Quantitative Reasoning for Proving Lock-Freedom

Jan Hoffmann, Michael Marmar, and Zhong Shao
Quantitative Reasoning for Proving Lock-Freedom

Jan Hoffmann, Michael Marmar, and Zhong Shao

Mike is at LICS too.
Quantitative Reasoning for Proving Lock-Freedom

Jan Hoffmann, Michael Marmar, and Zhong Shao
Threads

Shared Memory

Concurrent Data Structures  |  Multiprocessor OS Kernel
Threads

Shared Memory

Concurrent Data Structures | Multiprocessor OS Kernel
Threads

Shared Memory

Concurrent Data Structures

Multiprocessor OS Kernel
Threads

Shared Memory

Concurrent Data Structures | Multiprocessor OS Kernel
Threads

Shared Memory

Concurrent Data Structures  Multiprocessor OS Kernel
Threads

Shared Memory

Need synchronization to avoid race conditions.

Concurrent Data Structures
Non-Blocking Synchronization
Non-Blocking Synchronization

• **Classical Synchronization:** locks ensure mutual exclusion of threads

⇒ Performance issues on modern multiprocessor architectures
  • Blocking (busy waiting)
  • Cache-coherency (high memory contention)
Non-Blocking Synchronization

• **Classical Synchronization:** locks ensure mutual exclusion of threads
  ➡ Performance issues on modern multiprocessor architectures
    • Blocking (busy waiting)
    • Cache-coherency (high memory contention)

• **Non-Blocking Synchronization:** shared data is accessed without locks
  ➡ Outperforms lock-based synchronization in many scenarios
    • Interference of threads possible
    • Need to ensure consistency of the data structure
How to Ensure Consistency Without Locks?
How to Ensure Consistency Without Locks?

- Attempt to perform an operation
- Repeat operations after interference has been detected
How to Ensure Consistency Without Locks?

• Attempt to perform an operation

• Repeat operations after interference has been detected

Optimistic synchronization.
How to Ensure Consistency Without Locks?

• Attempt to perform an operation

• Repeat operations after interference has been detected

• Ensure that a concurrent execution is equivalent to some sequential execution

• Desired properties: linearizability or serializability
  
  ▸ Different program logics exist, e.g., in [Fu et al. 2010]
  
  ▸ Contextual refinement, e.g., in [Liang et al. 2013]
How to Ensure Consistency Without Locks?

• Attempt to perform an operation

• Repeat operations after interference has been detected

• Ensure that a concurrent execution is equivalent to some sequential execution

• Desired properties: linearizability or serializability
  ‣ Different program logics exist, e.g., in [Fu et al. 2010]
  ‣ Contextual refinement, e.g., in [Liang et al. 2013]
How to Ensure Consistency Without Locks?

• Attempt to perform an operation

• Repeat operations after interference has been detected

• Ensure that a concurrent execution is equivalent to some sequential execution

• Desired properties: linearizability or serializability
  - Different program logics exist, e.g., in [Fu et al. 2010]
  - Contextual refinement, e.g., in [Liang et al. 2013]

• But: We also need additional progress guarantees.
Sequential Consistency is Not Enough

Threads

Shared Memory

Sequential Consistency is Not Enough

Livelocks
Sequential Consistency is Not Enough

Livelocks

Threads

Interference

Shared Memory

Sequential Consistency is Not Enough | Livelocks
Sequential Consistency is Not Enough

Data structure is consistent but system is stuck (livelock).

Threads

Shared Memory

Sequential Consistency is Not Enough
Progress Properties

Let $D$ be a shared-memory data structure with operations $\pi_1, \ldots, \pi_k$

- Assume a system with $m$ threads that access $D$ exclusively via the operations $\pi_1, \ldots, \pi_k$

- Assume that all code outside the data structure operations $\pi_i$ terminates

- Fix an arbitrary scheduling of the $m$ threads in which one or more operations $\pi_i$ have been started
Progress Properties

• A **wait-free** implementation guarantees that every thread can complete any started operation of the data structure in a finite number of steps

• A **lock-free** implementation guarantees that some thread will complete an operation in a finite number of steps

• An **obstruction-free** implementation guarantees the completion of an operation for any thread that eventually executes in isolation

• Wait-freedom implies lock-freedom

• Lock-freedom implies obstruction-freedom
Progress Properties

- A **wait-free** implementation guarantees that every thread can complete any started operation of the data structure in a finite number of steps.

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

- An **obstruction-free** implementation guarantees the completion of an operation for any thread that eventually executes in isolation.

- Wait-freedom implies lock-freedom.

- Lock-freedom implies obstruction-freedom.

No livelocks and no starvation.
Progress Properties

• A **wait-free** implementation guarantees that every thread can complete any started operation of the data structure in a finite number of steps.

• A **lock-free** implementation guarantees that some thread will complete an operation in a finite number of steps.

• An **obstruction-free** implementation guarantees the completion of an operation for any thread that eventually executes in isolation.

• Wait-freedom implies lock-freedom.

• Lock-freedom implies obstruction-freedom.

No livelocks and no starvation.
Progress Properties

- A **wait-free** implementation guarantees that every thread can complete any started operation of the data structure in a finite number of steps.

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

- An **obstruction-free** implementation guarantees the completion of an operation for any thread that eventually executes in isolation.

- Wait-freedom implies lock-freedom.

- Lock-freedom implies obstruction-freedom.

Our Results

New quantitative technique to verify lock-freedom

• Uses quant. compensation schemes to pay for possible interference

• Enables local and modular reasoning

• Our paper: Formalization based on concurrent separation logic [O’Hearn 2007] and quantitative separation logic [Atkey 2010]

• Running example: Treiber’s non-blocking stack (a classic lock-free data structure)
Our Results

New quantitative technique to verify lock-freedom

• Uses quant. compensation schemes to pay for possible interference

• Enables local and modular reasoning

• Our paper: Formalization based on concurrent separation logic [O’Hearn 2007] and quantitative separation logic [Atkey 2010]

• Running example: Treiber’s non-blocking stack (a classic lock-free data structure)

Sweet spot: strong progress guaranty and efficient, elegant implementations.
Our Results

New quantitative technique to verify lock-freedom

• Uses quant. compensation schemes to pay for possible interference

• Enables local and modular reasoning

• Our paper: Formalization based on concurrent separation logic [O’Hearn 2007] and quantitative separation logic [Atkey 2010]

• Running example: Treiber’s non-blocking stack (a classic lock-free data structure)

Sweet spot: strong progress guaranty and efficient, elegant implementations.

Classically: temporal logic and whole program analysis.
Treiber’s Non-Blocking Stack

```c
struct Node {
    value_t data;
    Node *next;
};

Node *S;

void init() { S = NULL; }
```
Treiber’s Non-Blocking Stack

```c
struct Node {
    value_t data;
    Node *next;
};

Node *S;

void init() { S = NULL; }
```

Stack is a linked list.
Treiber’s Non-Blocking Stack

```c
struct Node {
    value_t data;
    Node *next;
};

Node *S;

void init() { S = NULL; }
```

Stack is a linked list.

Shared pointer to the top element.
Treiber’s Non-Blocking Stack

struct Node {
    value_t data;
    Node *next;
};

Node *S;

void init() { S = NULL; }

void push(value_t v) {
    Node *t, *x;
    x = new Node();
    x->data = v;
    do { t = S;
        x->next = t;
    } while(!CAS(&S,t,x));
}
Treiber’s Non-Blocking Stack

```c
struct Node {
    value_t data;
    Node *next;
};

Node *S;

void init() { S = NULL; }

void push(value_t v) {
    Node *t, *x;
    x = new Node();
    x->data = v;
    do {
        t = S;
        x->next = t;
    } while(!CAS(&S,t,x));
}
```

Prepare update.
Treiber’s Non-Blocking Stack

```c
struct Node {
    value_t data;
    Node *next;
};

Node *S;

void init() { S = NULL; }
```

```c
void push(value_t v) {
    Node *t, *x;
    x = new Node();
    x->data = v;
    do { t = S;
        x->next = t;
    } while(!CAS(&S,t,x));
}
```

Prepare update.

Compare-and-swap operation: if the address &S contains t then write x into &S and return true; else return false.
Treiber’s Non-Blocking Stack

```c
struct Node {
    value_t data;
    Node *next;
};
Node *S;
void init() { S = NULL; }
void push(value_t v) {
    Node *t, *x;
    x = new Node();
    x->data = v;
    do { t = S;
        x->next = t;
        } while(!CAS(&S,t,x));
}
```

CAS-guarded while-loop to detect interference.

```
Prepare update.

Compare-and-swap operation: if the address &S contains t then write x into &S and return true; else return false.
```

This example is based on an idea from James Aspnes.
Treiber’s Non-Blocking Stack

```c
struct Node {
    value_t data;
    Node *next;
};

Node *S;

void init() { S = NULL; }

value_t pop() {
    Node *t, *x;
    do { t = S;
        if (t == NULL) { return EMPTY; }
        x = t->next;
    } while(!CAS(&S,t,x));
    return t->data;
}

void push(value_t v) {
    Node *t, *x;
    x = new Node();
    x->data = v;
    do { t = S;
        x->next = t;
    } while(!CAS(&S,t,x));
}
```
Treiber’s Non-Blocking Stack

```c
struct Node {
    value_t data;
    Node *next;
};

Node *S;

void init() { S = NULL; }

value_t pop() {
    Node *t, *x;
    do { t = S;
        if (t == NULL) { return EMPTY; }
        x = t->next;
    } while(!CAS(&S,t,x));
    return t->data;
}

void push(value_t v) {
    Node *t, *x;
    x = new Node();
    x->data = v;
    do { t = S;
        x->next = t;
    } while(!CAS(&S,t,x));
}
```

CAS-guarded while-loop to detect interference.
Proving Lock-Freedom: Step One

Reducing the problem to termination

Let $D$ be a share-memory data structure with operations $\pi_1, \ldots, \pi_k$.

**Definitions**

$$S^n = \{ op_1; \ldots; op_n \mid \forall i : op_i \in \{\pi_1, \ldots, \pi_k\} \}$$

$$S = \bigcup_{n \in N} S^n$$

$$\mathcal{P}^m = \{ \mathbin{\parallel} s_i \mid \forall i : s_i \in S \}$$

$$\mathcal{P} = \bigcup_{m \in N} \mathcal{P}^m$$

**Theorem**

$D$ is lock-free $\iff$ every $P \in \mathcal{P}$ terminates
Proving Lock-Freedom: Step One

Reducing the problem to termination

Let $D$ be a share-memory data structure with operations $\pi_1, \ldots, \pi_k$.

**Definitions**

$$S^n = \{ \text{op}_1; \ldots; \text{op}_n \mid \forall i : \text{op}_i \in \{\pi_1, \ldots, \pi_k\} \}$$

$$S = \bigcup_{n \in \mathbb{N}} S^n$$

$$\mathcal{P}^m = \{ \big\| s_i \mid \forall i : s_i \in S \}_{i=1,\ldots,m}$$

$$\mathcal{P} = \bigcup_{m \in \mathbb{N}} \mathcal{P}^m$$

**Theorem**

$D$ is lock-free $\iff$ every $P \in \mathcal{P}$ terminates
Proving Lock-Freedom: Step One

Reducing the problem to termination

Let \( D \) be a share-memory data structure with operations \( \pi_1, \ldots, \pi_k \).

**Definitions**

\[
S^n = \{ \text{op}_1; \ldots; \text{op}_n \mid \forall i : \text{op}_i \in \{\pi_1, \ldots, \pi_k\} \}
\]

\[
S = \bigcup_{n \in \mathbb{N}} S^n
\]

\[
P^m = \left\{ \parallel s_i \mid \forall i : s_i \in S \right\}
\]

\[
P = \bigcup_{m \in \mathbb{N}} P^m
\]

**Theorem**

\( D \) is lock-free \( \iff \) every \( P \in P \) terminates

**Problem:** Termination of one thread depends on the behavior of other threads.
Proving Lock-Freedom: Step Two

Prove an upper bound on the number of loop iterations $P \in \mathcal{P}$

How to reason about one operation $\pi_i$ locally when the number of loop iterations depends on the other threads?

**Observation**

In a lock-free data structure, every iteration of an unsuccessful operation is caused by the successful completion of an operation by another thread.

Idea: Threads that perform a successful operation have to pay for the additional loop iterations that they (potentially) cause in other threads.
Proving Lock-Freedom: Step Two

Prove an upper bound on the number of loop iterations \( P \in \mathcal{P} \)

How to reason about one operation \( \pi_i \) locally when the number of loop iterations depends on the other threads?

**Observation**

In a lock-free data structure, every iteration of an unsuccessful operation is caused by the successful completion of an operation by another thread.

Idea: Threads that perform a successful operation have to pay for the additional loop iterations that they (potentially) cause in other threads.
A Quantitative Compensation Scheme for Treiber’s Stack
A Quantitative Compensation Scheme for Treiber’s Stack

• Every thread has a number of tokens to pay for its loop iterations

• For Treiber’s stack, a thread needs $m$ tokens to execute a push or pop

• After the execution the tokens disappear

  ▸ If $n$ operations are executed then $m \cdot n$ is a bound on the loop iterations
A Quantitative Compensation Scheme for Treiber’s Stack

• Every thread has a number of tokens to pay for its loop iterations

• For Treiber’s stack, a thread needs \( m \) tokens to execute a push or pop

• After the execution the tokens disappear

→ If \( n \) operations are executed then \( m \times n \) is a bound on the loop iterations
A Quantitative Compensation Scheme for Treiber’s Stack

- Every thread has a number of tokens to pay for its loop iterations

- For Treiber’s stack, a thread needs $m$ tokens to execute a push or pop

- After the execution the tokens disappear

  ▶ If $n$ operations are executed then $m \times n$ is a bound on the loop iterations

- **Successful push/pop:** 1 token is used to pay for the loop iteration
  $m-1$ tokens are transferred to other threads

- **Unsuccessful push/pop:** 1 token is used to pay for the loop iteration
  1 token is received from a successful thread
A Quantitative Compensation Scheme for Treiber’s Stack

• Every thread has a number of tokens to pay for its loop iterations

• For Treiber’s stack, a thread needs \( m \) tokens to execute a push or pop

• After the execution the tokens disappear

  ▸ If \( n \) operations are executed then \( m \times n \) is a bound on the loop iterations

• **Successful push/pop:** 1 token is used to pay for the loop iteration
  \( m-1 \) tokens are transferred to other threads

• **Unsuccessful push/pop:** 1 token is used to pay for the loop iteration
  1 token is received from a successful thread

Availability of \( m \) tokens is a loop invariant.
A Quantitative Compensation Scheme for Treiber’s Stack

Threads

Shared Memory
A Quantitative Compensation Scheme for Treiber’s Stack

Let \( P \) be a program that consists of all operations executed by a single thread. We prove that \( P \) is lock-free if and only if every program \( P \) terminates in a finite number of steps.

This property is the basis of a novel reasoning technique that we verified in a local and modular way. Assume that each of the threads \( T \) that run the program \( P \) in parallel. In this way, every thread is able to “pay” for its loop iterations without being aware of the other threads or the scheduler.

![Figure 1: An implementation of Treiber’s lock-free stack in a C-like language as given by Gotsman et al. (9).](image)

void push(value_t v) {
    Node *t, *x;
    x = new Node();
    x->data = v;
    do {
        t = S;
        x->next = t;
    } while(!CAS(&S,t,x));
}

A key insight of our work is that for many lock-free data structures, the token disappears from the system. Because it is not possible to create or duplicate tokens—tokens are an affine resource—there is a bound on the number of tokens that are initially present in the system. This bound on the total number of loop iterations executed.

To prove the other direction, assume now that every program \( P \) terminates for every \( \pi \). We now define the set of programs \( D \) and prove that \( D \) is lock-free. Let \( f \) be the sequential program that consists of all operations that have been started by thread \( T \) in their temporal order. Then program \( f \) completes an operation, it provides compensation to the completion of an operation by another thread. In return, when a thread completes an operation, it receives compensation from the other threads. Hence, every thread only executes one operation, that is, \( P \) in parallel.

Figure 2: A shared data structure that shows a limitation of the method of ROOF.

This example is based on an idea from James Aspnes.

![Figure 2: A shared data structure that shows a limitation of the method of ROOF.](image)

void ping() {
    //initialization
    Node *t, *x;
    do { t = S; x->data = v; x->next = t; }
    while(!CAS(&S,t,x));
}

Consider for example Treiber’s stack and a program \( P \) in which only the choices of the scheduler. So after a finite number of steps, we then follows that some operation will complete, independently of incomplete operations are left.

Finally, we set the number of operations per thread we avoid the limitations of the method of ROOF.

![Figure 2: A shared data structure that shows a limitation of the method of ROOF.](image)
A Quantitative Compensation Scheme for Treiber’s Stack

Threads

Need m tokens for each push operation.

Shared Memory

void push(value_t v) {
    Node *t, *x;
    x = new Node();
    x->data = v;
    do { t = S;
        x->next = t;
    } while(!CAS(&S,t,x));
}
A Quantitative Compensation Scheme for Treiber’s Stack

Threads

Shared Memory

Need m tokens for each push operation.

void push(value_t v) {
    Node *t, *x;
    x = new Node();
    x->data = v;
    do { t = S;
        x->next = t;
    } while(!CAS(&S,t,x));
}
A Quantitative Compensation Scheme for Treiber’s Stack

Threads

Shared Memory

void push(value_t v) {
Node *t, *x;
x = new Node();
x->data = v;
do { t = S;
x->next = t;
} while(!CAS(&S,t,x));
}
A Quantitative Compensation Scheme for Treiber’s Stack

Threads

8128 77 1 1

Shared Memory

void push(value_t v) {
    Node *t, *x;
    x = new Node();
    x->data = v;
    do { t = S;
        x->next = t;
    } while(!CAS(&S,t,x));
}
A Quantitative Compensation Scheme for Treiber’s Stack

Failed

8128

77

1

Failed

1

Failed

1

Succeeded

1

1

2

Threads

void push(value_t v) {
    Node *t, *x;
    x = new Node();
    x->data = v;
    do { t = S;
        x->next = t;
    } while(!CAS(&S,t,x));
}
A Quantitative Compensation Scheme for Treiber’s Stack

Failed

8128

Failed

77

Succeeded

2

Threads

Shared Memory

void push(value_t v) {
    Node *t, *x;
    x = new Node();
    x->data = v;
    do { t = S;
        x->next = t;
    } while(!CAS(&S,t,x));
}
A Quantitative Compensation Scheme for Treiber’s Stack

Iterate loop. Receive a token from successful thread.

Exits loop. Pays m-1 tokens to other threads.

void push(value_t v) {
    Node *t, *x;
    x = new Node();
    x->data = v;
    do { t = S;
          x->next = t;
    } while(!CAS(&S,t,x));
}
A Quantitative Compensation Scheme for Treiber’s Stack

Iterate loop. Receive a token from successful thread.

Exits loop. Pays m-1 tokens to other threads.

void push(value_t v) {
    Node *t, *x;
    x = new Node();
    x->data = v;
    do { t = S;
        x->next = t;
    } while(!CAS(&S,t,x));
}
A Quantitative Compensation Scheme for Treiber’s Stack

Threads

Shared Memory

void push(value_t v) {
    Node *t, *x;
    x = new Node();
    x->data = v;
    do { t = S;
        x->next = t;
    } while(!CAS(&S,t,x));
}
A Quantitative Compensation Scheme for Treiber’s Stack

Treiber’s stack is lock-free iff we have enough tokens to pay for all loop iterations.

void push(value_t v) {
Node *t, *x;
x = new Node();
x->data = v;
do { t = S;
   x->next = t;
} while(!CAS(&S,t,x));
}
Quantitative Reasoning in Separation Logic

\[ [P] \text{ C } [Q] \]

\[ \frac{[P] \text{ C } [Q]}{[P \ast R] \text{ C } [Q \ast R]} \quad \text{(FRAME)} \]
Quantitative Reasoning in Separation Logic

\[
[P] C [Q]
\]

\[
\frac{[P] C [Q]}{[P \times R] C [Q \times R]} \quad \text{(FRAME)}
\]

P and Q are predicates on program states.
Quantitative Reasoning in Separation Logic

P and Q are predicates on program states.

\[ [P] \quad \text{C} \quad [Q] \]

Total correctness: if program C is started in a state which satisfies P then C terminates and Q holds after the execution.

\[
\frac{[P] \quad \text{C} \quad [Q]}{[P \ast R] \quad \text{C} \quad [Q \ast R]} \quad (\text{FRAME})
\]
Quantitative Reasoning in Separation Logic

\[ [P] C [Q] \]

\[ [P] C [Q] \]

\[ [P \ast R] C [Q \ast R] \]  \quad \text{(FRAME)}

\textbf{Total correctness}: if program C is started in a state which satisfies P then C terminates and Q holds after the execution.

\textbf{Separating conjunction}: heap can be split so that one part satisfies P and the other part satisfies R.
Quantitative Reasoning in Separation Logic

Frame rule for modular and local reasoning.

\[ [P] C [Q] \]

\[ [P] C [Q] \]

\[ [P \ast R] C [Q \ast R] \]

(FRAME)

Total correctness: if program C is started in a state which satisfies P then C terminates and Q holds after the execution.

Separating conjunction: heap can be split so that one part satisfies P and the other part satisfies R.
Quantitative Reasoning in Separation Logic

Frame rule for modular and local reasoning.

\[
\frac{[P] \ C \ [Q] \quad [P * R] \ C \ [Q * R]}{[P] \ C \ [Q]} \quad \text{(FRAME)}
\]

Total correctness: if program C is started in a state which satisfies P then C terminates and Q holds after the execution.

\[
[P] \ C \ [Q]
\]
Quantitative Reasoning in Separation Logic

Frame rule for modular and local reasoning.

\[
\frac{[P] C [Q]}{[P \ast R] C [Q \ast R]} \quad \text{(FRAME)}
\]

(Affine) predicate for tokens:
\[
\Diamond \quad \Diamond \ast P \quad \Diamond \ast \ldots \ast \Diamond \leftrightarrow \Diamond^n
\]

Total correctness: if program \( C \) is started in a state which satisfies \( P \) then \( C \) terminates and \( Q \) holds after the execution.
Quantitative Reasoning in Separation Logic

Frame rule for modular and local reasoning.

\[
\frac{[P] \ C \ [Q]}{[P \ast R] \ C \ [Q \ast R]} \quad \text{(FRAME)}
\]

(Affine) predicate for tokens: \( \Diamond \) \( \Diamond \ast P \) \( \Diamond \ast \ldots \ast \Diamond \leftrightarrow \Diamond^n \)

**Total correctness**: if program C is started in a state which satisfies P then C terminates and Q holds after the execution.

Use separating conjunction to combine it with other predicates.
Quantitative Reasoning in Separation Logic

Frame rule for modular and local reasoning:

\[ [P] C [Q] \]

\[ [P] C [Q] \]

\[ [P \ast R] C [Q \ast R] \]  \text{(FRAME)}

Total correctness: if program C is started in a state which satisfies P then C terminates and Q holds after the execution.

There are n tokens available.

(Affine) predicate for tokens:

\( \Diamond \)

\( \Diamond \ast P \)

\( \Diamond \ast \ldots \ast \Diamond \leftrightarrow \Diamond^n \)

Use separating conjunction to combine it with other predicates.
Quantitative Reasoning in Separation Logic

Frame rule for modular and local reasoning.

\[
\frac{[P] C [Q]}{[P * R] C [Q * R]} \quad \text{(FRAME)}
\]

Total correctness: if program \(C\) is started in a state which satisfies \(P\) then \(C\) terminates and \(Q\) holds after the execution.

(Affine) predicate for tokens: \(\diamond\) \(\diamond \star \ P\) \(\diamond \ldots \star \diamond \leftrightarrow \diamond^n\)
Quantitative Reasoning in Separation Logic

**Frame rule for modular and local reasoning.**

\[
[P] C [Q] \\
\frac{[P] C [Q]}{[P \ast R] C [Q \ast R]} \quad \text{(FRAME)}
\]

(Affine) predicate for tokens: \(\Diamond\) \quad \Diamond \ast P \quad \Diamond \ast \ldots \ast \Diamond \leftrightarrow \Diamond^n \)

\[
P \land B \quad \Rightarrow \quad P' \ast \Diamond \quad \frac{[P'] C [P]}{[P] \text{ while } B \text{ do } C [P \land \lnot B]} \quad \text{(WHILE)}
\]

**Total correctness:** if program C is started in a state which satisfies P then C terminates and Q holds after the execution.
Quantitative Reasoning in Separation Logic

Frame rule for modular and local reasoning.

\[
[P] C [Q] \frac{[P] C [Q]}{[P \ast R] C [Q \ast R]} \quad \text{(FRAME)}
\]

Total correctness: if program C is started in a state which satisfies P then C terminates and Q holds after the execution.

(Affine) predicate for tokens: \(\Diamond\) \(\Diamond \ast P\) \(\Diamond \ast \ldots \ast \Diamond \leftrightarrow \Diamond^n\)

Consume one token in each iteration

\[
P \land B \quad \Rightarrow \quad P' \ast \Diamond \quad \frac{[P'] C [P]}{[P] \text{while } B \text{ do } C [P \land \neg B]} \quad \text{(WHILE)}
\]
Concurrent Separation Logic

\[ I \vdash [P] C [Q] \]

\[ \text{emp} \vdash [P * I] C [Q * I] \]

\[ I \vdash [P] \text{atomic}\{C\} [Q] \quad (\text{ATOM}) \]

\[ I \vdash [P_1] C_1 [Q_1] \quad I \vdash [P_2] C_2 [Q_2] \]

\[ I \vdash [P_1 * P_2] C_1 \parallel C_2 [Q_1 * Q_2] \quad (\text{PAR}) \]
Concurrent Separation Logic

Resource invariant: another predicate.

\[ I \vdash [P] C [Q] \]

\[ \text{emp} \vdash [P \ast I] C [Q \ast I] \]
\[ I \vdash [P] \text{atomic}\{C\} [Q] \quad \text{\((ATM)\)} \]

\[ I \vdash [P_1] C_1 [Q_1] \quad I \vdash [P_2] C_2 [Q_2] \]
\[ I \vdash [P_1 \ast P_2] C_1 \parallel C_2 [Q_1 \ast Q_2] \quad \text{\((PAR)\)} \]
Concurrent Separation Logic

Resource invariant: another predicate. Describes the shared memory.

\[ I \vdash [P] C [Q] \]

\[
\begin{align*}
\text{emp} & \vdash [P * I] C [Q * I] \\
I & \vdash [P] \text{atomic}\{C\} [Q]
\end{align*}
\]

(ATOM)

\[
\begin{align*}
I & \vdash [P_1] C_1 [Q_1] \\
I & \vdash [P_2] C_2 [Q_2]
\end{align*}
\]

(PAR)

\[
I \vdash [P_1 * P_2] C_1 \parallel C_2 [Q_1 * Q_2]
\]
Concurrent Separation Logic

Resource invariant: another predicate.

Describes the shared memory.

\[ I \vdash [P] C [Q] \]

\[ \text{emp} \vdash [P \ast I] C [Q \ast I] \]

\[ I \vdash [P] \text{atomic}\{C\} [Q] \] \hspace{1cm} \text{(ATOM)}

Shared memory can only be accessed in atomic blocks.

\[ I \vdash [P_1] C_1 [Q_1] \quad I \vdash [P_2] C_2 [Q_2] \]

\[ I \vdash [P_1 \ast P_2] C_1 \parallel C_2 [Q_1 \ast Q_2] \] \hspace{1cm} \text{(PAR)}
Concurrent Separation Logic

Resource invariant: another predicate.

\[ I \vdash [P] C [Q] \]

Describes the shared memory.

Assume invariant I holds.

\[ \text{emp} \vdash [P \times I] C [Q \times I] \]

\[ I \vdash [P] \text{atomic}\{C\} [Q] \] (ATOM)

Ensure invariant I holds.

Shared memory can only be accessed in atomic blocks.

\[ I \vdash [P_1] C_1 [Q_1] \quad I \vdash [P_2] C_2 [Q_2] \]

\[ I \vdash [P_1 \times P_2] C_1 \parallel C_2 [Q_1 \times Q_2] \] (PAR)
Concurrent Separation Logic

Resource invariant: another predicate.

Describes the shared memory.

\[ I \vdash [P] C [Q] \]

Assume invariant I holds.

Ensure invariant I holds.

\[ \text{emp} \vdash [P \ast I] C [Q \ast I] \]
\[ I \vdash [P] \text{atomic}\{C\} [Q] \quad \text{(ATOM)} \]

Shared memory can only be accessed in atomic blocks.

\[ I \vdash [P_1] C_1 [Q_1] \quad I \vdash [P_2] C_2 [Q_2] \]
\[ I \vdash [P_1 \ast P_2] C_1 \| C_2 [Q_1 \ast Q_2] \quad \text{(PAR)} \]

Parallel composition.
Concurrent Separation Logic

Resource invariant: another predicate.

\[ I \vdash [P] C [Q] \]

Describes the shared memory.

Assume invariant I holds.

\[
\text{emp} \vdash [P \ast I] C [Q \ast I] \\
I \vdash [P] \text{atomic}\{C\} [Q] \quad (\text{ATOM})
\]

Ensure invariant I holds.

Shared memory can only be accessed in atomic blocks.

Shared memory can only be accessed in atomic blocks.

\[
I \vdash [P_1] C_1 [Q_1] \\
I \vdash [P_2] C_2 [Q_2] \\
I \vdash [P_1 \ast P_2] C_1 \parallel C_2 [Q_1 \ast Q_2] \quad (\text{PAR})
\]

Modular reasoning.

Parallel composition.
Back to the Stack: Formal Specification

Resource Invariant

\[ I \triangleq \exists u. S \mapsto u * \bigotimes_{0 \leq i < n} \alpha(i, u) \]

\[ \alpha(i, u) \triangleq \exists a, c. C[i] \mapsto c * A[i] \mapsto a * (c = 0 \lor a = u \lor \Diamond) \]

Specifications of Push and Pop

\[ I \vdash [A[tid] \mapsto _* C[tid] \mapsto _* \Diamond^n] \text{ push}(v) \ [A[tid] \mapsto _* C[tid] \mapsto _] \]

\[ I \vdash [A[tid] \mapsto _* C[tid] \mapsto _* \Diamond^n] \text{ pop()} \ [A[tid] \mapsto _* C[tid] \mapsto _] \]
Back to the Stack: Formal Specification

Resource Invariant

\[ I \triangleq \exists u. \; S \mapsto u \ast \bigotimes_{0 \leq i < n} \alpha(i, u) \]

\[ \alpha(i, u) \triangleq \exists a, c. \; C[i] \mapsto c \ast A[i] \mapsto a \ast (c = 0 \lor a = u \lor \diamond) \]

Specifications of Push and Pop

\[ I \vdash [A[tid] \mapsto \_ \ast C[tid] \mapsto \_ \ast \diamond^n] \; \text{push}(v) \; [A[tid] \mapsto \_ \ast C[tid] \mapsto \_] \]

\[ I \vdash [A[tid] \mapsto \_ \ast C[tid] \mapsto \_ \ast \diamond^n] \; \text{pop()} \; [A[tid] \mapsto \_ \ast C[tid] \mapsto \_] \]
While loop of push

Verifying Treiber’s Stack

While loop of push

While loop of push

Verifying Treiber’s Stack

While loop of push
<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Tokens Per Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Treiber’s Stack</td>
<td>$n$</td>
</tr>
<tr>
<td>Michael and Scott’s Queue</td>
<td>$n + 1$</td>
</tr>
<tr>
<td>Hazard-Pointer Stack</td>
<td>$n + (c \cdot n)$</td>
</tr>
<tr>
<td>Hazard-Pointer Queue</td>
<td>$(n + 1) + (c \cdot n)$</td>
</tr>
<tr>
<td>Elimination-Backoff Stack</td>
<td>$n \cdot (n + 1)$</td>
</tr>
</tbody>
</table>

More Advanced Shared-Memory Data Structures

Details are in the Paper
Conclusion and Ongoing Work

• Compensation schemes simplify reasoning about lock-freedom

• Quantitative reasoning can be directly integrated in modern logics for safety properties

• The reasoning works for involved real-world data structures

• Current work: Adapt quantitative reasoning to prove termination-sensitive contextual refinement

• Future work: quantitative reasoning for classic lock-based synchronization
Conclusion and Ongoing Work

• Compensation schemes simplify reasoning about lock-freedom

• Quantitative reasoning can be directly integrated in modern logics for safety properties

• The reasoning works for involved real-world data structures

• Current work: Adapt quantitative reasoning to prove termination-sensitive contextual refinement

• Future work: quantitative reasoning for classic lock-based synchronization

Thank you!