Proving Properties of Concurrent Data Structures

Papers from LICS'13 and CONCUR'13

Gautham Shenoy R

CMI

28 July 2013

Outline

1. Quantitative Reasoning for Proving Lock-Freedom - Jan Hoffman, Michael Marmar, Zhong Shao: In the proceedings of LICS 2013

2. Aspect-Oriented Linearizability Proofs: Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis: In the proceedings of CONCUR 2013

Outline

1. Quantitative Reasoning for Proving Lock-Freedom - Jan Hoffman, Michael Marmar, Zhong Shao: In the proceedings of LICS 2013

2. Aspect-Oriented Linearizability Proofs: Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis: In the proceedings of CONCUR 2013

Summary of the contributions:

Reduces proving *lock-freedom* to modular thread local termination of concurrent programs in which each thread executes a finite number of data-structure operations.

Introduces a *compensation based* quantitative reasoning technique for proving lock-freedom.

Formalises the technique by extending *Concurrent Separation Logic (CSL)* for total correctness.

Demonstrates the lock-free property exhibited by data structures including Treiber's non-blocking stack, Michael and Scott's lock-free queue, Hendler et al.'s lock-free stack with elimination back off and Michale's lock-free hazard pointer stack.

Lock Freedom

Consider a shared memory data structure which provides the users with finitely many operations to access/modify the contents of the data-structure.

Assume that at a given time there is a fixed but arbitrary number of threads that are repeatedly accessing the data-structure via the operations it provides.

Choose a point in the execution in which one or more operations have started.

Definition

Then lock-free *implementation of the data-structure guarantees that* some *thread will complete an operation in a finite number of steps.*

Lock Freedom and Termination

Definition

Let D *be a shared-memory data structure with* k*-operations denoted by* $\pi_1, \pi_2, \ldots, \pi_k$.

Lock Freedom and Termination

Definition

Let D be a shared-memory data structure with k-operations denoted by $\pi_1, \pi_2, \ldots, \pi_k$.

Let P be a concurrent program with finitely many (say m) threads, with the i^{th} thread executing the program S_i.

$$P = S_1 \mid\mid \ldots \mid\mid S_m$$

Lock Freedom and Termination

Definition

Let D be a shared-memory data structure with k-operations denoted by $\pi_1, \pi_2, \ldots, \pi_k$.

Let P be a concurrent program with finitely many (say m) threads, with the i^{th} thread executing the program S_i.

$$P = S_1 \,||\, \ldots \,||\, S_m$$

where each S_i is a sequential program executing finitely many (say n_i) D-operations.

$$S_i = op_1; op_2; \ldots; op_{n_i} \text{ where } \forall j \in [1, \ldots, n_i], op_j \in \{\pi_1, \ldots \pi_k\}$$

Lock Freedom and Termination

Definition

Let D be a shared-memory data structure with k-operations denoted by $\pi_1, \pi_2, \ldots, \pi_k$.

Let P be a concurrent program with finitely many (say m) threads, with the i^{th} thread executing the program S_i.

$$P = S_1 \,||\, \ldots \,||\, S_m$$

where each S_i is a sequential program executing finitely many (say n_i) D-operations.

$$S_i = op_1; op_2; \ldots; op_{n_i} \text{ where } \forall j \in [1, \ldots, n_i], op_j \in \{\pi_1, \ldots \pi_k\}$$

Theorem

The data-structure D with operations $\pi_1, \ldots, \pi_n$ is lock free iff every such program P terminates.

Lock-free data structure: An example

Example

Let A be a heap location of type `Int`, shared between a a number of producer and consumer threads.

A producer checks if A is 0, and if so, it updates A with a newly produced non-zero value and terminates.

A consumer checks if A contains a non-zero value, and if so, consumes the value, sets the value of A to 0 and loops to check if A contains a new value to consume. If A contains 0 then it terminates.

We want to prove that if a consumer does not terminate then it is busy performing some useful work, i.e, consuming the data-produced by the producer.

Producer-Consumer Code

```
producer(int y):
    atomic ⟨
        if ([A] == 0):
            [A] = y;
        else :
            skip; ⟩
```

```
consumer():
    Int x = 1;

    while (x ≠ 0):
        atomic ⟨
            b = [A];
            if (b ≠ 0):
                x = b;
                [A] = 0;
            else :
                x = 0; ⟩
```

Lock Freedom: Observation

Informal reasoning about lock-freedom

In an operation of a lock-free data-structure, the failure of a thread to make progress is always caused by successful progress in an operation executed by another thread.

A thread which fails to make progress, typically retries the operation.

In concurrent execution of finitely many threads, each performing finitely many operations of a lock-free data structure, one can precompute the upper bound on the number of retries that each thread can perform.

Example

If m_c consumer threads and m_p producer threads were running concurrently, then the total number of loop iterations across all the consumer threads is at most $m_c + m_p$.

Introducing Quantitative reasoning

Definition (Affine Resource)

An affine resource is one which once consumed cannot be regenerated.

Introducing Quantitative reasoning

Definition (Affine Resource)

An affine resource is one which once consumed cannot be regenerated.

Quantitative Reasoning

Each thread begins with a finite number of tokens which are affine resources.

Each time a thread wants to *try* performing the operation, it pays the price of one token which gets consumed.

When a thread's operation succeeds, it doesn't need to retry. Hence it can *compensate* for the failure of other threads by *transferring* the remaining tokens to the other threads which failed to make progress.

When a thread's operation fails, it is compensated by the thread which makes progress and can thus pay for the subsequent retry.

Introducing Quantitative reasoning

Definition (Affine Resource)

An affine resource is one which once consumed cannot be regenerated.

Quantitative Reasoning

The total number of tokens the system begins with provides the upper bound on the number of retries.

Producer-Consumer Code

```
producer(int y):
// Tokens available= {•}
    atomic ⟨

        if ([A] == 0):

            [A] = y;

        else :

            skip;
                            ⟩
```

```
consumer():

    Int x = 1;

    while (x ≠ 0):

        atomic ⟨
            b = [A];

            if (b ≠ 0):

                x = b;
                [A] = 0;

            else :
                x = 0;
                            ⟩
```

Producer-Consumer Code

```
producer(int y):
    atomic ⟨
        // Tokens available= {•}
        if ([A] == 0):
            [A] = y;
        else :
            skip;
                                ⟩
```

```
consumer():
    Int x = 1;
    while (x ≠ 0):
        atomic ⟨
            b = [A];
            if (b ≠ 0):
                x = b;
                [A] = 0;
            else :
                x = 0;
                                ⟩
```

Producer-Consumer Code

```
producer(int y):

    atomic ⟨

        if ([A] == 0):
            // Tokens available= {●}
            [A] = y;

        else :

            skip;
                                ⟩
```

```
consumer():

    Int x = 1;

    while (x ≠ 0):

        atomic ⟨
            b = [A];

            if (b ≠ 0):

                x = b;
                [A] = 0;

            else :
                x = 0;
                                ⟩
```

Producer-Consumer Code

```
producer(int y):

    atomic ⟨

        if ([A] == 0):

            [A] = y;
            // Tokens available= ∅
        else :

            skip;
                                    ⟩

```

Tokens for compensation = $\{\bullet\}$

```
consumer():

    Int x = 1;

    while (x ≠ 0):

        atomic ⟨
            b = [A];

            if (b ≠ 0):

                x = b;
                [A] = 0;

            else :
                x = 0;
                                    ⟩


```

Producer-Consumer Code

```
producer(int y):
    atomic ⟨
        // Tokens available= {●}
        if ([A] == 0):
            [A] = y;
        else :
            skip;
                                    ⟩
```

```
consumer():
    Int x = 1;
    while (x ≠ 0):
        atomic ⟨
            b = [A];
            if (b ≠ 0):
                x = b;
                [A] = 0;
            else :
                x = 0;
                                    ⟩
```

Producer-Consumer Code

```
producer(int y):

    atomic ⟨

        if ([A] == 0):

            [A] = y;

        else :
            // Tokens available= {•}
            skip;
                                  ⟩

```

```
consumer():

    Int x = 1;

    while (x ≠ 0):

        atomic ⟨
            b = [A];

            if (b ≠ 0):

                x = b;
                [A] = 0;

            else :
                x = 0;
                                  ⟩


```

Producer-Consumer Code

```
producer(int y):
    atomic ⟨
        if ([A] == 0):
            [A] = y;
        else :
            skip;
            // Tokens available= ∅ ⟩
```

Tokens for compensation = $\{\bullet\}$

```
consumer():
    Int x = 1;
    while (x ≠ 0):
        atomic ⟨
            b = [A];
            if (b ≠ 0):
                x = b;
                [A] = 0;
            else :
                x = 0;
                                    ⟩
```

Producer-Consumer Code

```
producer(int y):
    atomic ⟨
        if ([A] == 0):
            [A] = y;
        else :
            skip;
                                ⟩
    // Tokens available = ∅
```

```
consumer():
    Int x = 1;
    while (x ≠ 0):
        atomic ⟨
            b = [A];
            if (b ≠ 0):
                x = b;
                [A] = 0;
            else :
                x = 0;
                                ⟩
```

Producer-Consumer Code

```
producer(int y):

    atomic ⟨

        if ([A] == 0):

            [A] = y;

        else :

            skip;
                                ⟩
```

```
consumer():
// Tokens available = {•}
    Int x = 1;

    while (x ≠ 0):

        atomic ⟨
            b = [A];

            if (b ≠ 0):

                x = b;
                [A] = 0;

            else :
                x = 0;
                                ⟩
```

Producer-Consumer Code

```
producer(int y):

    atomic ⟨

        if ([A] == 0):

            [A] = y;

        else :

            skip;
                                ⟩

```

```
consumer():

    Int x = 1;
    // Tokens available = {•}
    while (x ≠ 0):

        atomic ⟨
            b = [A];

            if (b ≠ 0):

                x = b;
                [A] = 0;

            else :
                x = 0;
                                ⟩


```

Producer-Consumer Code

```
producer(int y):

    atomic ⟨

        if ([A] == 0):

            [A] = y;

        else :

            skip;
                                ⟩
```

```
consumer():

    Int x = 1;

    while (x ≠ 0):
        // Tokens available = ∅
        atomic ⟨
            b = [A];

            if (b ≠ 0):

                x = b;
                [A] = 0;

            else :
                x = 0;
                                ⟩
```

Producer-Consumer Code

```
producer(int y):
    atomic ⟨
        if ([A] == 0):
            [A] = y;
        else :
            skip;
                        ⟩
```

```
consumer():
    Int x = 1;
    while (x ≠ 0):
        atomic ⟨
            b = [A];
            // Tokens available = ∅
            if (b ≠ 0):
                x = b;
                [A] = 0;
            else :
                x = 0;
                        ⟩
```

Producer-Consumer Code

```
producer(int y):

    atomic ⟨

        if ([A] == 0):

            [A] = y;

        else :

            skip;
                                    ⟩
```

Tokens for compensation = $\{\bullet\}$

```
consumer():

    Int x = 1;

    while (x ≠ 0):

        atomic ⟨
            b = [A];

            if (b ≠ 0):
                // Tokens Available =∅
                x = b;
                [A] = 0;

            else :
                x = 0;
                                        ⟩
```

Producer-Consumer Code

```
producer(int y):

    atomic ⟨

        if ([A] == 0):

            [A] = y;

        else :

            skip;
                                ⟩
```

```
consumer():

    Int x = 1;

    while (x ≠ 0):

        atomic ⟨
            b = [A];

            if (b ≠ 0):

                x = b;
                [A] = 0;
                // Tokens available = {•}
            else :
                x = 0;
                                ⟩
```

Producer-Consumer Code

```
producer(int y):

    atomic ⟨

        if ([A] == 0):

            [A] = y;

        else :

            skip;
                                ⟩
```

```
consumer():

    Int x = 1;

    while (x ≠ 0):

        atomic ⟨
            b = [A];
            // Tokens available = ∅
            if (b ≠ 0):

                x = b;
                [A] = 0;

            else :
                x = 0;
                                    ⟩
```

Producer-Consumer Code

```
producer(int y):

    atomic ⟨

        if ([A] == 0):

            [A] = y;

        else :

            skip;
                                ⟩
```

```
consumer():

    Int x = 1;

    while (x ≠ 0):

        atomic ⟨
            b = [A];

            if (b ≠ 0):

                x = b;
                [A] = 0;

            else :
                x = 0;
                // Tokens available = ∅ ⟩
```

Producer-Consumer Code

```
producer(int y):

    atomic ⟨

        if ([A] == 0):

            [A] = y;

        else :

            skip;
                                  ⟩
```

```
consumer():

    Int x = 1;

    while (x ≠ 0):

        atomic ⟨
            b = [A];

            if (b ≠ 0):

                x = b;
                [A] = 0;

            else :
                x = 0;
                                  ⟩
        // x ≠ 0 ⟹ Tokens available = {•}
```

Producer-Consumer Code

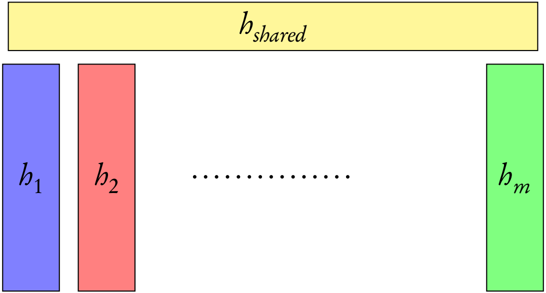
```
producer(int y):
    atomic ⟨
        if ([A] == 0):
            [A] = y;
        else :
            skip;
    ⟩
```

```
consumer():
    Int x = 1;
    while (x ≠ 0):
        atomic ⟨
            b = [A];
            if (b ≠ 0):
                x = b;
                [A] = 0;
            else :
                x = 0;
        ⟩
    // Tokens Available = ∅
```

Concurrent Separation Logic (CSL): A quick and dirty introduction

In a concurrent program of m threads, the memory is partitioned into disjoint portions $h_1, h_2, \ldots h_m$ and h_s where

- $\forall\, i \in [1, \ldots, m]$, h_i is the set of all memory locations accessible only to thread i called the private heap of i.
- h_{shared} is the remaining set of memory locations shared between the threads called the *shared heap*.



Concurrent Separation Logic (CSL): A quick and dirty introduction

Heaps are characterised using separation logic assertions.

$$P, Q ::= \mathit{true} \mid \mathit{emp} \mid [x] \mapsto y \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P * Q \mid \exists z.P \mid \forall z.P$$

For any heap h, $h \vDash [x] \mapsto y$ iff h is a single memory cell x which stores the value y.

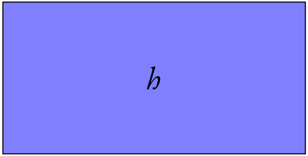
Concurrent Separation Logic (CSL): A quick and dirty introduction

Heaps are characterised using separation logic assertions.

$$P, Q ::= true \mid emp \mid [x] \mapsto y \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P * Q \mid \exists z.P \mid \forall z.P$$

For any heap h, $h \vDash [x] \mapsto y$ iff h is a single memory cell x which stores the value y.

Suppose P and Q are assertions, we say that a heap $h \vDash P * Q$ iff



$\vDash \; P * Q$

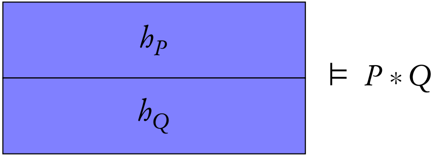
Concurrent Separation Logic (CSL): A quick and dirty introduction

Heaps are characterised using separation logic assertions.

$$P, Q ::= \mathit{true} \mid \mathit{emp} \mid [x] \mapsto y \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P * Q \mid \exists z.P \mid \forall z.P$$

For any heap h, $h \vDash [x] \mapsto y$ iff h is a single memory cell x which stores the value y.

Suppose P and Q are assertions, we say that a heap $h \vDash P * Q$ iff we can partition h into disjoint portions h_P and h_Q such that



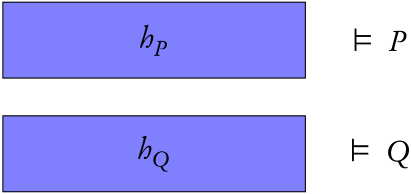
Concurrent Separation Logic (CSL): A quick and dirty introduction

Heaps are characterised using separation logic assertions.

$$P, Q ::= true \mid emp \mid [x] \mapsto y \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P * Q \mid \exists z.P \mid \forall z.P$$

For any heap h, $h \vDash [x] \mapsto y$ iff h is a single memory cell x which stores the value y.

Suppose P and Q are assertions, we say that a heap $h \vDash P * Q$ iff we can partition h into disjoint portions h_P and h_Q such that $h_P \vDash P$ and $h_Q \vDash Q$.



Concurrent Separation Logic (CSL): A quick and dirty introduction

Let I, P, Q denote separation logic assertions describing the heaps.

Concurrent Separation Logic judgement

The judgement $I \vdash [P]\ C\ [Q]$ is to be understood as follows:

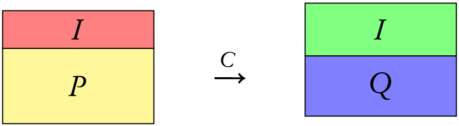
> A thread executing program C beginning with a private heap that satisfies P executes *safely* and *terminates* resulting in a private heap of the thread which satisfies Q.
>
> Throughout the execution of C (except inside the atomic sections), the shared heap satisfies I.

Concurrent Separation Logic (CSL): A quick and dirty introduction

Rule for parallel composition: PAR

$$\frac{I \vdash [P_1]\, C_1\, [Q_1] \quad \ldots \quad I \vdash [P_m]\, C_m\, [Q_m]}{I \vdash [P_1 * \cdots * P_m]\, C_1 \,||\, \cdots \,||\, C_m\, [Q_1 * \cdots * Q_m]}$$

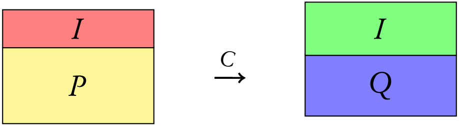
Concurrent Separation Logic (CSL): A quick and dirty introduction



Rule for Atomic sections: ATOM

$$\vdash [P * I]\, C\, [Q * I]$$

Concurrent Separation Logic (CSL): A quick and dirty introduction



Rule for Atomic sections: ATOM

$$\frac{\vdash [P * I]\, C\, [Q * I]}{I \vdash [P]\langle C\rangle[Q]}$$

Back to the Paper: Quantitative CSL

Let $\Diamond$ be a predicate such that for any heap h, $h \vDash \Diamond$ iff the heap h has at least one affine token.

We write $\Diamond^k$ as a shorthand for $\underbrace{\Diamond * \cdots * \Diamond}_{k \text{ times}}$.

Quantitative CSL

Rule for `while` loop in CSL:

$$\frac{I \vdash [P \wedge B]\ C\ [P]}{I \vdash [P]\ \texttt{while}(B)\ \texttt{do}\ \ C\ [P \wedge \neg Cond]}$$

Quantitative CSL

Rule for `while` loop in Quantitative CSL:

$$\frac{P \wedge B \implies P' * \Diamond \quad I \vdash [P']\, C\, [P]}{I \vdash [P]\ \texttt{while}(B)\ \texttt{do}\ C\ [P \wedge \neg B]}$$

Using Quantitative CSL to prove lock freedom of Producer-Consumer

Example

Setting $I := A \mapsto 0 \vee ((\exists u : u \neq 0 \wedge A \mapsto u) * \Diamond)$
Loop invariant $P := x = 0 \vee \Diamond$,
loop condition $B := x \neq 0$
and the use of ATOM rule, we can show that

$$I \vdash [\Diamond]\texttt{consumer()}[emp]$$

and

$$I \vdash [\Diamond]\texttt{producer()}[emp]$$

Using Quantitative CSL to prove lock freedom of Producer-Consumer

Example

If S_i is a sequential program invoking exactly n_i calls from $\{\texttt{producer()}, \texttt{consumer()}\}$ then by induction we can prove that

$$I \vdash [\Diamond^{n_i}]\, S_i\, [emp]$$

.

If P is a concurrent program $S_1 \,||\, S_2 \,||\, \cdots \,||\, S_m$ then by PAR rule we have

$$I \vdash [\Diamond^{n_{tot}}]\, P\, [emp]$$

where $n_{tot} = \Sigma_{i=0}^{m} n_i$.

This proves the termination of P.

Outline

1. Quantitative Reasoning for Proving Lock-Freedom - Jan Hoffman, Michael Marmar, Zhong Shao: In the proceedings of LICS 2013

2. Aspect-Oriented Linearizability Proofs: Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis: In the proceedings of CONCUR 2013

Contributions of this paper

Reduces the task of verifying linearizability of a queue implementation to establishing four basic properties each of which can be independently verified.

Demonstrates the linearizability of Herlihy-Wing queue using the proposed technique.

Uses RGSep, a combination of Rely-Guarantee Logic and Separation Logic to automate the verification of three of these four properties.

Linearizability

Suppose Q is a concurrent queue over the domain $Val = \mathbb{N} \cup \{\texttt{NULL}\}$ that supports two methods

- $\textsf{enq}(x : \mathbb{N})$ that enqueues the value x into the queue. Returns void.

  We denote an instance of this method call by $\langle \textsf{enq},\ x \rangle$.

  Each $\langle \textsf{enq},\ x \rangle$ method instance has an invocation event denoted by $\langle \textsf{enq},\ x \rangle_i$ and a response event denoted by $\langle \textsf{enq},\ x \rangle_r$.

- $\textsf{deq}(\texttt{void})$ which returns some value y from Val.

  We denote an instance of this method call by $\langle \textsf{deq},\ y \rangle$.

  Each $\langle \textsf{deq},\ y \rangle$ method instance has an invocation event denoted by $\langle \textsf{deq},\ y \rangle_i$ and a response event denoted by $\langle \textsf{deq},\ y \rangle_r$.

Linearizability

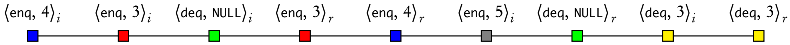
Definition (History)

A history c*, is a sequence of invocation and response events where every response event has a corresponding invocation event that appears before it in the sequence.*

Linearizability

Definition (History)

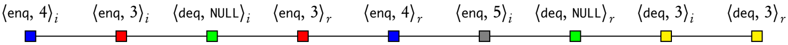
A history c*, is a sequence of invocation and response events where every response event has a corresponding invocation event that appears before it in the sequence.*

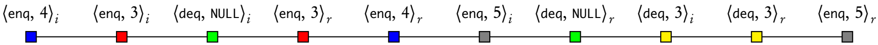


Linearizability

Definition (History)

A history c, is a sequence of invocation and response events where every response event has a corresponding invocation event that appears before it in the sequence.





Note

In a history c not every invocation events needs to have a corresponding response event. Such histories are called *incomplete histories*. Eg. $\langle \text{enq}, 5\rangle_i$

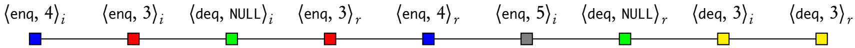
An incomplete history c can be *completed* by appending the response events for the unmatched invocation events to obtain it's completion $\hat{c}$.

There could be several completions of an incomplete history.

Linearizability

Definition (History)

A history c*, is a sequence of invocation and response events where every response event has a corresponding invocation event that appears before it in the sequence.*



Definition (Happened Before)

Let c *be a history and* $<_c$ *the total order on the set of events in* c.

We say that the method call m happened-before *a method call* m' *in* c*, denoted by* $m \xrightarrow{\text{hb}}_c m'$ *iff* $m_r <_c m'_i$.

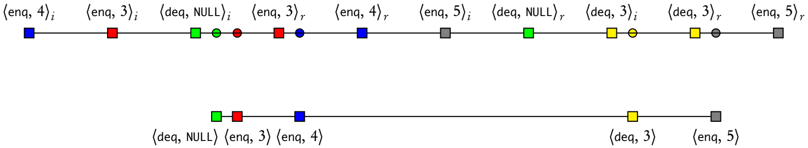
Eg: $\langle \text{enq}, 4\rangle \xrightarrow{\text{hb}}_c \langle \text{deq}, 3\rangle$.

Linearizability

Definition (Linearlizability)

A history c is said to be linearizable *iff there exists some completion $\hat{c}$ of c in which*

> *For every method m there is a* linearization point *at some instant between m_i and m_r.*
>
> *All methods appear to occur instantly at their linearization point, behaving as specified by the sequential specification.*



Linearizability

Definition

The set of histories C of concurrent queue implementation is linearizable iff all the concurrent histories $c \in C$ are linearizable.

Linearizability

Definition

The set of histories C of concurrent queue implementation is linearizable iff all the concurrent histories $c \in C$ are linearizable.

Proving Linearizability of a concurrent queue implementation

The most common technique to prove the linearizability of a queue implementation is to identify a point inside the code of **enq** and **deq** as the linearization points.

Linearizability

Definition

The set of histories C of concurrent queue implementation is linearizable iff all the concurrent histories $c \in C$ are linearizable.

Proving Linearizability of a concurrent queue implementation

The most common technique to prove the linearizability of a queue implementation is to identify a point inside the code of **enq** and **deq** as the linearization points.

However, this technique doesn't lend itself to proving linearizability of several concurrent queue implementations. Eg: Herlihy-Wing queue.

Proving Linearizability

```
int q.back = 0
Val q.items[] = {NULL, NULL, ...}

void enq(int x):
    int i;
    atomic ⟨
        i = q.back;
        q.back++; ⟩// E_1

    atomic ⟨
        q.items[i] = x ⟩// E_2
```

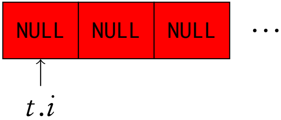
```
Val deq():
    int i, range;
    Val x;

    while (true):
        atomic ⟨
            range = q.back − 1; ⟩// D_1

        for i from 0 to range:
            atomic ⟨
                x = q.items[i]
                q.items[i] = NULL; ⟩// D_2

            if (x ≠ NULL) return x;
```

Proving Linearizability

Let t, u, v, w be four concurrent threads. Let $\circ$ denote context switch.

Consider the execution fragment:

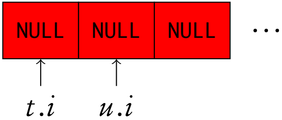
$$c = (t : E_1)$$



Proving Linearizability

Let t, u, v, w be four concurrent threads. Let $\circ$ denote context switch.
Consider the execution fragment:

$$c = (t : E_1) \circ (u : E_1)$$

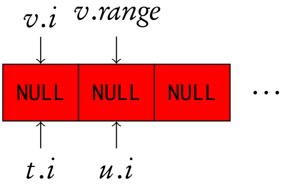


Proving Linearizability

Let t, u, v, w be four concurrent threads. Let $\circ$ denote context switch.

Consider the execution fragment:

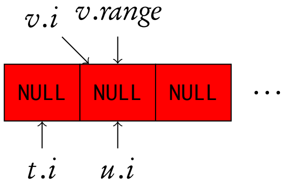
$$c = (t : E_1) \circ (u : E_1) \circ (v : D_1$$



Proving Linearizability

Let t, u, v, w be four concurrent threads. Let $\circ$ denote context switch.
Consider the execution fragment:

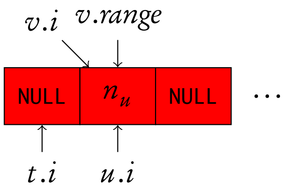
$$c = (t : E_1) \circ (u : E_1) \circ (v : D_1, D_2)$$



Proving Linearizability

Let t, u, v, w be four concurrent threads. Let $\circ$ denote context switch.
Consider the execution fragment:

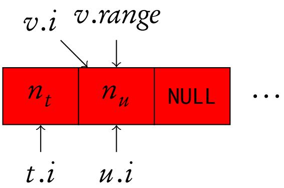
$$c = (t : E_1) \circ (u : E_1) \circ (v : D_1, D_2) \circ (u : E_2)$$



Proving Linearizability

Let t, u, v, w be four concurrent threads. Let $\circ$ denote context switch.
Consider the execution fragment:

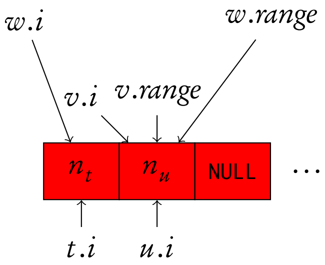
$$c = (t : E_1) \circ (u : E_1) \circ (v : D_1, D_2) \circ (u : E_2) \circ (t : E_2)$$



Proving Linearizability

Let t, u, v, w be four concurrent threads. Let $\circ$ denote context switch.
Consider the execution fragment:

$$c = (t : E_1) \circ (u : E_1) \circ (v : D_1, D_2) \circ (u : E_2) \circ (t : E_2) \circ (w : D_1)$$



Proving Linearizability

Let t, u, v, w be four concurrent threads. Let $\circ$ denote context switch.

Consider the execution fragment:

$$c = (t : E_1) \circ (u : E_1) \circ (v : D_1, D_2) \circ (u : E_2) \circ (t : E_2) \circ (w : D_1)$$

At the end of this execution fragment,

- t has enqueued an item in $q.items[0]$.
- u has enqueued an item in $q.items[1]$.
- v is ready to dequeue the value enqueued by u.
- w is ready to dequeue the value enqueued by t.

Proving Linearizability

Let t, u, v, w be four concurrent threads. Let $\circ$ denote context switch.
Consider the execution fragment:

$$c = (t : E_1) \circ (u : E_1) \circ (v : D_1, D_2) \circ (u : E_2) \circ (t : E_2) \circ (w : D_1)$$

Suppose we choose E_1 in **enq** to be the linearization point for t then the following extension of c is not linearizable via these linearization point.

$$(v : D_2, return) \circ (z : D_1, D_2, return)$$

Proving Linearizability

Let t, u, v, w be four concurrent threads. Let $\circ$ denote context switch.
Consider the execution fragment:

$$c = (t : E_1) \circ (u : E_1) \circ (v : D_1, D_2) \circ (u : E_2) \circ (t : E_2) \circ (w : D_1)$$

Suppose we choose E_1 in **enq** to be the linearization point for t then the following extension of c is not linearizable via these linearization point.

$$(v : D_2, return) \circ (z : D_1, D_2, return)$$

$t : \langle \textsf{enq},\ n_t \rangle$ takes effect before $u : \langle \textsf{enq},\ n_u \rangle$
$v : \langle \textsf{deq},\ n_u \rangle$ takes effect before $z : \langle \textsf{deq},\ n_t \rangle$.

Proving Linearizability

Let t, u, v, w be four concurrent threads. Let $\circ$ denote context switch.
Consider the execution fragment:

$$c = (t : E_1) \circ (u : E_1) \circ (v : D_1, D_2) \circ (u : E_2) \circ (t : E_2) \circ (w : D_1)$$

Similarly if we choose E_2 in **enq** to be the linearization point for t then we have the following extension of c which is not linearizable via this linearization point.

$$(w : D_2, return) \circ (z : D_1, D_2, D_2, return)$$

Proving Linearizability

Let t, u, v, w be four concurrent threads. Let $\circ$ denote context switch.
Consider the execution fragment:

$$c = (t : E_1) \circ (u : E_1) \circ (v : D_1, D_2) \circ (u : E_2) \circ (t : E_2) \circ (w : D_1)$$

Similarly if we choose E_2 in **enq** to be the linearization point for t then we have the following extension of c which is not linearizable via this linearization point.

$$(w : D_2, return) \circ (z : D_1, D_2, D_2, return)$$

$u : \langle \textbf{enq}, n_u \rangle$ takes effect before $t : \langle \textbf{enq}, n_t \rangle$
$v : \langle \textbf{deq}, n_t \rangle$ takes effect before $z : \langle \textbf{deq}, n_u \rangle$.

Aspect oriented Linearizability Proof

Intuitively, a *correct* concurrent history of a queue implementation should not have any of the four violations.

(**VFresh**): A dequeue event returning a value not inserted by any enqueue event.

(**VRepeat**): Two dequeue events returning the value inserted by the same enqueue event.

(**Vord**): Two ordered dequeue events returning values inserted by ordered enqueue events in the inverse order.

(**VWit**): A dequeue event returning `NULL` even though the queue is never logically empty during the execution of the dequeue event.

Aspect oriented Linearizability Proof

Intuitively, a *correct* concurrent history of a queue implementation should not have any of the four violations.

(**VFresh**): A dequeue event returning a value not inserted by any enqueue event.

(**VRepeat**): Two dequeue events returning the value inserted by the same enqueue event.

(**Vord**): Two ordered dequeue events returning values inserted by ordered enqueue events in the inverse order.

(**VWit**): A dequeue event returning `NULL` even though the queue is never logically empty during the execution of the dequeue event.

Theorem

A set of histories C of a concurrent queue is linearizable iff for every $c \in C$ there exists a completion $\hat{c}$ that has none of the **VFresh**, **VRepeat**, **Vord** , **VWit** *violations.*

Few remarks

Consider the following history

$$\hat{c} = \langle \mathsf{enq},\ 1\rangle_i \cdot \langle \mathsf{enq},\ 1\rangle_r \cdot \langle \mathsf{enq},\ 2\rangle_i \cdot \langle \mathsf{enq},\ 2\rangle_r \cdot \langle \mathsf{deq},\ 2\rangle_i \cdot \langle \mathsf{deq},\ 2\rangle_r \cdot \langle \mathsf{enq},\ 3\rangle_i \cdot \langle \mathsf{enq},\ 3\rangle_r$$

Few remarks

Consider the following history

$$\hat{c} = \langle \mathsf{enq},\ 1\rangle_i \cdot \langle \mathsf{enq},\ 1\rangle_r \cdot \langle \mathsf{enq},\ 2\rangle_i \cdot \langle \mathsf{enq},\ 2\rangle_r \cdot \langle \mathsf{deq},\ 2\rangle_i \cdot \langle \mathsf{deq},\ 2\rangle_r \cdot \langle \mathsf{enq},\ 3\rangle_i \cdot \langle \mathsf{enq},\ 3\rangle_r$$

Since it is a sequential history, we can rewrite it as

$$\hat{c} = \langle \mathsf{enq},\ 1\rangle \cdot \langle \mathsf{enq},\ 2\rangle \cdot \langle \mathsf{deq},\ 2\rangle \cdot \langle \mathsf{enq},\ 3\rangle$$

Few remarks

Consider the following history

$$\hat{c} = \langle \mathsf{enq},\ 1\rangle_i \cdot \langle \mathsf{enq},\ 1\rangle_r \cdot \langle \mathsf{enq},\ 2\rangle_i \cdot \langle \mathsf{enq},\ 2\rangle_r \cdot \langle \mathsf{deq},\ 2\rangle_i \cdot \langle \mathsf{deq},\ 2\rangle_r \cdot \langle \mathsf{enq},\ 3\rangle_i \cdot \langle \mathsf{enq},\ 3\rangle_r$$

Since it is a sequential history, we can rewrite it as

$$\hat{c} = \langle \mathsf{enq},\ 1\rangle \cdot \langle \mathsf{enq},\ 2\rangle \cdot \langle \mathsf{deq},\ 2\rangle \cdot \langle \mathsf{enq},\ 3\rangle$$

One may verify that it is a complete history and has none of the four violations.

Few remarks

Consider the following history

$$\hat{c} = \langle \text{enq}, 1\rangle_i \cdot \langle \text{enq}, 1\rangle_r \cdot \langle \text{enq}, 2\rangle_i \cdot \langle \text{enq}, 2\rangle_r \cdot \langle \text{deq}, 2\rangle_i \cdot \langle \text{deq}, 2\rangle_r \cdot \langle \text{enq}, 3\rangle_i \cdot \langle \text{enq}, 3\rangle_r$$

Since it is a sequential history, we can rewrite it as

$$\hat{c} = \langle \text{enq}, 1\rangle \cdot \langle \text{enq}, 2\rangle \cdot \langle \text{deq}, 2\rangle \cdot \langle \text{enq}, 3\rangle$$

One may verify that it is a complete history and has none of the four violations.

Yet, it is not a *correct* history as per the sequential specification.

Few remarks

Consider the following history

$$\hat{c} = \langle \text{enq},\ 1\rangle_i \cdot \langle \text{enq},\ 1\rangle_r \cdot \langle \text{enq},\ 2\rangle_i \cdot \langle \text{enq},\ 2\rangle_r \cdot \langle \text{deq},\ 2\rangle_i \cdot \langle \text{deq},\ 2\rangle_r \cdot \langle \text{enq},\ 3\rangle_i \cdot \langle \text{enq},\ 3\rangle_r$$

Since it is a sequential history, we can rewrite it as

$$\hat{c} = \langle \text{enq},\ 1\rangle \cdot \langle \text{enq},\ 2\rangle \cdot \langle \text{deq},\ 2\rangle \cdot \langle \text{enq},\ 3\rangle$$

One may verify that it is a complete history and has none of the four violations.

Yet, it is not a *correct* history as per the sequential specification.

Observation

For any complete history $\hat{c} \in C$, for any finite k, there exists values $v_1, \ldots, v_k \in \mathbb{N} \cup \{\text{NULL}\}$ such that the extension $\hat{c} \cdot \langle \text{deq},\ v_1\rangle_i \cdot \langle \text{deq},\ v_1\rangle_r \cdot \cdots \cdot \langle \text{deq},\ v_k\rangle_i \cdot \langle \text{deq},\ v_k\rangle_r \in C$.

Few remarks

Consider the following sequential history of a queue.

$$\hat{c}_{ext} = \langle \mathsf{enq},\ 1 \rangle \cdot \langle \mathsf{enq},\ 2 \rangle \cdot \langle \mathsf{deq},\ 2 \rangle \cdot \langle \mathsf{enq},\ 3 \rangle \cdot \langle \mathsf{deq},\ v_1 \rangle \cdot \langle \mathsf{deq},\ v_2 \rangle$$

Few remarks

Consider the following sequential history of a queue.

$$\hat{c}_{ext} = \langle \mathsf{enq},\ 1 \rangle \cdot \langle \mathsf{enq},\ 2 \rangle \cdot \langle \mathsf{deq},\ 2 \rangle \cdot \langle \mathsf{enq},\ 3 \rangle \cdot \langle \mathsf{deq},\ v_1 \rangle \cdot \langle \mathsf{deq},\ v_2 \rangle$$

If $v_1 \neq 3$ then $\hat{c}_{ext}$ is going to violate one of the four violations.

- If $v_1 \notin \{1, 2, 3, \texttt{NULL}\}$, $\hat{c}_{ext}$ violates **VFresh**.
- If $v_1 = 2$, $\hat{c}_{ext}$ violates **VRepeat**.
- If $v_1 = 1$, $\hat{c}_{ext}$ violates **VOrd**.
- If $v_1 = \texttt{NULL}$, $\hat{c}_{ext}$ violates **VWit**.

Few remarks

Consider the following sequential history of a queue.

$$\hat{c}_{ext} = \langle \mathsf{enq},\ 1\rangle \cdot \langle \mathsf{enq},\ 2\rangle \cdot \langle \mathsf{deq},\ 2\rangle \cdot \langle \mathsf{enq},\ 3\rangle \cdot \langle \mathsf{deq},\ 3\rangle \cdot \langle \mathsf{deq},\ v_2\rangle$$

We cannot assign any value to v_2 without violating one of the four properties.

Few remarks

Consider the following sequential history of a queue.

$$\hat{c}_{ext} = \langle \text{enq},\ 1\rangle \cdot \langle \text{enq},\ 2\rangle \cdot \langle \text{deq},\ 2\rangle \cdot \langle \text{enq},\ 3\rangle \cdot \langle \text{deq},\ 3\rangle \cdot \langle \text{deq},\ v_2\rangle$$

We cannot assign any value to v_2 without violating one of the four properties.

If $v_2 \notin \{1, 2, 3, \texttt{NULL}\}$, $\hat{c}_{ext}$ violates **VFresh**.

Few remarks

Consider the following sequential history of a queue.

$$\hat{c}_{ext} = \langle \mathsf{enq},\ 1\rangle \cdot \langle \mathsf{enq},\ 2\rangle \cdot \langle \mathsf{deq},\ 2\rangle \cdot \langle \mathsf{enq},\ 3\rangle \cdot \langle \mathsf{deq},\ 3\rangle \cdot \langle \mathsf{deq},\ v_2\rangle$$

We cannot assign any value to v_2 without violating one of the four properties.

If $v_2 \notin \{1, 2, 3, \mathtt{NULL}\}$, $\hat{c}_{ext}$ violates **VFresh**.

If $v_2 \in \{2, 3\}$, $\hat{c}_{ext}$ violates **VRepeat**.

Few remarks

Consider the following sequential history of a queue.

$$\hat{c}_{ext} = \langle \text{enq},\ 1\rangle \cdot \langle \text{enq},\ 2\rangle \cdot \langle \text{deq},\ 2\rangle \cdot \langle \text{enq},\ 3\rangle \cdot \langle \text{deq},\ 3\rangle \cdot \langle \text{deq},\ v_2\rangle$$

We cannot assign any value to v_2 without violating one of the four properties.

If $v_2 \notin \{1, 2, 3, \texttt{NULL}\}$, $\hat{c}_{ext}$ violates **VFresh**.

If $v_2 \in \{2, 3\}$, $\hat{c}_{ext}$ violates **VRepeat**.

If $v_2 = 1$, $\hat{c}_{ext}$ violates **VOrd**.

Few remarks

Consider the following sequential history of a queue.

$$\hat{c}_{ext} = \langle \text{enq},\ 1\rangle \cdot \langle \text{enq},\ 2\rangle \cdot \langle \text{deq},\ 2\rangle \cdot \langle \text{enq},\ 3\rangle \cdot \langle \text{deq},\ 3\rangle \cdot \langle \text{deq},\ v_2\rangle$$

We cannot assign any value to v_2 without violating one of the four properties.

If $v_2 \notin \{1, 2, 3, \texttt{NULL}\}$, $\hat{c}_{ext}$ violates **VFresh**.

If $v_2 \in \{2, 3\}$, $\hat{c}_{ext}$ violates **VRepeat**.

If $v_2 = 1$, $\hat{c}_{ext}$ violates **VOrd**.

If $v_2 = \texttt{NULL}$, $\hat{c}_{ext}$ violates **VWit**.

Few remarks

Consider the following sequential history of a queue.

$$\hat{c}_{ext} = \langle \text{enq},\ 1\rangle \cdot \langle \text{enq},\ 2\rangle \cdot \langle \text{deq},\ 2\rangle \cdot \langle \text{enq},\ 3\rangle \cdot \langle \text{deq},\ 3\rangle \cdot \langle \text{deq},\ v_2\rangle$$

We cannot assign any value to v_2 without violating one of the four properties.

If $v_2 \notin \{1, 2, 3, \texttt{NULL}\}$, $\hat{c}_{ext}$ violates **VFresh**.

If $v_2 \in \{2, 3\}$, $\hat{c}_{ext}$ violates **VRepeat**.

If $v_2 = 1$, $\hat{c}_{ext}$ violates **VOrd**.

If $v_2 = \texttt{NULL}$, $\hat{c}_{ext}$ violates **VWit**.

Few remarks

Consider the following sequential history of a queue.

$$\hat{c}_{ext} = \langle \text{enq},\ 1\rangle \cdot \langle \text{enq},\ 2\rangle \cdot \langle \text{deq},\ 2\rangle \cdot \langle \text{enq},\ 3\rangle \cdot \langle \text{deq},\ 3\rangle \cdot \langle \text{deq},\ v_2\rangle$$

We cannot assign any value to v_2 without violating one of the four properties.

- If $v_2 \notin \{1, 2, 3, \texttt{NULL}\}$, $\hat{c}_{ext}$ violates **VFresh**.
- If $v_2 \in \{2, 3\}$, $\hat{c}_{ext}$ violates **VRepeat**.
- If $v_2 = 1$, $\hat{c}_{ext}$ violates **VOrd**.
- If $v_2 = \texttt{NULL}$, $\hat{c}_{ext}$ violates **VWit**.

Since the complete history $\hat{c}_{ext} \in C$ and it has at least one of these violations, by the theorem, C is not linearizable.

Thank You!