequeue} : (K + 1) \ltimes Q)$$

Abstract Data Types, Coinductively

Example (Queue)

$$\textbf{enqueue}[k : K] \rightsquigarrow 1$$
$$\textbf{dequeue} \rightsquigarrow K + 1$$

$$Q \cong (\textbf{quit} : \mathsf{F}1) \times (\textbf{enqueue} : K \to Q) \times (\textbf{dequeue} : (K + 1) \ltimes Q)$$

Example (Renting an Apartment)

$$\textbf{remain} \rightsquigarrow 1$$

Abstract Data Types, Coinductively

Example (Queue)

$$\textbf{enqueue}[k : K] \rightsquigarrow 1$$
$$\textbf{dequeue} \rightsquigarrow K + 1$$

$$Q \cong (\textbf{quit} : \mathrm{F}1) \times (\textbf{enqueue} : K \to Q) \times (\textbf{dequeue} : (K + 1) \ltimes Q)$$

Example (Renting an Apartment)

$$\textbf{remain} \rightsquigarrow 1$$

$$R \cong (\textbf{quit} : \mathrm{F}1) \times (\textbf{remain} : R)$$

Object-Oriented Programming

Remark

These coinductive types look like object-oriented programming.

Object-Oriented Programming

Remark

These coinductive types look like object-oriented programming.

$$R \cong (\mathbf{quit} : \mathsf{F}1) \times (\mathbf{remain} : R)$$

Example

Suppose $r : R$; then:

$$r.\mathbf{remain}.\mathbf{remain}.\mathbf{remain}.\mathbf{quit} : \mathsf{F}1.$$

Amortized Analysis

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. *—Tarjan*

Amortized Analysis

Renting

Payment Scheme: Daily

$$R \cong (\textbf{quit} : \mathsf{F}1) \times (\textbf{remain} : R)$$

Payment Scheme: Daily

$$R \cong (\textbf{quit} : \mathsf{F}1) \times (\textbf{remain} : R)$$

Daily Payment

$$\begin{aligned} \texttt{daily} &: R \\ \textbf{quit}(\texttt{daily}) &= \\ \textbf{remain}(\texttt{daily}) &= \end{aligned}$$

Payment Scheme: Daily

$$R \cong (\textbf{quit} : \mathsf{F}1) \times (\textbf{remain} : R)$$

Daily Payment

$$\begin{aligned} \texttt{daily} &: R \\ \textbf{quit}(\texttt{daily}) &= \mathsf{ret}(\langle\rangle) \\ \textbf{remain}(\texttt{daily}) &= \end{aligned}$$

Payment Scheme: Daily

$$R \cong (\textbf{quit} : \mathsf{F}1) \times (\textbf{remain} : R)$$

Daily Payment

$$\texttt{daily} : R$$
$$\textbf{quit}(\texttt{daily}) = \mathsf{ret}(\langle\rangle)$$
$$\textbf{remain}(\texttt{daily}) = \mathsf{step}_R^{\$20}(\texttt{daily})$$

Payment Scheme: Monthly

$$R \cong (\textbf{quit} : \mathsf{F1}) \times (\textbf{remain} : R)$$

Monthly Payment

$$\begin{aligned} \texttt{monthly} &: \mathbb{N}_{<30} \to R \\ \textbf{quit}(\texttt{monthly}\ d) &= \\ \textbf{remain}(\texttt{monthly}\ 29) &= \\ \textbf{remain}(\texttt{monthly}\ d) &= \end{aligned}$$

- d is the day of the month

Payment Scheme: Monthly

$$R \cong (\textbf{quit} : \mathsf{F1}) \times (\textbf{remain} : R)$$

Monthly Payment

$$\begin{aligned} \texttt{monthly} &: \mathbb{N}_{<30} \to R \\ \textbf{quit}(\texttt{monthly}\ d) &= \\ \textbf{remain}(\texttt{monthly}\ 29) &= \\ \textbf{remain}(\texttt{monthly}\ d) &= \end{aligned}$$

- d is the day of the month
- $\Phi(d) = \$20d$ is the money owed for the month so far

Payment Scheme: Monthly

$$R \cong (\textbf{quit} : \mathsf{F1}) \times (\textbf{remain} : R)$$

Monthly Payment

$$\begin{aligned}
\texttt{monthly} &: \mathbb{N}_{<30} \to R \\
\textbf{quit}(\texttt{monthly}\ d) &= \mathsf{step}^{\Phi(d)}_{\mathsf{F1}}(\mathsf{ret}(\langle\rangle)) \\
\textbf{remain}(\texttt{monthly}\ 29) &= \\
\textbf{remain}(\texttt{monthly}\ d) &=
\end{aligned}$$

- d is the day of the month
- $\Phi(d) = \$20d$ is the money owed for the month so far

Payment Scheme: Monthly

$$R \cong (\textbf{quit} : \mathsf{F1}) \times (\textbf{remain} : R)$$

Monthly Payment

$$\begin{aligned} \texttt{monthly} &: \mathbb{N}_{<30} \to R \\ \textbf{quit}(\texttt{monthly}\ d) &= \mathsf{step}_{\mathsf{F1}}^{\Phi(d)}(\mathsf{ret}(\langle\rangle)) \\ \textbf{remain}(\texttt{monthly}\ 29) &= \mathsf{step}_{R}^{\$600}(\texttt{monthly}\ 0) \\ \textbf{remain}(\texttt{monthly}\ d) &= \end{aligned}$$

- d is the day of the month
- $\Phi(d) = \$20d$ is the money owed for the month so far

Payment Scheme: Monthly

$$R \cong (\textbf{quit} : \mathsf{F1}) \times (\textbf{remain} : R)$$

Monthly Payment

$$\begin{aligned}
\texttt{monthly} &: \mathbb{N}_{<30} \to R \\
\textbf{quit}(\texttt{monthly}\ d) &= \mathsf{step}_{\mathsf{F1}}^{\Phi(d)}(\mathsf{ret}(\langle\rangle)) \\
\textbf{remain}(\texttt{monthly}\ 29) &= \mathsf{step}_{R}^{\$600}(\texttt{monthly}\ 0) \\
\textbf{remain}(\texttt{monthly}\ d) &= \texttt{monthly}\ (d+1)
\end{aligned}$$

- d is the day of the month
- $\Phi(d) = \$20d$ is the money owed for the month so far

Coinductive Equivalence

Theorem

For all days of the month d, $\texttt{monthly}\ d = \texttt{step}_R^{\Phi(d)}(\texttt{daily})$.

Coinductive Equivalence

Theorem

For all days of the month d, $\texttt{monthly}\ d = \texttt{step}_{R}^{\Phi(d)}(\texttt{daily})$.

Proof.

By coinduction:

Coinductive Equivalence

Theorem

For all days of the month d, $\texttt{monthly}\ d = \texttt{step}_{R}^{\Phi(d)}(\texttt{daily})$.

Proof.

By coinduction:

- In the **quit** case, both incur the same number of steps.

Coinductive Equivalence

Theorem

For all days of the month d, $\texttt{monthly}\ d = \mathsf{step}_R^{\Phi(d)}(\texttt{daily})$.

Proof.

By coinduction:

- In the **quit** case, both incur the same number of steps.
- In the **remain** case:

Coinductive Equivalence

Theorem

For all days of the month d, $\texttt{monthly}\ d = \mathrm{step}_R^{\Phi(d)}(\texttt{daily})$.

Proof.

By coinduction:

- In the **quit** case, both incur the same number of steps.
- In the **remain** case:
  - If $d = 29$, both incur \$600; peel off and use co-IH.

Coinductive Equivalence

Theorem

For all days of the month d, $\texttt{monthly}\ d = \mathrm{step}_{R}^{\Phi(d)}(\texttt{daily})$.

Proof.

By coinduction:

- In the **quit** case, both incur the same number of steps.
- In the **remain** case:
  - If $d = 29$, both incur \$600; peel off and use co-IH.
  - Otherwise, push cost forward and use co-IH.

Coinductive Equivalence

Theorem

For all days of the month d, $\texttt{monthly}\ d = \text{step}_R^{\Phi(d)}(\texttt{daily})$.

Proof.

By coinduction:

- In the **quit** case, both incur the same number of steps.
- In the **remain** case:
  - If $d = 29$, both incur \$600; peel off and use co-IH.
  - Otherwise, push cost forward and use co-IH.

Essential: pushing cost over computation types. □

Amortizing Full Stays

Amortizing Full Stays

Definition (Full-Stay Evaluation)

$$
\begin{aligned}
&\texttt{eval} : \mathbb{N} \to \mathsf{U}R \to \mathsf{F}1 \\
&\texttt{eval}\ 0 \qquad\quad r = \textbf{quit}(r) \\
&\texttt{eval}\ (n+1)\ r = \texttt{eval}\ n\ (\textbf{remain}\ r)
\end{aligned}
$$

Amortizing Full Stays

Definition (Full-Stay Evaluation)

$$\begin{aligned}
&\texttt{eval} : \mathbb{N} \to \mathsf{U}R \to \mathsf{F}1\\
&\texttt{eval}\ 0 \qquad\qquad r = \textbf{quit}(r)\\
&\texttt{eval}\ (n+1)\ r = \texttt{eval}\ n\ (\textbf{remain}\ r)
\end{aligned}$$

Definition (Full-Stay Evaluation Equivalence)

Say $r_1 \approx r_2$ iff for all n,

$$\texttt{eval}\ n\ r_1 = \texttt{eval}\ n\ r_2.$$

Amortizing Full Stays

Definition (Full-Stay Evaluation)

$$\begin{aligned}
&\texttt{eval} : \mathbb{N} \to \mathsf{U}R \to \mathsf{F1} \\
&\texttt{eval}\ 0\ \qquad r = \textbf{quit}(r) \\
&\texttt{eval}\ (n+1)\ r = \texttt{eval}\ n\ (\textbf{remain}\ r)
\end{aligned}$$

Definition (Full-Stay Evaluation Equivalence)

Say $r_1 \approx r_2$ iff for all n,

$$\texttt{eval}\ n\ r_1 = \texttt{eval}\ n\ r_2.$$

Theorem

For all r_1 and r_2, $r_1 = r_2$ iff $r_1 \approx r_2$.

Amortizing Full Stays

Definition (Full-Stay Evaluation)

$$\begin{aligned}
&\texttt{eval} : \mathbb{N} \to \mathsf{U}R \to \mathsf{F}1\\
&\texttt{eval}\ 0 \qquad\quad r = \textbf{quit}(r)\\
&\texttt{eval}\ (n+1)\ r = \texttt{eval}\ n\ (\textbf{remain}\ r)
\end{aligned}$$

Definition (Full-Stay Evaluation Equivalence)

Say $r_1 \approx r_2$ iff for all n,

$$\texttt{eval}\ n\ r_1 = \texttt{eval}\ n\ r_2.$$

Theorem

For all r_1 and r_2, $r_1 = r_2$ iff $r_1 \approx r_2$.

Proof.

By ($\Rightarrow$) induction on n and ($\Leftarrow$) coinduction on $r_1 = r_2$. $\square$

Amortized Analysis

Queue

Queue Implementation: Specification

$$Q \cong (\mathbf{quit} : \mathsf{F}1) \times (\mathbf{enqueue} : K \to Q) \times (\mathbf{dequeue} : (K + 1) \ltimes Q)$$

Queue Implementation: Specification

$$Q \cong (\textbf{quit} : \mathsf{F}1) \times (\textbf{enqueue} : K \to Q) \times (\textbf{dequeue} : (K + 1) \ltimes Q)$$

Specification

$$\begin{aligned} \mathtt{spec} &: \mathsf{list}(K) \to Q \\ \textbf{quit}(\mathtt{spec}\ l) &= \\ \textbf{enqueue}(\mathtt{spec}\ l) &= \\ \textbf{dequeue}(\mathtt{spec}\ []) &= \\ \textbf{dequeue}(\mathtt{spec}\ (k :: l)) &= \end{aligned}$$

Queue Implementation: Specification

$$Q \cong (\mathbf{quit} : \mathsf{F}1) \times (\mathbf{enqueue} : K \to Q) \times (\mathbf{dequeue} : (K + 1) \ltimes Q)$$

Specification

$$\begin{aligned} \texttt{spec} &: \mathsf{list}(K) \to Q \\ \mathbf{quit}(\texttt{spec}\ l) &= \mathsf{ret}(\langle\rangle) \\ \mathbf{enqueue}(\texttt{spec}\ l) &= \\ \mathbf{dequeue}(\texttt{spec}\ []) &= \\ \mathbf{dequeue}(\texttt{spec}\ (k :: l)) &= \end{aligned}$$

Queue Implementation: Specification

$$Q \cong (\mathbf{quit} : \mathsf{F}1) \times (\mathbf{enqueue} : K \to Q) \times (\mathbf{dequeue} : (K + 1) \ltimes Q)$$

Specification

$$\begin{aligned}
\mathtt{spec} &: \mathsf{list}(K) \to Q \\
\mathbf{quit}(\mathtt{spec}\ l) &= \mathsf{ret}(\langle\rangle) \\
\mathbf{enqueue}(\mathtt{spec}\ l) &= \lambda k.\ \mathsf{step}^1_Q(\mathtt{spec}\ (l \mathbin{+\!\!+} [k])) \\
\mathbf{dequeue}(\mathtt{spec}\ []) &= \\
\mathbf{dequeue}(\mathtt{spec}\ (k :: l)) &=
\end{aligned}$$

Queue Implementation: Specification

$$Q \cong (\textbf{quit} : \mathsf{F}1) \times (\textbf{enqueue} : K \to Q) \times (\textbf{dequeue} : (K + 1) \ltimes Q)$$

Specification

$$\begin{aligned}
\mathsf{spec} &: \mathsf{list}(K) \to Q \\
\textbf{quit}(\mathsf{spec}\ l) &= \mathsf{ret}(\langle\rangle) \\
\textbf{enqueue}(\mathsf{spec}\ l) &= \lambda k.\ \mathsf{step}_Q^1(\mathsf{spec}\ (l +\!\!+ [k])) \\
\textbf{dequeue}(\mathsf{spec}\ []) &= \langle \mathsf{none}, \mathsf{spec}\ [] \rangle \\
\textbf{dequeue}(\mathsf{spec}\ (k :: l)) &=
\end{aligned}$$

Queue Implementation: Specification

$$Q \cong (\textbf{quit} : \mathsf{F}1) \times (\textbf{enqueue} : K \to Q) \times (\textbf{dequeue} : (K + 1) \ltimes Q)$$

Specification

$$\begin{aligned}
\mathsf{spec} &: \mathsf{list}(K) \to Q \\
\textbf{quit}(\mathsf{spec}\ l) &= \mathsf{ret}(\langle\rangle) \\
\textbf{enqueue}(\mathsf{spec}\ l) &= \lambda k.\ \mathsf{step}^1_Q(\mathsf{spec}\ (l +\!\!+ [k])) \\
\textbf{dequeue}(\mathsf{spec}\ []) &= \langle \mathsf{none}, \mathsf{spec}\ [] \rangle \\
\textbf{dequeue}(\mathsf{spec}\ (k :: l)) &= \langle \mathsf{some}(k), \mathsf{spec}\ l \rangle
\end{aligned}$$

Queue Implementation: Batched (Amortized)

Batched Queue

$$\texttt{batched} : \mathsf{list}(K) \to \mathsf{list}(K) \to Q$$

$$\textbf{quit}(\texttt{batched}\ bl\ \ fl) =$$

$$\textbf{enqueue}(\texttt{batched}\ bl\ \ fl) =$$

$$\textbf{dequeue}(\texttt{batched}\ bl\ [\,]) =$$

$$\textbf{dequeue}(\texttt{batched}\ bl\ (k :: fl)) =$$

Here, $\Phi(bl, fl) = |bl|$ (how much spec has already paid).

Queue Implementation: Batched (Amortized)

Batched Queue

$$\texttt{batched} : \mathsf{list}(K) \to \mathsf{list}(K) \to Q$$

$$\textbf{quit}(\texttt{batched}\ bl\ \ fl) = \mathsf{step}^{\Phi(bl,fl)}_{\mathsf{F1}}(\mathsf{ret}(\langle\rangle))$$

$$\textbf{enqueue}(\texttt{batched}\ bl\ \ fl) =$$

$$\textbf{dequeue}(\texttt{batched}\ bl\ [\,]) =$$

$$\textbf{dequeue}(\texttt{batched}\ bl\ (k :: fl)) =$$

Here, $\Phi(bl, fl) = |bl|$ (how much spec has already paid).

Queue Implementation: Batched (Amortized)

Batched Queue

$$\texttt{batched} : \mathsf{list}(K) \to \mathsf{list}(K) \to Q$$

$$\textbf{quit}(\texttt{batched}\ bl\ \ fl) = \mathsf{step}^{\Phi(bl,fl)}_{\mathsf{F1}}(\mathsf{ret}(\langle\rangle))$$

$$\textbf{enqueue}(\texttt{batched}\ bl\ \ fl) = \lambda k.\ \texttt{batched}\ (k :: bl)\ fl$$

$$\textbf{dequeue}(\texttt{batched}\ bl\ []) =$$

$$\textbf{dequeue}(\texttt{batched}\ bl\ (k :: fl)) =$$

Here, $\Phi(bl, fl) = |bl|$ (how much spec has already paid).

Queue Implementation: Batched (Amortized)

Batched Queue

$$\texttt{batched} : \mathsf{list}(K) \to \mathsf{list}(K) \to Q$$

$$\mathbf{quit}(\texttt{batched}\ bl\ \ fl) = \mathsf{step}^{\Phi(bl,fl)}_{\mathsf{F1}}(\mathsf{ret}(\langle\rangle))$$

$$\mathbf{enqueue}(\texttt{batched}\ bl\ \ fl) = \lambda k.\ \texttt{batched}\ (k :: bl)\ fl$$

$$\mathbf{dequeue}(\texttt{batched}\ bl\ []) = \mathsf{step}^{|bl|}(-)$$

$$\begin{cases} \langle \mathsf{none}, \texttt{batched}\ []\ []\rangle & \texttt{rev}\ bl = [] \\ \langle \mathsf{some}(k), \texttt{batched}\ []\ fl\rangle & \texttt{rev}\ bl = k :: fl \end{cases}$$

$$\mathbf{dequeue}(\texttt{batched}\ bl\ (k :: fl)) =$$

Here, $\Phi(bl, fl) = |bl|$ (how much spec has already paid).

Queue Implementation: Batched (Amortized)

Batched Queue

$$\texttt{batched} : \mathsf{list}(K) \to \mathsf{list}(K) \to Q$$

$$\textbf{quit}(\texttt{batched}\ bl\ fl) = \mathsf{step}_{\mathsf{F1}}^{\Phi(bl,fl)}(\mathsf{ret}(\langle\rangle))$$

$$\textbf{enqueue}(\texttt{batched}\ bl\ fl) = \lambda k.\ \texttt{batched}\ (k :: bl)\ fl$$

$$\textbf{dequeue}(\texttt{batched}\ bl\ []) = \mathsf{step}^{|bl|}(-)$$

$$\begin{cases} \langle \mathsf{none}, \texttt{batched}\ []\ [] \rangle & \texttt{rev}\ bl = [] \\ \langle \mathsf{some}(k), \texttt{batched}\ []\ fl \rangle & \texttt{rev}\ bl = k :: fl \end{cases}$$

$$\textbf{dequeue}(\texttt{batched}\ bl\ (k :: fl)) = \langle \mathsf{some}(k), \texttt{batched}\ bl\ fl \rangle$$

Here, $\Phi(bl, fl) = |bl|$ (how much spec has already paid).

Coinductive Amortized Analysis

Theorem

For all $bl, fl : \mathsf{list}(K)$,

$$\texttt{batched}\ bl\ fl = \mathsf{step}_{Q}^{\Phi(bl,fl)}(\texttt{spec}\ (fl +\!\!+ \texttt{rev}\ bl)).$$

Coinductive Amortized Analysis

Theorem

For all bl, fl : $\mathsf{list}(K)$,

$$\texttt{batched}\ \mathit{bl}\ \mathit{fl} = \mathsf{step}_{Q}^{\Phi(\mathit{bl},\mathit{fl})}(\texttt{spec}\ (\mathit{fl} +\!\!+\ \texttt{rev}\ \mathit{bl})).$$

Proof.

By coinduction. □

Amortizing Finite Sequences of Operations

Amortizing Finite Sequences of Operations

Definition (Sequence of Operations, Free Monad)

$$P(A) \cong (\mathbf{ret} : A) + (\mathbf{enq} : K \times P(A)) + (\mathbf{deq} : \mathsf{U}(K + 1 \to \mathsf{F}(P(A))))$$

Amortizing Finite Sequences of Operations

Definition (Sequence of Operations, Free Monad)

$$P(A) \cong (\mathbf{ret} : A) + (\mathbf{enq} : K \times P(A)) + (\mathbf{deq} : \mathrm{U}(K + 1 \to \mathrm{F}(P(A))))$$

Definition (Sequence Evaluation)

$$\texttt{eval} : P(A) \to \mathrm{U}Q \to A \ltimes \mathrm{F}1$$

By induction on the operation sequence $P(A)$.

Amortizing Finite Sequences of Operations

Definition (Sequence of Operations, Free Monad)

$$P(A) \cong (\textbf{ret} : A) + (\textbf{enq} : K \times P(A)) + (\textbf{deq} : \mathrm{U}(K + 1 \to \mathrm{F}(P(A))))$$

Definition (Sequence Evaluation)

$$\texttt{eval} : P(A) \to \mathrm{U}Q \to A \ltimes \mathrm{F}1$$

By induction on the operation sequence $P(A)$.

Definition (Classic Amortized Equivalence)

Say $q_1 \approx q_2$ iff for all A and $p : P(A)$,

$$\texttt{eval}\ p\ q_1 = \texttt{eval}\ p\ q_2.$$

Amortizing Finite Sequences of Operations

Definition (Sequence of Operations, Free Monad)

$$P(A) \cong (\mathbf{ret} : A) + (\mathbf{enq} : K \times P(A)) + (\mathbf{deq} : \mathsf{U}(K + 1 \to \mathsf{F}(P(A))))$$

Definition (Sequence Evaluation)

$$\texttt{eval} : P(A) \to \mathsf{U}Q \to A \ltimes \mathsf{F}1$$

By induction on the operation sequence $P(A)$.

Definition (Classic Amortized Equivalence)

Say $q_1 \approx q_2$ iff for all A and $p : P(A)$,

$$\texttt{eval}\ p\ q_1 = \texttt{eval}\ p\ q_2.$$

Theorem (Coinductive vs. Classic Amortized Analysis)

For all q_1 and q_2, $q_1 = q_2$ iff $q_1 \approx q_2$.

Conclusion

Summary

1. In call-by-push-value, effects propagate through computation types, including the mixed product in **calf**.

Summary

1. In call-by-push-value, effects propagate through computation types, including the mixed product in **calf**.
2. Sequential-use data structures are coinductive/object-oriented "machines".

Summary

1. In call-by-push-value, effects propagate through computation types, including the mixed product in **calf**.
2. Sequential-use data structures are coinductive/object-oriented "machines".
3. Coinductive equivalence pushes cost forward, capturing amortized analysis.

Summary

1. In call-by-push-value, effects propagate through computation types, including the mixed product in **calf**.
2. Sequential-use data structures are coinductive/object-oriented "machines".
3. Coinductive equivalence pushes cost forward, capturing amortized analysis.
4. This coincides with the traditional sequence-of-operations description of amortized analysis!

Summary

1. In call-by-push-value, effects propagate through computation types, including the mixed product in **calf**.
2. Sequential-use data structures are coinductive/object-oriented "machines".
3. Coinductive equivalence pushes cost forward, capturing amortized analysis.
4. This coincides with the traditional sequence-of-operations description of amortized analysis!
5. Results are formalized in **calf**/Agda (renting, batched queues, and dynamically-resizing arrays).

Bonus

Coinductive Equivalence

Theorem

For all d, $\texttt{monthly}\ d = \texttt{step}^{\Phi(d)}(\texttt{daily})$.

Coinductive Equivalence

Theorem

For all d, $\texttt{monthly}\ d = \mathsf{step}^{\Phi(d)}(\texttt{daily})$.

Proof.

We prove by coinduction, showing:

1. $\textbf{quit}(\texttt{monthly}\ d) = \textbf{quit}(\mathsf{step}^{\Phi(d)}(\texttt{daily}))$
2. $\textbf{remain}(\texttt{monthly}\ d) = \textbf{remain}(\mathsf{step}^{\Phi(d)}(\texttt{daily}))$

Coinductive Equivalence

Theorem

For all d, $\texttt{monthly}\ d = \mathsf{step}^{\Phi(d)}(\texttt{daily})$.

Proof.

$$\textbf{quit}(\texttt{daily}) = \mathsf{ret}(\langle\rangle)$$
$$\textbf{quit}(\texttt{monthly}\ d) = \mathsf{step}_{\mathsf{F1}}^{\Phi(d)}(\mathsf{ret}(\langle\rangle))$$

We show:

$$\begin{aligned}\textbf{quit}(\texttt{monthly}\ d) &= \mathsf{step}^{\Phi(d)}(\mathsf{ret}(\langle\rangle)) \\ &= \mathsf{step}^{\Phi(d)}(\textbf{quit}(\texttt{daily})) \\ &= \textbf{quit}(\mathsf{step}^{\Phi(d)}(\texttt{daily}))\end{aligned}$$

Coinductive Equivalence

Theorem

For all d, $\texttt{monthly}\ d = \mathsf{step}^{\Phi(d)}(\texttt{daily})$.

Proof.

$$\textbf{remain}(\texttt{daily}) = \mathsf{step}_R^{\$20}(\texttt{daily})$$
$$\textbf{remain}(\texttt{monthly}\ 29) = \mathsf{step}_R^{\$600}(\texttt{monthly}\ 0)$$

We show:

$$\begin{aligned}
\textbf{remain}(\texttt{monthly}\ 29) &= \mathsf{step}^{\$600}(\texttt{monthly}\ 0) \\
&= \mathsf{step}^{\$600}(\texttt{daily}) && \text{(co-IH)} \\
&= \mathsf{step}^{\Phi(29)}(\mathsf{step}^{\$20}(\texttt{daily})) \\
&= \mathsf{step}^{\Phi(29)}(\textbf{remain}(\texttt{daily})) \\
&= \textbf{remain}(\mathsf{step}^{\Phi(29)}(\texttt{daily}))
\end{aligned}$$

Coinductive Equivalence

Theorem

For all d, $\texttt{monthly}\ d = \mathsf{step}^{\Phi(d)}(\texttt{daily})$.

Proof.

$$\mathbf{remain}(\texttt{daily}) = \mathsf{step}_R^{\$20}(\texttt{daily})$$
$$\mathbf{remain}(\texttt{monthly}\ d) = \texttt{monthly}\ (d+1)$$

We show:

$$\begin{aligned}
\mathbf{remain}(\texttt{monthly}\ d) &= \texttt{monthly}\ (d+1) \\
&= \mathsf{step}^{\Phi(d+1)}(\texttt{daily}) && \text{(co-IH)} \\
&= \mathsf{step}^{\Phi(d)}(\mathsf{step}^{\$20}(\texttt{daily})) \\
&= \mathsf{step}^{\Phi(d)}(\mathbf{remain}(\texttt{daily})) \\
&= \mathbf{remain}(\mathsf{step}^{\Phi(d)}(\texttt{daily}))
\end{aligned}$$

□

References

References i

A. Balan and A. Kurz.
On Coalgebras over Algebras.
Electronic Notes in Theoretical Computer Science, 264(2):47–62, Aug. 2010.

W. 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.

W. 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, Oct. 2009. Association for Computing Machinery.

B. Jacobs.
Mongruences and cofree coalgebras.
In V. S. Alagar and M. Nivat, editors, *Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science, pages 245–260, Berlin, Heidelberg, 1995. Springer.

B. Jacobs.
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.

P. B. Levy.
Call-By-Push-Value.
PhD thesis, University of London, 2001.

Y. Niu, J. Sterling, H. Grodin, and R. Harper.
A cost-aware logical framework.
Proceedings of the ACM on Programming Languages, 6(POPL):9:1–9:31, Jan. 2022.

References iv

J. Power and O. Shkaravska.
From Comodels to Coalgebras: State and Arrays.
Electronic Notes in Theoretical Computer Science, 106:297–314, Dec. 2004.

R. E. Tarjan.
Amortized Computational Complexity.
SIAM Journal on Algebraic Discrete Methods, 6(2):306–318, Apr. 1985.