Amortized Analysis via Coinduction

Harrison Grodin, j.w.w. Robert Harper June 19, 2023

Carnegie Mellon University

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.

Goal

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

- 1. Call-By-Push-Value and calf
- 2. Abstract Data Types, Coinductively
- 3. Amortized Analysis

Renting

Queue

4. Conclusion

Call-By-Push-Value and calf

Types
0 <i>A</i> + <i>B</i>
1 $A \times B$
$\mu(A. B(A))$

Positive/Value Types
<i>A</i> , <i>B</i> , <i>C</i> ::=
0 A+B
$1 A \times B$
$\mu(A. \ B(A))$

Interpreted in Set.

Positive/Value Types	Negative/Computation Types
A, B, C ::=	X, Y, Z ::=
0 A+B	1 $X \times Y$
1 A imes B	$\mathcal{A} ightarrow \mathcal{X}$
$\mu(A. B(A))$	$\nu(X. Y(X))$

Interpreted in Set.

Positive/Value Types	Negative/Computation Types
A, B, C ::=	<i>X</i> , <i>Y</i> , <i>Z</i> ::=
0 A+B	1 X imes Y
$1 A \times B$	$oldsymbol{\mathcal{A}} o X$
$\mu(A. B(A))$	u(X, Y(X))
Interpreted in Set .	Interpreted in Set ^T , for monad T.

Positive/Value Types	Negative/Computation Types
A, B, C ::= U X	X, Y, Z ::=
0 A+B	$1 X \times Y$
$1 A \times B$	${oldsymbol{\mathcal{A}}} ightarrow X$
$\mu(A. B(A))$	$\nu(X. Y(X))$
Interpreted in Set .	Interpreted in Set ^T , for monad T .

Positive/Value Types	Negative/Computation Types
A, B, C ::= U X	X, Y, Z ::= F A
0 <i>A</i> + <i>B</i>	$1 X \times Y$
$1 A \times B$	${oldsymbol{\mathcal{A}}} o {oldsymbol{X}}$
$\mu(A. B(A))$	$\nu(X. Y(X))$
Interpreted in Set .	Interpreted in Set ^T , for monad T .

In **Set**^T, an object X has a set UX and a map $\alpha_X : T(UX) \to UX$.

In **Set**^T, an object X has a set UX and a map $\alpha_X : T(UX) \to UX$.

Definition (Free Algebra)

U(FA) = TA $\alpha_{FA} = TTA \xrightarrow{\mu} TA$

In **Set**^T, an object X has a set UX and a map $\alpha_X : T(UX) \to UX$.

Definition (Free Algebra)

U(FA) = TA $\alpha_{FA} = TTA \xrightarrow{\mu} TA$

Definition (Product Algebra)

 $U(X \times Y) = UX \times UY$ $\alpha_{X \times Y} = T(UX \times UY) \rightarrow T(UX) \times T(UY) \xrightarrow{\alpha_X \times \alpha_Y} UX \times UY$

In **Set**^T, an object X has a set UX and a map $\alpha_X : T(UX) \to UX$.

Definition (Free Algebra)

U(FA) = TA $\alpha_{FA} = TTA \xrightarrow{\mu} TA$

Definition (Product Algebra)

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

Key Idea

Effects "flow over" computation types (accumulating at F types).

 $\frac{\Gamma \vdash e : X}{\Gamma \vdash \operatorname{step}_X^c(e) : X}$

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

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

 $\frac{\Gamma \vdash e : X}{\Gamma \vdash \operatorname{step}_X^c(e) : X}$

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

Example (Summing a List)

Cost model: 1 cost per addition.

```
sum : list(\mathbb{N}) \rightarrow F(\mathbb{N})
sum [] =
sum (x :: l) =
```

 $\frac{\Gamma \vdash e : X}{\Gamma \vdash \operatorname{step}_X^c(e) : X}$

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

Example (Summing a List)

Cost model: 1 cost per addition.

```
sum : list(\mathbb{N}) \to F(\mathbb{N})
sum [] = ret(0)
sum (x :: l) =
```

 $\frac{\Gamma \vdash e : X}{\Gamma \vdash \operatorname{step}_X^c(e) : X}$

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

Example (Summing a List)

Cost model: 1 cost per addition.

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

 $\frac{\Gamma \vdash e : X}{\Gamma \vdash \operatorname{step}_X^c(e) : X}$

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

Example (Summing a List)

Cost model: 1 cost per addition.

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

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

 $A \ltimes X$

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

 $A \ltimes X$

Definition (Mixed Product Algebra)

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

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

 $A \ltimes X$

Definition (Mixed Product Algebra)

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

Lemma

$1 \ltimes X \cong X$

Consider an operation signature:

 $op_1 \rightsquigarrow A_1$: $op_n \rightsquigarrow A_n$

Consider an operation signature:

 $op_1 \rightsquigarrow A_1$ \vdots $op_n \rightsquigarrow A_n$

Work with cofree comonad:

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

Consider an operation signature:

 $op_1 \rightsquigarrow A_1$ \vdots $op_n \rightsquigarrow A_n$

Work with cofree comonad:

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

Consider an operation signature:

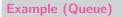
 $op_1 \rightsquigarrow A_1$ \vdots $op_n \rightsquigarrow A_n$

Work with cofree comonad:

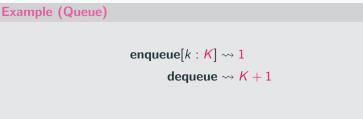
 $DX \triangleq \nu(Z. (quit: X) \times (op_1 : A_1 \ltimes Z) \times \cdots \times (op_n : A_n \ltimes Z))$ $\cong (quit: X) \times (op_1 : A_1 \ltimes DX) \times \cdots \times (op_n : A_n \ltimes DX)$

Here, always let $X = F1 \cong (\mathbb{N}, + : \mathbb{N} \times \mathbb{N} \to \mathbb{N}).$

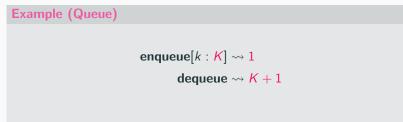
 $D \cong (quit: F1) \times (op_1 : A_1 \ltimes D) \times \cdots \times (op_n : A_n \ltimes D)$



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



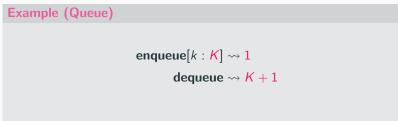
 $Q \cong (quit:F1) \times (enqueue: K \to Q) \times (dequeue: (K + 1) \ltimes Q)$



 $Q \cong (quit : F1) \times (enqueue : K \to Q) \times (dequeue : (K + 1) \ltimes Q)$

Example (Renting an Apartment)

remain $\rightsquigarrow 1$



 $Q \cong (quit : F1) \times (enqueue : K \to Q) \times (dequeue : (K + 1) \ltimes Q)$

Example (Renting an Apartment)

 $remain \rightsquigarrow 1$

 $R \cong (quit : F1) \times (remain : R)$

Remark

These coinductive types look like object-oriented programming.

Remark

These coinductive types look like object-oriented programming.

 $R \cong (quit : F1) \times (remain : R)$

Example

Suppose r : R; then:

r.remain.remain.quit : F1.

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

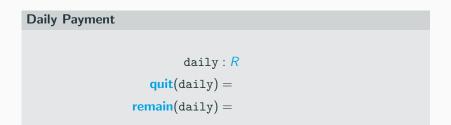
Amortized Analysis

Renting

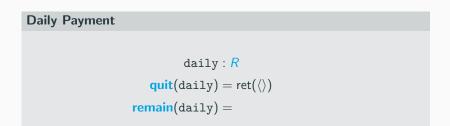
Payment Scheme: Daily

 $R \cong (quit : F1) \times (remain : R)$

 $R \cong (quit : F1) \times (remain : R)$



 $R \cong (quit : F1) \times (remain : R)$

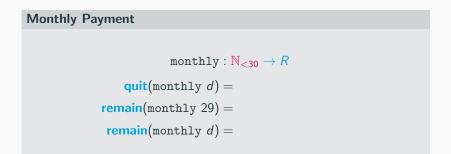


 $R \cong (quit : F1) \times (remain : R)$

Daily Payment

daily: R $quit(daily) = ret(\langle \rangle)$ $remain(daily) = step_R^{20}(daily)$

 $R \cong (quit : F1) \times (remain : R)$



• *d* is the day of the month

 $R \cong (quit : F1) \times (remain : R)$

Monthly Payment monthly: $\mathbb{N}_{<30} \rightarrow R$ quit(monthly d) = remain(monthly 29) = remain(monthly d) =

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

 $R \cong (quit : F1) \times (remain : R)$

Monthly Payment

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

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

 $R \cong (quit : F1) \times (remain : R)$

Monthly Payment

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

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

 $R \cong (quit : F1) \times (remain : R)$

Monthly Payment

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

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

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

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

Proof.

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

Proof.

By coinduction:

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

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

Proof.

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

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

Proof.

- 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.

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

Proof.

- 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.

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

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.

Definition (Full-Stay Evaluation)

eval: $\mathbb{N} \to \mathbb{U}R \to F1$ eval 0 r = quit(r)eval (n+1) r = eval n (remain r)

Definition (Full-Stay Evaluation)

eval: $\mathbb{N} \to \mathbb{U}R \to F1$ eval 0 r = quit(r)eval (n+1) r = eval n (remain r)

Definition (Full-Stay Evaluation Equivalence)

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

eval $n r_1 = eval n r_2$.

Definition (Full-Stay Evaluation)

eval: $\mathbb{N} \to \mathbb{U}R \to F1$ eval 0 r = quit(r)eval (n+1) r = eval n (remain r)

Definition (Full-Stay Evaluation Equivalence)

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

eval $n r_1 = eval n r_2$.

Theorem

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

Definition (Full-Stay Evaluation)

eval: $\mathbb{N} \to \mathbb{U}R \to F1$ eval 0 r = quit(r)eval (n+1) r = eval n (remain r)

Definition (Full-Stay Evaluation Equivalence)

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

eval $n r_1 = eval n r_2$.

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 $r_1 = r_2$.

Amortized Analysis

Queue

 $Q \cong (quit: F1) \times (enqueue: K \to Q) \times (dequeue: (K + 1) \ltimes Q)$

 $Q \cong (\textbf{quit}: F1) \times (\textbf{enqueue}: \mathcal{K} \rightarrow Q) \times (\textbf{dequeue}: (\mathcal{K} + 1) \ltimes Q)$

Specification spec : $list(K) \rightarrow Q$ quit(spec l) = enqueue(spec l) = dequeue(spec []) =dequeue(spec (k :: l)) = $Q \cong (\textbf{quit}: F1) \times (\textbf{enqueue}: \mathcal{K} \rightarrow Q) \times (\textbf{dequeue}: (\mathcal{K} + 1) \ltimes Q)$

Specification

 $spec : list(K) \rightarrow Q$ $quit(spec l) = ret(\langle \rangle)$ enqueue(spec l) = dequeue(spec []) =dequeue(spec (k :: l)) = $Q \cong (\textbf{quit}: F1) \times (\textbf{enqueue}: \mathcal{K} \rightarrow Q) \times (\textbf{dequeue}: (\mathcal{K} + 1) \ltimes Q)$

Specification

 $spec : list(\mathcal{K}) \to Q$ $quit(spec \ l) = ret(\langle \rangle)$ $enqueue(spec \ l) = \lambda k. \ step_Q^1(spec \ (l + [k]))$ $dequeue(spec \ []) =$ $dequeue(spec \ (k :: l)) =$

 $Q \cong (\textbf{quit}: F1) \times (\textbf{enqueue}: \textit{K} \rightarrow \textit{Q}) \times (\textbf{dequeue}: (\textit{K} + 1) \ltimes \textit{Q})$

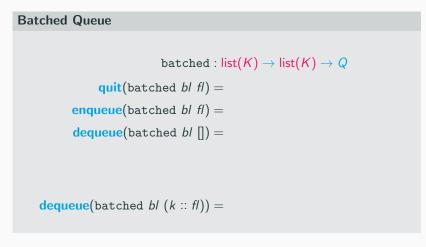
Specification

 $spec : list(\mathcal{K}) \to Q$ $quit(spec \ l) = ret(\langle \rangle)$ $enqueue(spec \ l) = \lambda k. \ step_Q^1(spec \ (l + [k]))$ $dequeue(spec \ []) = \langle none, spec \ [] \rangle$ $dequeue(spec \ (k :: l)) =$

 $Q \cong (\textbf{quit}: F1) \times (\textbf{enqueue}: \textit{K} \rightarrow \textit{Q}) \times (\textbf{dequeue}: (\textit{K} + 1) \ltimes \textit{Q})$

Specification

spec : $list(\mathcal{K}) \rightarrow Q$ quit(spec l) = ret($\langle \rangle$) enqueue(spec l) = λk . step¹_Q(spec (l + [k])) dequeue(spec []) = $\langle none, spec [] \rangle$ dequeue(spec (k :: l)) = $\langle some(k), spec l \rangle$



Batched Queue

 $\begin{array}{l} \texttt{batched} : \texttt{list}(\mathcal{K}) \to \texttt{list}(\mathcal{K}) \to Q\\ \texttt{quit}(\texttt{batched} \ bl \ fl) = \texttt{step}_{\texttt{F1}}^{\Phi(bl, fl)}(\texttt{ret}(\langle \rangle))\\ \texttt{enqueue}(\texttt{batched} \ bl \ fl) =\\ \texttt{dequeue}(\texttt{batched} \ bl \ []) = \end{array}$

dequeue(batched bl(k :: fl)) =

Batched Queue

 $\begin{array}{l} \texttt{batched}: \texttt{list}(\mathcal{K}) \to \texttt{list}(\mathcal{K}) \to \mathcal{Q}\\ \texttt{quit}(\texttt{batched} \ \textit{bl} \ \textit{fl}) = \texttt{step}_{\texttt{F1}}^{\Phi(\textit{bl},\textit{fl})}(\texttt{ret}(\langle\rangle))\\ \texttt{enqueue}(\texttt{batched} \ \textit{bl} \ \textit{fl}) = \lambda\textit{k}. \ \texttt{batched} \ (\textit{k} :: \textit{bl}) \ \textit{fl}\\ \texttt{dequeue}(\texttt{batched} \ \textit{bl} \ \textit{fl}) = \end{array}$

dequeue(batched bl(k :: fl)) =

Batched Queue

 $\begin{array}{l} \texttt{batched}:\texttt{list}(\mathcal{K}) \to \texttt{list}(\mathcal{K}) \to Q\\ \texttt{quit}(\texttt{batched} \ \textit{bl} \ \textit{fl}) = \texttt{step}_{\texttt{F1}}^{\Phi(\textit{bl},\textit{fl})}(\texttt{ret}(\langle\rangle))\\ \texttt{enqueue}(\texttt{batched} \ \textit{bl} \ \textit{fl}) = \lambda k. \ \texttt{batched} \ (k :: \textit{bl}) \ \textit{fl}\\ \texttt{dequeue}(\texttt{batched} \ \textit{bl} \ \textit{fl}) = \texttt{step}^{|\textit{bl}|}(-)\\ \begin{cases} \langle \texttt{none},\texttt{batched} \ \textit{ll} \ \textit{ll} \rangle & \texttt{rev} \ \textit{bl} = \textit{ll} \\ \langle \texttt{some}(k),\texttt{batched} \ \textit{ll} \ \textit{fl} \rangle & \texttt{rev} \ \textit{bl} = k :: \textit{fl} \end{cases} \end{cases} \\ \texttt{dequeue}(\texttt{batched} \ \textit{bl} \ (k :: \textit{fl})) = \end{array}$

Batched Queue

 $\begin{array}{l} \texttt{batched} : \texttt{list}(\mathcal{K}) \to \texttt{list}(\mathcal{K}) \to Q\\ \texttt{quit}(\texttt{batched} \ bl \ fl) = \texttt{step}_{\texttt{F1}}^{\Phi(bl,fl)}(\texttt{ret}(\langle\rangle))\\ \texttt{enqueue}(\texttt{batched} \ bl \ fl) = \lambda k. \ \texttt{batched} \ (k :: bl) \ fl\\ \texttt{dequeue}(\texttt{batched} \ bl \ []) = \texttt{step}^{|bl|}(-)\\ \begin{cases} \langle \texttt{none},\texttt{batched} \ [] \ [] \rangle & \texttt{rev} \ bl = []\\ \langle \texttt{some}(k),\texttt{batched} \ [] \ fl \rangle & \texttt{rev} \ bl = k :: fl \end{cases}\\ \texttt{dequeue}(\texttt{batched} \ bl \ (k :: fl)) = \langle \texttt{some}(k),\texttt{batched} \ bl \ fl \rangle \end{cases}$

For all bl, fl : list(K), batched bl $fl = step_{Q}^{\Phi(bl,fl)}(spec (fl + rev bl)).$

Theorem

For all bl, fl : list(K),

batched
$$bl \ fl = \operatorname{step}_Q^{\Phi(bl, fl)}(\operatorname{spec} \ (fl + \operatorname{rev} \ bl)).$$

Proof.

By coinduction.

Definition (Sequence of Operations, Free Monad)

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

Definition (Sequence of Operations, Free Monad)

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

Definition (Sequence Evaluation)

 $eval: P(A) \rightarrow UQ \rightarrow A \ltimes F1$

By induction on the operation sequence P(A).

Definition (Sequence of Operations, Free Monad)

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

Definition (Sequence Evaluation)

 $eval: P(A) \rightarrow UQ \rightarrow A \ltimes F1$

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*),

eval $p q_1 = eval p q_2$.

Definition (Sequence of Operations, Free Monad)

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

Definition (Sequence Evaluation)

 $eval: P(A) \rightarrow UQ \rightarrow A \ltimes F1$

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*),

eval $p q_1 = eval p q_2$.

Theorem (Coinductive vs. Classic Amortized Analysis) For all q_1 and q_2 , $q_1 = q_2$ iff $q_1 \approx q_2$.

Conclusion

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

- 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".

- 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.

- 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!

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

Theorem

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

Theorem

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

Proof.

We prove by coinduction, showing:

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

Coinductive Equivalence

Theorem

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

Proof.

$$\begin{aligned} & \texttt{quit}(\texttt{daily}) = \texttt{ret}(\langle \rangle) \\ & \texttt{quit}(\texttt{monthly } d) = \texttt{step}_{\texttt{F1}}^{\Phi(d)}(\texttt{ret}(\langle \rangle)) \end{aligned}$$

We show:

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

Coinductive Equivalence

Theorem

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

Proof.

$$\begin{aligned} & \textbf{remain}(\texttt{daily}) = \texttt{step}_R^{\$20}(\texttt{daily}) \\ & \textbf{remain}(\texttt{monthly 29}) = \texttt{step}_R^{\$600}(\texttt{monthly 0}) \end{aligned}$$

We show:

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

Coinductive Equivalence

Theorem

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

Proof.

$$\begin{aligned} & \texttt{remain}(\texttt{daily}) = \texttt{step}_R^{\$20}(\texttt{daily}) \\ & \texttt{remain}(\texttt{monthly} \ d) = \texttt{monthly} \ (d+1) \end{aligned}$$

We show:

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

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.

References ii



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.

References iii



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.

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.