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.
Table of contents

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

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

**Positive/Value Types**

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

0 \( A + B \)

1 \( A \times B \)

\( \mu(A. B(A)) \)
In call-by-push-value, types are separated into two sorts:

### Positive/Value Types

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

Interpreted in \textbf{Set}. 
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 \ ::= )</td>
<td>( X, Y, Z \ ::= )</td>
</tr>
<tr>
<td>0 ( A + B )</td>
<td>1 ( X \times Y )</td>
</tr>
<tr>
<td>1 ( A \times B )</td>
<td>( A \rightarrow X )</td>
</tr>
<tr>
<td>( \mu(A. B(A)) )</td>
<td>( \nu(X. Y(X)) )</td>
</tr>
</tbody>
</table>

Interpreted in \textbf{Set}.
In call-by-push-value, types are separated into two sorts:

### Positive/Value Types

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

\[ 0 \ A + B \]

\[ 1 \ A \times B \]

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

Interpreted in \( \text{Set} \).

### Negative/Computation Types

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

\[ 1 \ X \times Y \]

\[ A \rightarrow X \]

\[ \nu(X. \ Y(X)) \]

Interpreted in \( \text{Set}^T \), for monad \( T \).
Type Polarity

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

**Positive/Value Types**

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

- 0 \( A + B \)
- 1 \( A \times B \)
- \( \mu(A, B(A)) \)

Interpreted in \( \textbf{Set} \).

**Negative/Computation Types**

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

- 1 \( X \times Y \)
- \( A \rightarrow X \)
- \( \nu(X. Y(X)) \)

Interpreted in \( \textbf{Set}^T \), for monad \( T \).
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 \ A + B )</td>
<td>( 1 \ X \times Y )</td>
</tr>
<tr>
<td>( 1 \ A \times B )</td>
<td>( A \rightarrow X )</td>
</tr>
<tr>
<td>( \mu(A. B(A)) )</td>
<td>( \nu(X. Y(X)) )</td>
</tr>
</tbody>
</table>

Interpreted in **Set**.

Interpreted in **Set**\(^T\), for monad \( T \).
Semantics of Computation Types

In $\text{Set}^T$, an object $X$ has a set $UX$ and a map $\alpha_X : T(UX) \rightarrow UX$. 

Definition (Free Algebra) $U(FA) = T\alpha FA = TT\alpha FA$ 

Definition (Product Algebra) $U(X \times Y) = U X \times U Y \alpha _X \times \alpha _Y \rightarrow T(U X \times U Y) \rightarrow U X \times U Y$ 

Key Idea Effects "flow over" computation types (accumulating at $F$ types).
In $\text{Set}^T$, 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 $\text{Set}^T$, 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)**

$$U(X \times Y) = UX \times UY$$

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

**Key Idea**

Effects "flow over" computation types (accumulating at $F$ types).
In $\mathbf{Set}^T$, 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)**

$$U(X \times Y) = UX \times UY$$

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

**Key Idea**

Effects “flow over” computation types (accumulating at $F$ types).
In *calf* (based on CBPV), costs are annotated via an effect:

\[
\Gamma \vdash e : X \\
\Gamma \vdash \text{step}_X^c(e) : X
\]
In calf (based on CBPV), costs are annotated via an effect:

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

Here, monad \( T = \mathbb{N} \times (-) \).
In calf (based on CBPV), costs are annotated via an effect:

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

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

**Example (Summing a List)**

Cost model: 1 cost per addition.

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

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

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

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

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

**Example (Summing a List)**

Cost model: 1 cost per addition.

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

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

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

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

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

**Example (Summing a List)**

Cost model: 1 cost per addition.

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

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

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

**Example (Summing a List)**

Cost model: 1 cost per addition.

- \( \text{sum} : \text{list}(\mathbb{N}) \to F(\mathbb{N}) \)
- \( \text{sum }[] = \text{ret}(0) \)
- \( \text{sum } (x :: l) = n \leftarrow \text{sum } l; \text{step}^1(x + n) \)
In **calf** (CBPV with writer monad), we have a “mixed product”: 

\[ A \triangleright X \]
In calf (CBPV with writer monad), we have a “mixed product”:

\[ A \ltimap X \]

**Definition (Mixed Product Algebra)**

\[
\begin{align*}
U(A \ltimap X) &= A \times UX \\
\alpha_{A \ltimap X} &= \mathbb{N} \times (A \times UX) \cong A \times (\mathbb{N} \times UX) \\
&\xrightarrow{\text{id}_A \times \alpha_X} A \times UX
\end{align*}
\]
In **calf** (CBPV with writer monad), we have a “mixed product”:

\[ A \ltimes X \]

### Definition (Mixed Product Algebra)

\[ U(A \ltimes X) = A \times UX \]

\[ \alpha_{A \ltimes X} = N \times (A \times UX) \cong A \times (N \times UX) \xrightarrow{id_A \times \alpha_X} A \times UX \]

### Lemma

\[ 1 \ltimes X \cong X \]
Abstract Data Types, Coinductively
Consider an operation signature:

\[
\begin{align*}
op_1 & \rightsquigarrow A_1 \\
\vdots \\
op_n & \rightsquigarrow A_n
\end{align*}
\]
Abstract Data Types, Coinductively

Consider an operation signature:

\[
\text{op}_1 \rightsquigarrow A_1 \\
\vdots \\
\text{op}_n \rightsquigarrow A_n
\]

Work with \textit{cofree comonad}:

\[
DX \triangleq \nu \{ Z. (\text{quit} : X) \times (\text{op}_1 : A_1 \times Z) \times \cdots \times (\text{op}_n : A_n \times Z) \}\]
Abstract Data Types, Coinductively

Consider an operation signature:

\[
\begin{align*}
\text{op}_1 & \leadsto A_1 \\
\vdots & \\
\text{op}_n & \leadsto A_n
\end{align*}
\]

Work with cofree comonad:

\[
DX \triangleq \nu(Z. (\text{quit} : X) \times (\text{op}_1 : A_1 \times Z) \times \cdots \times (\text{op}_n : A_n \times Z)) \\
\cong (\text{quit} : X) \times (\text{op}_1 : A_1 \times DX) \times \cdots \times (\text{op}_n : A_n \times DX)
\]
Abstract Data Types, Coinductively

Consider an operation signature:

\[
\text{op}_1 \leadsto A_1 \\
\vdots \\
\text{op}_n \leadsto A_n
\]

Work with cofree comonad:

\[
DX \triangleq \nu (Z. (\text{quit} : X) \times (\text{op}_1 : A_1 \times Z) \times \cdots \times (\text{op}_n : A_n \times Z)) \\
\cong (\text{quit} : X) \times (\text{op}_1 : A_1 \times DX) \times \cdots \times (\text{op}_n : A_n \times DX)
\]

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

\[
D \cong (\text{quit} : F1) \times (\text{op}_1 : A_1 \times D) \times \cdots \times (\text{op}_n : A_n \times D)
\]
# Example (Queue)

\[
\begin{align*}
\text{enqueue}[k : K] & \rightsquigarrow 1 \\
\text{dequeue} & \rightsquigarrow K + 1
\end{align*}
\]
Abstract Data Types, Coinductively

Example (Queue)

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

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

Example (Queue)

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

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

Example (Renting an Apartment)

\[
\text{remain} \rightsquigarrow 1
\]
Abstract Data Types, Coinductively

Example (Queue)

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

Example (Renting an Apartment)

\[ \text{remain} \rightsquigarrow 1 \]
\[ R \cong (\text{quit} : F1) \times (\text{remain} : R) \]
Remark

These coinductive types look like object-oriented programming.
Remark

These coinductive 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
Payment Scheme: Daily

\[ R \approx (\text{quit} : F1) \times (\text{remain} : R) \]
### Payment Scheme: Daily

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

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

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

**Daily Payment**

\[
\begin{align*}
\text{daily : } R \\
\text{\texttt{quit}(daily)} &= \text{ret(\{}\}\}) \\
\text{\texttt{remain}(daily)} &=
\end{align*}
\]
Payment Scheme: Daily

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

### Daily Payment

- **daily** : \( R \)
  - \( \text{quit}(\text{daily}) = \text{ret}(\langle \rangle) \)
  - \( \text{remain}(\text{daily}) = \text{step}_{R}^{20}(\text{daily}) \)
$R \cong (\text{quit} : F1) \times (\text{remain} : R)$

### Monthly Payment

<table>
<thead>
<tr>
<th>monthly : $\mathbb{N}_{&lt;30} \rightarrow R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{quit}(\text{monthly } d) =$</td>
</tr>
<tr>
<td>$\text{remain}(\text{monthly } 29) =$</td>
</tr>
<tr>
<td>$\text{remain}(\text{monthly } d) =$</td>
</tr>
</tbody>
</table>

- $d$ is the day of the month
Payment Scheme: Monthly

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

### Monthly Payment

\[
\text{monthly} : \mathbb{N} \{ < 30 \} \rightarrow R
\]

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

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

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

**Monthly Payment**

<table>
<thead>
<tr>
<th>monthly : (\mathbb{N}_{&lt;30} \rightarrow R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{quit}(\text{monthly } d) = \text{step}_{\text{F1}}^{\Phi(d)}(\text{ret}(\langle\rangle))</td>
</tr>
<tr>
<td>\text{remain}(\text{monthly } 29) =</td>
</tr>
<tr>
<td>\text{remain}(\text{monthly } d) =</td>
</tr>
</tbody>
</table>

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

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

Monthly Payment

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

\[
\text{quit}(\text{monthly } d) = \text{step}_{\text{F1}}^{\phi(d)}(\text{ret}(\langle \rangle))
\]

\[
\text{remain}(\text{monthly } 29) = \text{step}_{R}^{\$600}(\text{monthly } 0)
\]

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

- \(d\) 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

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

\[
\text{quit}(\text{monthly } d) = \text{step}^{\Phi(d)}_{F1} (\text{ret}(\langle \rangle))
\]

\[
\text{remain}(\text{monthly } 29) = \text{step}^{\$600}_{R} (\text{monthly } 0)
\]

\[
\text{remain}(\text{monthly } d) = \text{monthly } (d + 1)
\]

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

Coinductive Equivalence

**Theorem**

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

**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.
Theorem

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

Proof.

By coinduction:

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

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

Proof.

By coinduction:

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

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

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.
Theorem

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

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.
Theorem

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

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

Definition (Full-Stay Evaluation)

\[ \text{eval} : N \rightarrow UR \rightarrow F1 \]

\[ \text{eval} _0 (r) = \text{quit} (r) \]

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

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 \( r_2 \),

\[ r_1 = r_2 \iff r_1 \approx r_2 \]

Proof.

By \((\Rightarrow)\) induction on \( n \) and \((\Leftarrow)\) coinduction on \( r_1 = r_2 \).
Amortizing Full Stays

Definition (Full-Stay Evaluation)

\[ \text{eval} : \mathbb{N} \rightarrow UR \rightarrow F1 \]
\[ \text{eval} \ 0 \quad r = \text{quit}(r) \]
\[ \text{eval} \ (n + 1) \quad r = \text{eval} \ n \ (\text{remain} \ r) \]
Amortizing Full Stays

Definition (Full-Stay Evaluation)

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

Definition (Full-Stay Evaluation Equivalence)

Say \( r_1 \approx r_2 \) iff for all \( n \),

\[
\text{eval} \ n \ r_1 = \text{eval} \ n \ r_2.
\]
Amortizing Full Stays

**Definition (Full-Stay Evaluation)**

\[
\text{eval} : \mathbb{N} \to UR \to F1 \\
\text{eval } 0 \quad r = \text{quit}(r) \\
\text{eval } (n + 1) \ r = \text{eval } n \ (\text{remain } r)
\]

**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 \( r_1 \approx r_2 \).
### Definition (Full-Stay Evaluation)

\[
eval : \mathbb{N} \rightarrow UR \rightarrow F1
\]

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

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

### 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 (⇒) induction on \( n \) and (⇐) coinduction on \( r_1 = r_2 \).
Amortized Analysis

Queue
Queue Implementation: Specification

\[ Q \equiv (\text{quit} : F1) \times (\text{enqueue} : K \to Q) \times (\text{dequeue} : (K + 1) \times Q) \]
Queue Implementation: Specification

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

**Specification**

\[
\begin{align*}
\text{spec : list}(K) & \to Q \\
\text{quit}(\text{spec } l) & = \\
\text{enqueue}(\text{spec } l) & = \\
\text{dequeue}(\text{spec } []) & = \\
\text{dequeue}(\text{spec } (k :: l)) & =
\end{align*}
\]
Queue Implementation: Specification

\[ Q \equiv (\text{quit} : K_1 \times (\text{enqueue} : K \to Q) \times (\text{dequeue} : (K + 1) \times Q) \]

**Specification**

\[
\begin{align*}
\text{spec} & : \text{list}(K) \to Q \\
\text{quit}(\text{spec } l) & = \text{ret}(\langle \rangle) \\
\text{enqueue}(\text{spec } l) & = \\
\text{dequeue}(\text{spec } []) & = \\
\text{dequeue}(\text{spec } (k :: l)) & = 
\end{align*}
\]
Queue Implementation: Specification

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

Specification

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

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

\[ \text{enqueue}(\text{spec } l) = \lambda k. \text{step}_Q^1(\text{spec } (l + [k])) \]

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

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

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

**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

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

<table>
<thead>
<tr>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{spec} : \text{list}(K) \rightarrow Q )</td>
</tr>
<tr>
<td>( \text{quit}(\text{spec } l) = \text{ret}(\langle \rangle) )</td>
</tr>
<tr>
<td>( \text{enqueue}(\text{spec } l) = \lambda k. \text{step}_Q^1(\text{spec } (l + [k])) )</td>
</tr>
<tr>
<td>( \text{dequeue}(\text{spec } []) = \langle \text{none}, \text{spec } [] \rangle )</td>
</tr>
<tr>
<td>( \text{dequeue}(\text{spec } (k :: l)) = \langle \text{some}(k), \text{spec } l \rangle )</td>
</tr>
</tbody>
</table>
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 has already paid).
Batched Queue

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

\[
\text{quit}(\text{batched } bl \ fl) = \text{step}^{\Phi(bl, fl)}_{F1}(\text{ret}(\langle \rangle))
\]

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

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

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

**Batched Queue**

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Batched Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \text{batched} : \text{list}(K) \to \text{list}(K) \to Q ]</td>
</tr>
<tr>
<td>[ \text{quit} (\text{batched } bl \ fl) = \text{step}_{F_1}^{\Phi(\text{bl}, \text{fl})} \text{(ret(\textlangle\textrangle))} ]</td>
</tr>
<tr>
<td>[ \text{enqueue} (\text{batched } bl \ fl) = \lambda k. \text{batched } (k :: bl) \ fl ]</td>
</tr>
<tr>
<td>[ \text{dequeue} (\text{batched } bl \ []) = \text{step}_{1}^{</td>
</tr>
</tbody>
</table>

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

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

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

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

\[
\text{quit}(\text{batched } bl \ fl) = \text{step}^{\Phi(bl, fl)}_{F1} (\text{ret}(\langle \rangle))
\]

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

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

\[
\left\{ \begin{array}{l}
\langle \text{none}, \text{batched } [] \ [] \rangle \quad \text{rev } bl = [] \\
\langle \text{some}(k), \text{batched } [] \ fl \rangle \quad \text{rev } bl = k :: fl
\end{array} \right.
\]

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

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

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

$$\text{batched } bl\ fl = \text{step}_Q^{\Phi(bl, fl)}(\text{spec } (fl \mathbin{+\!+} \text{rev } bl)).$$
Coinductive Amortized Analysis

Theorem

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

\[
\text{batched } bl \; fl = \text{step}_Q^{\Phi(bl, fl)}(\text{spec } (fl \; \| \; \text{rev } bl)).
\]

Proof.

By coinduction.

Definition (Sequence of Operations, Free Monad)

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

Definition (Sequence Evaluation)

\[ \text{eval}: P(A) \to U Q \to A \bowtie F1 \]

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

Definition (Classic Amortized Equivalence)

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

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

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, Free Monad)

\[ P(A) \triangleq (\text{ret} : A) + (\text{enq} : K \times P(A)) + (\text{deq} : U(K + 1 \rightarrow F(P(A)))) \]
Definition (Sequence of Operations, Free Monad)

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

Definition (Sequence Evaluation)

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

By induction on the operation sequence \( P(A) \).
Definition (Sequence of Operations, Free Monad)

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

Definition (Sequence Evaluation)

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

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

Definition (Classic Amortized Equivalence)

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

\[ \text{eval} \ p \ q_1 = \text{eval} \ p \ q_2. \]
### Definition (Sequence of Operations, Free Monad)

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

### Definition (Sequence Evaluation)

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

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

### Definition (Classic Amortized Equivalence)

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

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

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

For all \( q_1 \) and \( q_2 \), \( q_1 = q_2 \) iff \( q_1 \approx q_2 \).
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”.

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

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
Theorem

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

**Theorem**

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

**Proof.**

We prove by coinduction, showing:

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

Theorem

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

Proof.

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

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

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

**Proof.**

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

We show:

\[
\text{remain}(\text{monthly 29}) = \text{step}^{\$600 R}(\text{monthly 0}) \\
= \text{step}^{\$600 R}(\text{daily}) \quad \text{(co-IH)} \\
= \text{step}^{\Phi(29)}(\text{step}^{\$20 R}(\text{daily})) \\
= \text{step}^{\Phi(29)}(\text{remain}(\text{daily})) \\
= \text{remain}(\text{step}^{\Phi(29)}(\text{daily}))
\]
Coinductive Equivalence

Theorem

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

Proof.

$$\text{remain}(\text{daily}) = \text{step}^{20}(\text{daily})$$

$$\text{remain}(\text{monthly } d) = \text{monthly } (d + 1)$$

We show:

$$\text{remain}(\text{monthly } d) = \text{monthly } (d + 1)$$

$$= \text{step}^{\Phi(d+1)}(\text{daily}) \quad \text{(co-IH)}$$

$$= \text{step}^{\Phi(d)}(\text{step}^{20}(\text{daily}))$$

$$= \text{step}^{\Phi(d)}(\text{remain}(\text{daily}))$$

$$= \text{remain}(\text{step}^{\Phi(d)}(\text{daily}))$$
References
A. Balan and A. Kurz.

On Coalgebras over Algebras.


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.**  
J. Power and O. Shkaravska.
From Comodels to Coalgebras: State and Arrays.

R. E. Tarjan.
Amortized Computational Complexity.