Amortized Analysis via Coinduction

Harrison Grodin, j.w.w. Robert Harper
May 19, 2023
Carnegie Mellon University
### Table of contents

<table>
<thead>
<tr>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understand <em>amortized analysis in call-by-push-value</em>, using <em>coinduction</em>.</td>
</tr>
</tbody>
</table>

1. Call-By-Push-Value

2. Abstract Data Types, Coinductively

3. Amortized Analysis
   - Renting
   - Queue

4. Conclusion
Call-By-Push-Value
In call-by-push-value, types are separated into two sorts:

**Positive/Value Types**

\[ A, B, C ::= \]

\[ U \times A + B \]

\[ A \otimes B \mu (A . B (A)) \]

**Negative/Computation Types**

\[ X, Y, Z ::= \]

\[ F A \times X \times Y A \to X \nu (X . Y (X)) \]

\[ A \triangledown X \]
In call-by-push-value, types are separated into two sorts:

### Positive/Value Types

\[
A, B, C ::= \\
0 \\
A + B \\
1 \\
A \otimes B \\
\mu(A, B(A))
\]
In call-by-push-value, types are separated into two sorts:

**Positive/Value Types**

\[
A, B, C ::= \\
0 \\
A + B \\
1 \\
A \otimes B \\
\mu(A. B(A))
\]

**Negative/Computation Types**

\[
X, Y, Z ::= \\
1 \\
X \times Y \\
A \rightarrow X \\
\nu(X. Y(X)) \\
A \ltimes X
\]
In call-by-push-value, types are separated into two sorts:

<table>
<thead>
<tr>
<th>Positive/Value Types</th>
<th>Negative/Computation Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A, B, C ::= UX$</td>
<td>$X, Y, Z ::= 1$</td>
</tr>
<tr>
<td>0</td>
<td>$X \times Y$</td>
</tr>
<tr>
<td>$A + B$</td>
<td>$A \rightarrow X$</td>
</tr>
<tr>
<td>1</td>
<td>$\nu(X. Y(X))$</td>
</tr>
<tr>
<td>$A \otimes B$</td>
<td>$A \ltimes X$</td>
</tr>
</tbody>
</table>
| $\mu(A. B(A))$               | 平

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

<table>
<thead>
<tr>
<th>Positive/Value Types</th>
<th>Negative/Computation Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A, B, C ::= UX )</td>
<td>( X, Y, Z ::= FA )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( A + B )</td>
<td>( X \times Y )</td>
</tr>
<tr>
<td>1</td>
<td>( A \rightarrow X )</td>
</tr>
<tr>
<td>( A \otimes B )</td>
<td>( \nu(X \cdot Y(X)) )</td>
</tr>
<tr>
<td>( \mu(A \cdot B(A)) )</td>
<td>( A \triangleleft X )</td>
</tr>
</tbody>
</table>
In **calf** (based on CBPV), costs are annotated via an effect:

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

- **Cost model**: 1 cost per addition.

```plaintext
sum : list(nat) → F(nat)
sum[] = ret(0)
sum(x::l) = n←sum l; step_{x}(1)(x+n)
```
In \texttt{calf} (based on CBPV), costs are annotated via an effect:

\[
\Gamma \vdash e : X \\
\Gamma \vdash \text{step}^c_X(e) : X
\]

**Summing a List**

Cost model: 1 cost per addition.

\[
\text{sum} : \text{list(nat)} \rightarrow F(\text{nat})
\]

\[
\text{sum} \ [] =
\]

\[
\text{sum} \ (x :: l) =
\]
In **calf** (based on CBPV), costs are annotated via an effect:

\[ \Gamma \vdash e : X \quad \Gamma \vdash \text{step}_X^c(e) : X \]

**Summing a List**

Cost model: 1 cost per addition.

\[ \text{sum} : \text{list(nat)} \rightarrow \text{F(nat)} \]

\[ \text{sum} [] = \text{ret}(0) \]

\[ \text{sum} (x :: l) = \]
Cost as an Effect

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

\[
\Gamma \vdash e : X \\
\Gamma \vdash \text{step}^c_X(e) : X
\]

Summing a List

Cost model: 1 cost per addition.

\[
\text{sum} : \text{list(nat)} \rightarrow F(\text{nat}) \\
\text{sum} [] = \text{ret}(0) \\
\text{sum} (x :: l) = n \leftarrow \text{sum} \; l;
\]
In `calf` (based on CBPV), costs are annotated via an effect:

\[
\Gamma \vdash e : X \\
\Gamma \vdash \text{step}^c_X(e) : X
\]

### Summing a List

Cost model: 1 cost per addition.

\[
\begin{align*}
\text{sum} &: \text{list(nat)} \rightarrow F(\text{nat}) \\
\text{sum} \; [] &= \text{ret}(0) \\
\text{sum} \; (x :: l) &= n \leftarrow \text{sum} \; l; \; \text{step}^1(x + n)
\end{align*}
\]
Negative types know how to “consume” cost.
Negative types know how to “consume” cost.

\[ \text{step}^c_{XY}(\langle x, y \rangle) \triangleq \langle \text{step}^c_X(x), \text{step}^c_Y(y) \rangle \]
Negative types know how to “consume” cost.

\[ \text{step}^c_{X \times Y}(\langle x, y \rangle) \triangleq \langle \text{step}^c_X(x), \text{step}^c_Y(y) \rangle \]

Ultimately, steps accumulate when computing a value \( FA \).
Negative types know how to “consume” cost.

\[
\text{step}_x^c((x, y)) \triangleq (\text{step}_x^c(x), \text{step}_y^c(y))
\]

Ultimately, steps accumulate when computing a value \(FA\).

**Key Idea**

Cost incurred at a negative type gets “pushed down” to \(F\) types.
Abstract Data Types, Coinductively
### Queue

<table>
<thead>
<tr>
<th>enqueue([k])</th>
<th>~ 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>dequeue</td>
<td>~ (K + 1)</td>
</tr>
</tbody>
</table>
### Queue

| enqueue\[k\] | 1 |
| dequeue | \(K + 1\) |

\[Q \cong (\text{quit} : F1) \times (\text{enqueue} : K \rightarrow Q) \times (\text{dequeue} : (K + 1) \times Q)\]
### Queue

| enqueue[k] | \sim | 1 |
| dequeue    | \sim | K + 1 |

\[
Q \cong (\text{quit} : F1) \times (\text{enqueue} : K \rightarrow Q) \times (\text{dequeue} : (K + 1) \times Q)
\]

### Renting an Apartment

| remain      | \sim | 1 |
### Queue

- $\text{enqueue}[k] \sim 1$
- $\text{dequeue} \sim K + 1$

$$Q \equiv (\text{quit} : F1) \times (\text{enqueue} : K \to Q) \times (\text{dequeue} : (K + 1) \times Q)$$

### Renting an Apartment

- $\text{remain} \sim 1$

$$R \equiv (\text{quit} : F1) \times (\text{remain} : R)$$
Remark

The coinductive “machine” types look like object-oriented programming.
Remark

The coinductive “machine” types look like object-oriented programming.

\[ R \cong (\text{quit} : F1) \times (\text{remain} : R) \]

Example

Suppose \( r : R \); then:

\[ r.\text{remain}.\text{remain}.\text{remain}.\text{remain}.\text{quit} : F1. \]
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
\[ R \cong (\text{quit} : F1) \times (\text{remain} : R) \]
Payment Scheme: Daily

\[ R \cong (\text{quit} : F_1) \times (\text{remain} : R) \]

<table>
<thead>
<tr>
<th>Daily Payment</th>
</tr>
</thead>
<tbody>
<tr>
<td>daily: ( R )</td>
</tr>
<tr>
<td>quit(daily) =</td>
</tr>
<tr>
<td>remain(daily) =</td>
</tr>
</tbody>
</table>
Payment Scheme: Daily

\[ R \cong (\text{quit} : F1) \times (\text{remain} : R) \]

Daily Payment

\[ \text{daily} : R \]
\[ \text{quit} (\text{daily}) = \text{ret}(\langle \rangle) \]
\[ \text{remain} (\text{daily}) = \]
Payment Scheme: Daily

\[ R \cong (\text{quit} : F1) \times (\text{remain} : R) \]

<table>
<thead>
<tr>
<th>Daily Payment</th>
</tr>
</thead>
<tbody>
<tr>
<td>daily ( : R )</td>
</tr>
<tr>
<td>( \text{quit}(\text{daily}) = \text{ret}(\emptyset) )</td>
</tr>
<tr>
<td>( \text{remain}(\text{daily}) = \text{step}^{$20}_R (\text{daily}) )</td>
</tr>
</tbody>
</table>
**Payment Scheme: Monthly**

\[ R \cong (\text{quit} : F1) \times (\text{remain} : R) \]

---

**Monthly Payment**

\[
\text{monthly} : \text{nat}_{<30} \rightarrow R
\]

\[
\text{quit}(\text{monthly} \ d) = \\
\text{remain}(\text{monthly} \ 29) = \\
\text{remain}(\text{monthly} \ d) =
\]

- \( d \) is the day of the month
Monthly Payment

\[
R \equiv (\text{quit : } F1) \times (\text{remain : } R)
\]

\[
\text{monthly : } \text{nat}_{<30} \rightarrow R
\]

\[
\text{quit(monthly } d) =
\]

\[
\text{remain(monthly } 29) =
\]

\[
\text{remain(monthly } d) =
\]

- \(d\) is the day of the month
- \(\Phi(d) = $20d\) is the money owed for the month so far
\[ R \cong (\text{quit} : F_1) \times (\text{remain} : R) \]

**Monthly Payment**

\[
\text{monthly} : \text{nat}_{< 30} \rightarrow R \\
\text{quit} (\text{monthly } d) = \text{step}_{F_1}^{\Phi(d)} (\text{ret}(\langle \rangle)) \\
\text{remain} (\text{monthly } 29) = \\
\text{remain} (\text{monthly } d) =
\]

- \( d \) is the day of the month
- \( \Phi(d) = 20d \) is the money owed for the month so far
Payment Scheme: Monthly

\[ R \cong (\text{quit} : F1) \times (\text{remain} : R) \]

**Monthly Payment**

\[
\begin{align*}
\text{monthly} : \text{nat}_{<30} &\rightarrow R \\
\text{quit} (\text{monthly } d) &= \text{step}^{\Phi(d)}_{F1} (\text{ret}(\emptyset)) \\
\text{remain} (\text{monthly } 29) &= \text{step}^{\$600}_{R} (\text{monthly } 0) \\
\text{remain} (\text{monthly } d) &= \text{step}^{\$20d}_{R} (\text{remain} (\text{monthly } d))
\end{align*}
\]

- \( d \) is the day of the month
- \( \Phi(d) = \$20d \) is the money owed for the month so far
$R \equiv (\text{quit} : F1) \times (\text{remain} : R)$

**Monthly Payment**

$$\text{monthly} : \text{nat}_{\leq 30} \rightarrow R$$

\[
\begin{align*}
\text{quit}(\text{monthly } d) &= \text{step}_{F1}^{\Phi(d)}(\text{ret}()) \\
\text{remain}(\text{monthly } 29) &= \text{step}_R^{600}(\text{monthly } 0) \\
\text{remain}(\text{monthly } d) &= \text{monthly } (d + 1)
\end{align*}
\]

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

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

Theorem

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

Proof.

By coinduction:

- In the quit case, both incur the same number of steps.
- In the remain case, push cost forward and use the co-IH.
Amortizing Full Stays

What about the usual definition of equivalence?

\begin{definition}
\text{(Full-Stay Evaluation)}
\end{definition}

\begin{align*}
\text{eval } & : \mathbb{N} \to \mathcal{U} \to \mathcal{F}_1 \\
\text{eval}_0 & : r \mapsto \text{quit}(r) \\
\text{eval} & : (n+1) \cdot r \mapsto \text{eval}_n \cdot \text{remain}(r)
\end{align*}

\begin{definition}
\text{(Full-Stay Evaluation Equivalence)}
\end{definition}

Say \( r_1 \approx r_2 \) iff for all \( n \), \( \text{eval}_n \cdot r_1 = \text{eval}_n \cdot r_2 \).

\begin{theorem}
For all \( r_1 \) and \( r_2 \), \( r_1 = r_2 \) iff \( r_1 \approx r_2 \).
\end{theorem}

\begin{proof}
By routine induction on \( n \) and coinduction on \( r_1 \approx r_2 \).
\end{proof}
Amortizing Full Stays

What about the usual definition of equivalence?

<table>
<thead>
<tr>
<th>Definition (Full-Stay Evaluation)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>eval</strong>: ( \text{nat} \rightarrow \text{UR} \rightarrow \text{F1} )</td>
</tr>
<tr>
<td>eval 0 ( r ) = <strong>quit</strong>(( r ))</td>
</tr>
<tr>
<td>eval ((n + 1)) ( r ) = eval ( n ) (<strong>remain</strong> ( r ))</td>
</tr>
</tbody>
</table>
Amortizing Full Stays

What about the usual definition of equivalence?

### Definition (Full-Stay Evaluation)

\[
\text{eval} : \text{nat} \to \text{UR} \to \text{F1}
\]

\[
\text{eval} \ 0 \quad r = \text{quit}(r)
\]

\[
\text{eval} \ (n + 1) \ r = \text{eval} \ n \ (\text{remain} \ r)
\]

### Definition (Full-Stay Evaluation Equivalence)

Say \( r_1 \approx r_2 \) iff for all \( n \), \( \text{eval} \ n \ r_1 = \text{eval} \ n \ r_2 \).
Amortizing Full Stays

What about the usual definition of equivalence?

Definition (Full-Stay Evaluation)

\[
\text{eval} : \text{nat} \to UR \to \text{F1}
\]

\[
\text{eval } 0 \quad r = \text{quit}(r)
\]

\[
\text{eval } (n + 1) \; r = \text{eval } n \; (\text{remain } r)
\]

Definition (Full-Stay Evaluation Equivalence)

Say \( r_1 \approx r_2 \) iff for all \( n \), \( \text{eval } n \; r_1 = \text{eval } n \; r_2 \).

Theorem

For all \( r_1 \) and \( r_2 \), \( r_1 = r_2 \) iff \( r_1 \approx r_2 \).
Amortizing Full Stays

What about the usual definition of equivalence?

Definition (Full-Stay Evaluation)

\[
\begin{align*}
\text{eval} &: \text{nat} \rightarrow UR \rightarrow F1 \\
\text{eval} 0 r &= \text{quit}(r) \\
\text{eval} (n + 1) r &= \text{eval} n (\text{remain } r)
\end{align*}
\]

Definition (Full-Stay Evaluation Equivalence)

Say \( r_1 \approx r_2 \) iff for all \( n \), \( \text{eval } n r_1 = \text{eval } n r_2 \).

Theorem

For all \( r_1 \) and \( r_2 \), \( r_1 = r_2 \) iff \( r_1 \approx r_2 \).

Proof.

By routine induction on \( n \) and coinduction on \( r_1 \approx r_2 \).
Amortized Analysis

Queue
Queue Implementation: Specification

\[ Q \triangleq (\text{quit} : \mathbb{F}1) \times (\text{enqueue} : K \rightarrow Q) \times (\text{dequeue} : (K + 1) \times Q) \]
Queue Implementation: Specification

\[ Q \simeq (\text{quit} : F1) \times (\text{enqueue} : K \rightarrow Q) \times (\text{dequeue} : (K + 1) \times Q) \]

**Specification**

\[
\text{spec} : \text{list}(K) \rightarrow Q
\]

\[
\text{quit}(\text{spec } l) = \text{ret}(\langle \rangle)
\]

\[
\text{enqueue}(\text{spec } l) = 
\]

\[
\text{dequeue}(\text{spec } []) = 
\]

\[
\text{dequeue}(\text{spec } (k :: l)) = 
\]
Queue Implementation: Specification

\[ Q \cong (\text{quit} : F1) \times (\text{enqueue} : K \to Q) \times (\text{dequeue} : (K + 1) \times Q) \]

**Specification**

\[
\text{spec} : \text{list}(K) \to Q \\
\text{quit}(\text{spec } l) = \text{ret}([\emptyset]) \\
\text{enqueue}(\text{spec } l) = \lambda k. \text{step}_Q^1(\text{spec } (l + [k])) \\
\text{dequeue}(\text{spec } []) = \\
\text{dequeue}(\text{spec } (k :: l)) =
\]
Queue Implementation: Specification

\[ Q \simeq (\text{quit} : F1) \times (\text{enqueue} : K \rightarrow Q) \times (\text{dequeue} : (K + 1) \times Q) \]

**Specification**

\[
\begin{align*}
\text{spec} &: \text{list}(K) \rightarrow Q \\
\text{quit}(\text{spec } l) &= \text{ret}(\langle \rangle) \\
\text{enqueue}(\text{spec } l) &= \lambda k. \text{step}^1_Q(\text{spec } (l + [k])) \\
\text{dequeue}(\text{spec } []) &= \langle \text{none}, \text{spec } [] \rangle \\
\text{dequeue}(\text{spec } (k :: l)) &=
\end{align*}
\]
Queue Implementation: Specification

\[ Q \cong (\text{quit} : F \times 1) \times (\text{enqueue} : K \to Q) \times (\text{dequeue} : (K + 1) \times Q) \]

### Specification

\[
\begin{align*}
\text{spec} &: \text{list}(K) \to Q \\
\text{quit}(\text{spec } l) &= \text{ret}(\langle \rangle) \\
\text{enqueue}(\text{spec } l) &= \lambda k. \text{step}_Q^1(\text{spec } (l + [k])) \\
\text{dequeue}(\text{spec } []) &= \langle \text{none}, \text{spec } [] \rangle \\
\text{dequeue}(\text{spec } (k :: l)) &= \langle \text{some}(k), \text{spec } l \rangle
\end{align*}
\]
Batched Queue

\[
\text{batched} : \text{list}(K) \rightarrow \text{list}(K) \rightarrow Q
\]

\[
\text{quit} (\text{batched } bl \ fl) = \\
\text{enqueue} (\text{batched } bl \ fl) = \\
\text{dequeue} (\text{batched } bl \ []) = \\
\]

\[
\text{dequeue} (\text{batched } bl \ (k :: fl)) = \\
\]

Here, \( \Phi(bl, fl) = |bl| \) (how much spec should have already paid).
Queue Implementation: Batched (Amortized)

**Batched Queue**

\[
\text{batched} : \text{list}(K) \rightarrow \text{list}(K) \rightarrow Q
\]

\[
\text{quit}(\text{batched } bl \ fl) = \text{step}^{\Phi_{F_1}}(\text{ret}(\langle \rangle))
\]

\[
\text{enqueue}(\text{batched } bl \ fl) =
\]

\[
\text{dequeue}(\text{batched } bl \ [\ ]) =
\]

\[
\text{dequeue}(\text{batched } bl \ (k :: fl)) =
\]

Here, \( \Phi(bl, fl) = |bl| \) (how much spec should have already paid).
Batched Queue

\[ \text{batched} : \text{list}(K) \to \text{list}(K) \to Q \]

\[ \text{quit}(\text{batched } bl \ fl) = \text{step}_{F_1}^{\Phi(bl,fl)}(\text{ret}(\langle \rangle)) \]

\[ \text{enqueue}(\text{batched } bl \ fl) = \lambda k. \ \text{batched} \ (k :: bl) \ fl \]

\[ \text{dequeue}(\text{batched } bl \ []) = \]

\[ \text{dequeue}(\text{batched } bl \ (k :: fl)) = \]

Here, \( \Phi(bl, fl) = |bl| \) (how much \text{spec} should have already paid).
Queue Implementation: Batched (Amortized)

**Batched Queue**

\[ \text{batched} : \text{list}(K) \to \text{list}(K) \to Q \]

\[ \text{quit}(\text{batched } bl \ fl) = \text{step}_{F_1}^{\Phi(bl, fl)}(\text{ret}(\langle \rangle)) \]

\[ \text{enqueue}(\text{batched } bl \ fl) = \lambda k. \text{batched } (k :: bl) \ fl \]

\[ \text{dequeue}(\text{batched } bl [\ ]) = \text{step}^{|bl|}(-) \]

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

\[ \text{dequeue}(\text{batched } bl (k :: fl)) = \]

Here, \( \Phi(bl, fl) = |bl| \) (how much \texttt{spec} should have already paid).
Queue Implementation: Batched (Amortized)

**Batched Queue**

\[
\text{batched} : \text{list}(K) \rightarrow \text{list}(K) \rightarrow Q
\]

\[
\text{quit}(\text{batched } bl \ fl) = \text{step}_{F_1}^{\Phi(bl, fl)}(\text{ret}(\langle \rangle))
\]

\[
\text{enqueue}(\text{batched } bl \ fl) = \lambda k. \text{batched } (k :: bl) \ fl
\]

\[
\text{dequeue}(\text{batched } bl \ []) = \text{step}^{|bl|}(-)
\]

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

\[
\text{dequeue}(\text{batched } bl \ (k :: fl)) = \langle \text{some}(k), \text{batched } bl \ fl \rangle
\]

Here, \( \Phi(bl, fl) = |bl| \) (how much \texttt{spec} should have already paid).
Theorem

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

\[
\text{batched } bl \ fl = \text{step}^{\Phi(bl,fl)}(\text{spec } (fl + \text{rev } bl)).
\]
Amortizing Finite Sequences of Operations

Definition (Sequence of Operations)

\[ P(A) \sim \phantom{=} \left( \text{ret: } A \right) + \left( \text{enq: } K \otimes P(A) \right) + \left( \text{deq: } U(K+1 \rightarrow F(P(A)) \right) \]

Definition (Sequence Evaluation)

\[ \text{eval}: P(A) \rightarrow UQ \rightarrow A \bowtie F \]

By recursion on the operation sequence \( P(A) \).

Definition (Classic Amortized Equivalence)

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

\[ \text{eval} p q_1 = \text{eval} p q_2. \]

Theorem (Coinductive vs. Classic Amortized Analysis)

For all \( q_1 \) and \( q_2 \), \( q_1 = q_2 \) iff \( q_1 \approx q_2 \).
Amortizing Finite Sequences of Operations

**Definition (Sequence of Operations)**

\[ P(A) \equiv (\text{ret} : A) + (\text{enq} : K \otimes P(A)) + (\text{deq} : U(K + 1 \rightarrow F(P(A)))) \]
### Amortizing Finite Sequences of Operations

<table>
<thead>
<tr>
<th>Definition (Sequence of Operations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P(A) \triangleright (\text{ret} : A) + (\text{enq} : K \otimes P(A)) + (\text{deq} : U(K + 1 \rightarrow F(P(A))))$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Definition (Sequence Evaluation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{eval} : P(A) \rightarrow UQ \rightarrow A \times F1$</td>
</tr>
</tbody>
</table>

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

---

15
Amortizing Finite Sequences of Operations

**Definition (Sequence of Operations)**

\[ P(A) \sim (\text{ret} : A) + (\text{enq} : K \otimes P(A)) + (\text{deq} : U(K + 1 \to F(P(A)))) \]

**Definition (Sequence Evaluation)**

\[ \text{eval} : P(A) \to UQ \to A \times F1 \]

By recursion on the operation sequence \( P(A) \).

**Definition (Classic Amortized Equivalence)**

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

\[ \text{eval } p \ q_1 = \text{eval } p \ q_2. \]
## Amortizing Finite Sequences of Operations

### Definition (Sequence of Operations)

\[
P(A) \simeq (\text{ret} : A) + (\text{enq} : K \otimes P(A)) + (\text{deq} : U(K + 1 \rightarrow F(P(A))))
\]

### Definition (Sequence Evaluation)

\[
\text{eval} : P(A) \rightarrow UQ \rightarrow A \times F1
\]

By recursion on the operation sequence \( P(A) \).

### Definition (Classic Amortized Equivalence)

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

\[
\text{eval } p q_1 = \text{eval } p q_2.
\]

### Theorem (Coinductive vs. Classic Amortized Analysis)

For all \( q_1 \text{ and } q_2 \), \( q_1 = q_2 \) iff \( q_1 \approx q_2 \).
Conclusion
1. In call-by-push-value, effects propagate through negative types.
1. In call-by-push-value, effects propagate through negative types.
2. Sequential-use data structures are coinductive/object-oriented “machines”.
1. In call-by-push-value, effects propagate through negative types.
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 negative types.
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 negative types.
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).
Summary

1. In call-by-push-value, effects propagate through negative types.
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).

To appear at CALCO 2023.
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.
Bonus
Theorem

For all \(d\), \textbf{monthly} \(d = \text{step}^{\Phi(d)}(\text{daily})\).
Theorem

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

Proof.
We prove by coinduction, showing:

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

<table>
<thead>
<tr>
<th>Theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>For all ( d ), monthly ( d = \text{step}^{\Phi(d)}(\text{daily}) ).</td>
</tr>
</tbody>
</table>

Proof.

\[
\text{quit}(\text{daily}) = \text{ret}(\langle \rangle) \\
\text{quit}(\text{monthly } d) = \text{step}_{F_1}^{\Phi(d)}(\text{ret}(\langle \rangle))
\]

We show:

\[
\text{quit}(\text{monthly } d) = \text{step}^{\Phi(d)}(\text{ret}(\langle \rangle)) \\
= \text{step}^{\Phi(d)}(\text{quit}(\text{daily})) \\
= \text{quit}(\text{step}^{\Phi(d)}(\text{daily}))
\]
Coinductive Equivalence

**Theorem**

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

**Proof.**

$$\text{remain}(\text{daily}) = \text{step}^R_{20}(\text{daily})$$
$$\text{remain}(\text{monthly 29}) = \text{step}^R_{600}(\text{monthly 0})$$

We show:

$$\text{remain}(\text{monthly 29}) = \text{step}^R_{600}(\text{monthly 0})$$
$$= \text{step}^R_{600}(\text{daily})$$
$$= \text{step}^{\Phi(29)}(\text{step}^R_{20}(\text{daily}))$$
$$= \text{step}^{\Phi(29)}(\text{remain}(\text{daily}))$$
$$= \text{remain}(\text{step}^{\Phi(29)}(\text{daily}))$$
Coinductive Equivalence

Theorem

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

Proof.

\[
\text{remain}(\text{daily}) = \text{step}^{S_20}(\text{daily}) \\
\text{remain}(\text{monthly } d) = \text{monthly } (d + 1)
\]

We show:

\[
\text{remain}(\text{monthly } d) = \text{monthly } (d + 1) \\
= \text{step}^{\Phi(d+1)}(\text{daily}) \quad \text{(co-IH)} \\
= \text{step}^{\Phi(d)}(\text{step}^{S_20}(\text{daily})) \\
= \text{step}^{\Phi(d)}(\text{remain}(\text{daily})) \\
= \text{remain}(\text{step}^{\Phi(d)}(\text{daily}))
\]
W. R. Cook.

Object-oriented programming versus abstract data types.

W. R. Cook.

On understanding data abstraction, revisited.
B. Jacobs.

**Mongruences and cofree coalgebras.**

B. Jacobs.

**Objects And Classes, Co-Algebraically.**

P. B. Levy.

**Call-By-Push-Value.**
A cost-aware logical framework.  

R. E. Tarjan.  
Amortized Computational Complexity.  