CONCURRENT/DISTRIBUTED DATA TYPES
Correctness Criteria, Verification

Constantin Enea
Ecole Polytechnique
Parallel Data Processing

Software systems that support high-frequency parallel accesses to high-quantity data.

Rely on high-performance shared resources (data types).

• key-value maps, sets, queues, stacks, etc.

Deployed over a shared-memory or a network
Concurrent Objects

Multi-threaded programming

e.g. Java Development Kit SE
dozens of objects, including queues, maps, sets, lists, locks, atomic integers, …
Shared-State in Parallel Applications

- Parallelizing applications for efficiency

**Sequential**

```java
// reading data for future processing
q = new Queue();
while (...){
    X = readFile();
    q.enqueue(X);
}
```
Shared-State in Parallel Applications

- Parallelizing applications for efficiency

### Sequential

```java
// reading data for future processing
q = new Queue();
while(...){
    X = readFile();
    q.enqueue(X);
}
```

### Parallel

```java
q = new Queue();
while(...){
    X = readFile1();
    q.enqueue(X);
}
while(...){
    X = readFile2();
    q.enqueue(X);
}
```

- multi-threading
- distributed over a network

```java
||
X = readFile1();
q.enqueue(X);
||
X = readFile2();
q.enqueue(X);
```
Shared-State in Parallel Applications

- Parallelizing applications for efficiency

### Sequential

```java
// reading data for future processing
q = new Queue();
while(...){
    X = readFile();
    q.enqueue(X);
}
```

### Parallel

```java
q = new Queue();
l = new Lock();
while(...){
    X = readFile1();
    l.lock();
    q.enqueue(X);
    l.unlock();
}
while(...){
    X = readFile2();
    l.lock();
    q.enqueue(X);
    l.unlock();
}
```

- multi-threading
- distributed over a network
Shared-State in Parallel Applications

- Parallelizing applications for efficiency

### Sequential

```java
// reading data for future processing
q = new Queue();
while(...){
    X = readFile();
    q.enqueue(X);
}
```

### Parallel

```java
q = new Queue();
l = new Lock();
while(...){
    X = readFile1();
    l.lock();
    q.enqueue(X);
    l.unlock();
}
```

How to implement concurrent/distributed objects?
How is correctness defined? Verification?
Verification Ingredients

- Specifying a Library: $\varphi$
- Implementing a Library: $\bot$
- Verifying a Library implementation: $\bot \models \varphi$
Specifying Sequential Objects
How can we specify an object? (Library)
- Objects API
- Use cases
- Pre and Post Conditions?

What are the behaviors of a client using the library?
💡 for any client making library calls record the inputs and outputs of each call
What is a client?

- What is a client of the Library?
  - Program that issues calls to a library instance

```c
// do something
q.enqueue(v)
// do something
x = q.dequeue()
// ...
```

- How do we specify a Data Structure (DS) generically?
  - Histories of calls and returns
  - Constraint possible return values

```c
q.enqueue(v) return
... q.dequeue() return
```
Well Encapsulated Objects

- *Global* object state:
  - Possibly *local* thread state
  - A set of *operations* or *methods*
    - Input and output types
    - Methods are the only way to operate on the state
Sequential Object Specifications

- Library $L = \langle \Sigma, m_1, m_2, m_3 \rangle$

```
SMGC(L):
  while true do
    $m_i = \text{choseMethodFrom}(L);$;
    args = \text{choseInputsFor}(m);
    $m_i(args);$;
  od
```

- Client C: Issues calls to the library methods
  - (Sequential) Most General Client [SMGC]

- We will talk about histories of calls with values
  - $\epsilon$ denotes the empty sequence,
  - $o$ denotes an operation (eg. $<\text{pop}(), v>$), and
  - $\delta$ denotes a sequence of operations
Specifying a Register

- Inductive histories of a Register:
  1. $\epsilon$ is a Register History (RH)
  2. $\langle\text{read}(), 0\rangle^*$ is a Register History
  3. if $\delta$ is a RH, then so is $\delta \cdot \langle\text{write}(v), _\rangle$
  4. if $\delta \cdot \langle\text{write}(v), _\rangle$ is a RH, then so it is $\delta \cdot \langle\text{read}(), v\rangle^*$

Some examples on the board
Specifying a Stack

- Inductive histories of a Stack:
  1. $\varepsilon$ is a Stack History (SH)
  2. if $\delta \cdot \langle \text{pop}(), v \rangle$ is a SH, then so is $\langle \text{push}(w), \_ \rangle \cdot \delta$
  3. if $\delta$ is a SH, and $|\{ \langle \text{pop}(), v \rangle : \delta | v \neq \bot \}| = |\text{push}(v), \_ : \delta||$, then so it is $\delta \cdot \langle \text{pop}(), \bot \rangle^*$
  4. same conditions as 3, and $\langle \text{pop}(), \bot \rangle$ does not occur in $\delta$ then, $\langle \text{push}(w), \bot \rangle \cdot \delta \cdot \langle \text{pop}(), w \rangle$ is a SH
  5. if $\delta_0 \cdot \langle \text{pop}(), \bot \rangle$ is a SH, and $\delta_1$ is a SH, then $\delta_0 \cdot \delta_1$ is SH
Specifying a Queue

Exercise
Implementations

- A set implementation based on sorted linked lists

public class Entry {
    public Object value;
    public Entry next;
}

public class Set {
    Entry first;
    public boolean add(Object x) {...}
    public boolean remove(Object x) {...}
    public boolean contains(Object x) {...}
}

Sentinel node never deleted
A set implementation based on sorted linked lists

adding an entry

removing an entry
Specifying Concurrent Objects
What about Concurrency?

while true do
    \( m_i = \text{choseMethodFrom}(L); \)
    \( \text{args} = \text{choseInputsFor}(m); \)
    \( m_i(\text{args}); \)
    od

while true do
    \( m_i = \text{choseMethodFrom}(L); \)
    \( \text{args} = \text{choseInputsFor}(m); \)
    \( m_i(\text{args}); \)
    od

\( \text{s.push}(v) \quad \text{return} \quad \text{s.pop}() \quad \text{return} \quad v \)

Concurrent Consistency Criteria

Should this be legal?
Concurrent Clients

- Most General Client (seq)

- Most General Client (concurrent n threads)

- Concurrent Library Verification w.r.t. \( \text{CMGC}_n(L) \) for any n
Concurrent Consistency Criteria

- Quiescence Consistency
- Sequential Consistency
- Linearizability

We will work with Registers to exemplify the definitions.
Method calls should appear to happen one-at-a-time, sequential order

\[
\begin{align*}
\text{r.write(1)}; & \quad \text{r.read(); } \quad \text{r.write(1)} & \quad \text{ret} & \quad \text{r.write(2)} & \quad \text{ret} \\
\text{r.write(2)}; & \quad \text{r.read(); } \quad \text{r.read()} & \quad \text{ret} & \quad 2 & \quad \text{r.read()} & \quad \text{ret} & \quad 0
\end{align*}
\]
Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order
- Method calls separated by a period of quiescence should appear to take effect in their real time order

\[ \langle \text{r.write(1)} , \_ \rangle, \langle \text{r.read()},2 \rangle \]

\[ \langle \text{r.write(2)},\_ \rangle, \langle \text{r.read()},0 \rangle \]

✗
Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order.

- Method calls separated by a period of quiescence should appear to take effect in their real time order.

\[
\begin{align*}
\text{r.write(1)} & \quad \text{ret} \\
\text{r.read()} & \quad \text{ret 2} \\
\text{r.write(2)} & \quad \text{ret} \\
\text{r.read()} & \quad \text{ret 0}
\end{align*}
\]

\[
\begin{align*}
\langle \text{r.write(1)}, \_ \rangle & \quad \langle \text{r.read()}, 0 \rangle \\
\langle \text{r.read()}, 2 \rangle & \quad \langle \text{r.write(2)}, \_ \rangle \\
\langle \text{r.read()}, 0 \rangle & \quad \langle \text{r.write(1)}, \_ \rangle \quad \langle \text{r.write(2)}, \_ \rangle \\
\langle \text{r.read()}, 2 \rangle & \quad \langle \text{r.read()}, 0 \rangle
\end{align*}
\]
Sequential Consistency

- How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Computer Programs [Lamport'79]
  - Each process issues operations in the order specified by its program.
  - Operations from all processors issued to a single object are serviced from a single FIFO queue. Issuing an operation consists in entering a request on this queue.
Sequential Consistency

\[ r.\text{write}(1); \quad r.\text{read}(); \quad r.\text{write}(2); \quad r.\text{read}(); \]

- **Incorrect**: The diagram shows a conflict, indicated by the red 'X', where the order of operations does not match the sequential consistency model.

\[ <r.\text{write}(1), \_> \rightarrow <r.\text{write}(2), \_> \]

\[ <r.\text{read}(), 2> \rightarrow <r.\text{read}(), 0> \]
Sequential Consistency

```plaintext
r.write(1); r.read();
r.write(2); r.read();
```

Diagram:
- `<r.write(1), _>` → `<r.write(2), _>`
- `<r.read(), 0>` → `<r.read(), 1>`

Annotations:
- `r.write(1)` → `r.read()` → `ret 0`
- `r.write(2)` → `r.read()` → `ret 1`
Sequential Consistency

- Quiescent Consistency +
- Method calls should appear to take effect in Program Order

Program order:
```
  r.write(1);  r.read();
  r.write(2);  r.read();
```

- Each history $\delta$ induces a per-thread total order of operations
  - $o_1 \prec_\delta o_2$ iff $o_1$ and $o_2$ are on the same thread, and $o_1$ occurs before $o_2$ in $\delta$

- A history $\delta$ is Sequentially Consistent if there exists an equivalent *Sequential* history $\delta'$ (i.e. same operations), and
  - $o_1 \prec_\delta o_2$ implies $o_1 \prec_{\delta'} o_2$
Sequential Consistency

x.write(1)  ret  x.write(2)  ret
        |        |        |
        |        |        |
        |        |        |
x.read()  ret1  x.read()  ret0

✔️

✗
Linearizability

- Same conditions as Sequential Consistency +
- Each method call should appear to take effect instantaneously at some moment between its invocation (call) and response (return)
- That is: we can pretend that the execution of each method is uninterrupted by other calls to the object
- De-facto standard for Concurrent Object Correctness (eg. java.util.concurrent)

Linearizability: A Correctness Condition for Concurrent Objects
[Herlihy and Wing ‘90]
Linearizability

- Each history $\delta$ induces a partial order on operations such that
  - $o_1 \preceq_\delta o_2$ iff $\text{ret } o_1$ occurs before $\text{call } o_2$ in $\delta$
- A history $\delta$ is Linearizable if there exists an equivalent *Sequential* history $\delta'$ (i.e. same operations), and
  - $o_1 \preceq_\delta o_2$ implies $o_1 \preceq_{\delta',\delta} o_2$
- Ignoring uncompleted operations
- Strictly stronger than Sequential Consistency
Linearizability

- Each operation takes place atomically within its call/return.
Linearizability

- Each operation takes place atomically within its call/return

Not Linearizable
Linearizability

- Each operation takes place atomically within its call/return
Linearizability

- Each operation takes place atomically within its call/return
Linearizability vs. Sequential Consistency

```
x.write(1)    ret

x.read(1)    ret

x.write(2)    ret
```

Not linearizable to begin with!
Observational Refinement

- Linearizability => observational refinement

Reference implementation

```cpp
class AtomicStack {
    cell* top;
    Lock l;

    void push (int v) {
        l.lock();
        top->next = malloc(sizeof *x);
        top = top->next;
        top->data = v;
        l.unlock();
    }

    int pop () {
        ...
    }
}
```

Efficient implementation

```cpp
class TreiberStack {
    cell* top;

    void push (int v) {
        cell* t;
        x = malloc(sizeof *x);
        x->data = v;
        do {
            t = top;
            x->next = top;
        } while (!CAS(&top,t,x));

        int pop () {
            ...
        }
    }
}
```

For every Client, Client x Impl included in Client x Spec
Linearizability: Compositionallity

- **Theorem**: A history $\delta$ is linearizable if and only if for each object $\circ$ in $\delta$, $\delta\circ$ is linearizable

  *Proof*: Simple induction on the number of operations appearing in $\delta$

- **Corollary**: It is enough to show that each Library is linearizable to know that the system is
Some Object Implementations
Lock-Free Implementations

Blocking Reference Implementation

Efficient Nonblocking Implementation

atomic instructions, e.g., compare-and-swap

mutual exclusion
“Basic” Objects
Spin Lock

```c
int Lock = 0;
TID owner = null;

void lock(){
    bool l;
    do {
        while(Lock == 1);
        l = cas(Lock, 0, 1);
    } until (l);
    owner = getTID();
    return;
}

void unlock(){
    owner = null;
    Lock = 0;
    return;
}
```
class IntPtr {
    int val;
}
IntPtr COU;

void inc(int v) {
    int n;
    while(true) {
        n = COU->val;
        if (cas(COU->val, n, n+v))
            break;
    }
    return;
}

void dec(int v) {
    int n;
    while(true) {
        n = COU->val;
        if (cas(COU->val, n, n-v))
            break;
    }
    return;
}

int read() {
    return COU->val;
}
Stack Implementations
class Node {   class NodePtr {     Node tl;         Node val;     int val;         } TOP;   }

void push(int e) {  
    Node y, n;
    y = new();
    y->val = e;
    while(true) {  
        n = TOP->val;
        y->tl = n;
        if (cas(TOP->val, n, y))
            break;
    }
}

int pop() {  
    Node y, z;
    while(true) {  
        y = TOP->val;
        if (y==0) return EMPTY;
        z = y->tl;
        if (cas(TOP->val, y, z))
            break;
    }
    return y->val;
}
Treiber Stack (ABA bug)

pushed: 1, 2, 3
popped: 1, 3, EMPTY

PROBLEM
not admitted by atomic stack
HSY Elimination Stack

Extremely simplified version: 1 collision

```java
class Node {
    Node tl;
    int val;
}
class NodePtr {
    Node val;
} TOP;
class TidPtr {
    int val;
} clash;

void push(int e) {
    Node y, n;
    TID hisId;
    y = new();
    y->val = e;
    while (true) {
        n = TOP->val;
        y->tl = n;
        if (cas(TOP->val, n, y))
            return;
        //elimination scheme
        TidPtr t = new TidPtr();
        t->val = e;
        if (cas(clash,null,t)){
            wait(DELAY);
            //not eliminated
            if (cas(clash,t,null))
                continue;
            else break; //eliminated
        }
    }
}

int pop() {
    Node y,z;
    int t;
    TID hisId;
    while (true) {
        y = TOP->val;
        if (y == 0)
            return EMPTY;
        z = y->tl;
        t = y->val;
        if (cas(TOP->val, y, z)
            return t;
        //elimination scheme
        pusher = clash;
        while (pusher!=null){
            if (cas(clash,pusher,null))
                //eliminated push
            return pusher->val;
        }
    }
}
[Hendler et al.'04]
```
Queue Implementations
Two Locks Queue

class Node {
    int val;
    Node tl;
}

class Queue {
    Node head;
    Node tail;
    thread_id hlock;
    thread_id tlock;
} Q;

void enqueue(int v) {
    Node n, t;
    n = new();
    n->val = v;
    n->tl = NULL;
    lock (&Q->tlock);
    temp = Q->tail;
    temp->tl = node;
    Q->tail = node;
    unlock (&Q->tlock);
}

int dequeue() {
    Node n, new_h;
    int v;
    lock (&Q->hlock);
    n = Q->head;
    new_h = n->tl;
    if (new_h == NULL) {
        unlock (&Q->hlock);
        return EMPTY;
    } else {
        value = new_h->val;
        Q->head = new_h;
        unlock (&Q->tlock);
        //dispose(n);
        return v;
    }
}

Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms [Michael,Scott'96]
Michael and Scott Queue

```java
class Node {
    int val;
    Node nd, nxt, tl;
    Node tl;
}

class Queue {
    Node head;
    Node tail;
} Q;

type int dequeue() {
    Node nxt, hd, tl;
    int pval;
    while(true) {
        hd = Q->head;
        tl = Q->tail;
        nxt = hd->tl;
        if (Q->head != hd) continue;
        if (hd == tl) {
            if (nxt == NULL) return EMPTY;
            cas(Q->tail, tl, nxt);
        } else {
            pval = next->val;
            if (cas(Q->head, hd, nxt)) return pval;
        }
    }
}

type void enqueue(int v) {
    Node nd, nxt, tl;
    int b1;
    nd = new();
    nd->val = v;
    nd->tl = NULL;
    while(true) {
        tl = Q->tail;
        nxt = tl->tl;
        if (Q->tail == tl) b1 = 1;
        else b1 = 0;
        if (b1!=0) {
            if (nxt == 0) {
                if (cas(tl->tl,nxt,nd)) break;
            } else cas(Q->tail,tl, nxt);
        }
        cas(Q->tail, tl, nd);
    }
}
```

[Michael,Scott'96]
class Node {
  int val; // -1 NAN
  Node tl;
  thread_id alloc;
}

class Queue {
  Node head;
  Node tail;
  } Q;

void enqueue(int value) {
  Node nd, tl;
  nd = new();
  nd->alloc = TID;
  nd->val = -1;
  nd->tl = NULL;
  atomic {
    tl = Q->tail;
    tl->tl = nd;
    Q->tail = nd;
  } // end of slot reservation;
  nd->val = value;//value written;
}

int dequeue() {
  Node curr, tail;
  int pval;
  while (true) {
    curr = Q->head;
    tail = Q->tail;
    while (curr != tail) {
      atomic { //atomic swap
        pval = curr->val;
        curr->val = -1;
        if (pval != -1)
          return pval;
        curr = curr->tl;
      }
    }
    curr = curr->tl;
    if (pval != -1)
      return pval;
    return -1;
  }
}

Linearizability: A Correctness Condition for Concurrent Objects
[Herlihy and Wing ‘90]
Set Implementations (TAMP slides)
Replicated objects

Distributed systems

Conflicting concurrent updates: how are they observed on different replicas?

Adversarial environments: crashes, network partitions
Pessimistic Replication

Using consensus algorithms to agree on an order between conflicting concurrent updates.

- Strong consistency
- Availability

CAP theorem [Gilbert et al.’02]: strong consistency + availability + partition tolerance is impossible.
Optimistic Replication

Each update is applied on the local replica and propagated asynchronously to other replicas.

- **Strong consistency** is not achieved.
- **Availability** is the key feature.

Replicas may store different versions of data: **weak consistency**.