On Correctness of Concurrent Data Structures under Reads-Write Concurrency

Kfir Lev-Ari\textsuperscript{1} \hspace{2cm} Gregory Chockler\textsuperscript{2} \hspace{2cm} Idit Keidar\textsuperscript{1}

\textsuperscript{1}Technion, Israel
\textsuperscript{2}Royal Holloway, UK
Data Structure Model

- Shared memory, consists of read-write atomic registers.

- Operation - a sequence of steps.
  - Invoke, Read, Write and Return.

- *Shared state* - assignment to shared variables.

- *Local state* - assignment to operation’s private variables.
From Sequential to Concurrent

Parallelize

Key challenge: is it correct?
What is a Correct Concurrent Behaviour?

Two aspects:

- Inner Correctness
- Outer Correctness
Outer Correctness

Reflected in method invocations and responses (i.e., histories):

*Linearizability* -
Every concurrent history has an equivalent sequential history.

*Regularity* (see below), *Sequential consistency* (see paper).
Inner Correctness

Validity -
Concurrent execution does not reach local states that are not reachable in sequential ones.

Avoid situations like division by zero, null reference, etc.
Focus on Reads-Write Concurrency

- Concurrent modifications are hard.
  
  ![CONCURRENCY](Y U SO HARD?)

- Contention among writers can hamper performance.
- Like many approaches, **do not allow write-write concurrency**.
  - e.g., flat combining, RCU, coarse grain r/w locks
Generality VS Performance

Tradeoff

Generality
(e.g., TM)

Performance
(e.g., fine grained synchronization)

Standard general approach: read-set validation (inner & outer concurrency)
Read-set Validation

All read values must belong to the same initial shared state.
**New Concept: Base Conditions**

Defined per local state of r/o operation

Base condition: \( \exists \text{path from head to 13.} \)

*Forget other values in readset.*

*Consistent Snapshot:* some shared state satisfies a base condition.
Base Conditions - Example

**local state**

- \{\}\n- \{tmp = 1\}\n- \{tmp = 1, res = 7\}

**base condition**

- \(\Phi_1: true\)
- \(\Phi_2: lastPos = 1\)
- \(\Phi_3: lastPos = 1 \land v[1] = 7\)

**Operation**

```
readLast()
tmp ← read(lastPos)
res ← read(v[tmp])
return(res)
```
**Local state**

- \{\}
- \{tmp = 1\}
- \{tmp = 1, res = 7\}

**Base condition**

- \(\Phi_1: true\)
- \(\Phi_2: lastPos = 1\)
- \(\Phi_3: lastPos = 1 \land v[1] = 7\)

**Operation**

\[
\text{readLast()}
\]

\[
tmp \leftarrow \text{read}(lastPos)
\]

\[
res \leftarrow \text{read}(v[tmp])
\]

\[
\text{return (res)}
\]

Two shared states satisfying \(\Phi_3\)

![Two shared states diagram](image)

Every sequential execution of `readLast` from these shared states reaches the same final local state.
**Base Conditions - Example**

```plaintext
writeSafe(val)
i ← read(lastPos)
write(v[i+1], val)
write(lastPos, i + 1)

writeUnsafe(val)
i ← read(lastPos)
write(lastPos, i + 1)
write(v[i+1], val)
```
Base Conditions - Example

```plaintext
writeSafe(val)
i ← read(lastPos)
write(v[i+1], val)
write(lastPos, i + 1)
```

```plaintext
writeUnsafe(val)
i ← read(lastPos)
write(lastPos, i +1)
write(v[i+1],val)
```

With `writeSafe`, every concurrent `readLast` sees values of `lastPos` and `v[tmp]` from the same sequential reachable shared state.
**Model**

**Motivation**

**Base Conditions**

**Base Points**

**Single VMP**

**Methodology**

**Conclusions**

---

### Base Conditions - Example

```
writeSafe(val)
i ← read(lastPos)
write(v[i+1], val)
write(lastPos, i + 1)
```

```
writeUnsafe(val)
i ← read(lastPos)
write(lastPos, i + 1)
write(v[i+1],val)
```

With `writeSafe`, every concurrent `readLast` sees values of `lastPos` and `v[tmp]` from the same sequential reachable shared state.

With `writeUnsafe`, they do not.
For validity - every local state \( l \) has a shared state that satisfies a base condition of \( l \). We call that shared state a \textit{base point} of \( l \).

For linearizability - a base point of \( ro \) is one of the following:

\[\begin{align*}
\leftarrow uo & \rightarrow \leftarrow uo \\
\downarrow & \downarrow \\
\leftarrow uo & \rightarrow \leftarrow uo \\
\downarrow & \downarrow \\
\leftarrow uo & \rightarrow \leftarrow uo \\
\downarrow & \downarrow \\
\leftarrow uo & \rightarrow \leftarrow uo \\
\end{align*}\]

(More options for sequential consistency.)
Wait, are base points enough for linearizability?
Wait, are base points enough for linearizability? Well..
There is no sequential execution of the operations where \( r_{o1} \) returns John and \( r_{o2} \) returns Doe.
But we do achieve *regularity*! (and validity)

Inspired by Lamport: *Regularity* - for every concurrent history, if we remove all but one read-only operation, the history is linearizable.
Every update operation should appear to other threads as if it has a single write step.

Combined with base conditions $\Rightarrow$ linearizability!
To prove correctness:

1. Write base conditions for all steps of read-only operations.
   - Based on sequential code.
   - Prove they are base conditions.

2. Reason about read threads executed concurrently with update operations. Verify that every write step maintains every base condition.

3. For linearizability: Verify that every update operation has a single visible mutation point.
Linked List Example

Base condition: \( \exists \) path from head to 13.

Base conditions:

\[
\begin{align*}
\Phi_1 &: \text{true} \quad \text{next} \leftarrow \text{read(head.next)} \\
&\quad \text{while next} \neq \bot \\
&\quad \text{n} \leftarrow \text{next} \\
\Phi_2 &: \text{head} \Rightarrow n \quad \text{next} \leftarrow \text{read(n.next)} \\
\Phi_3 &: \text{head} \Rightarrow n \quad \text{return(n)}
\end{align*}
\]
Linked List Example

**Base conditions:**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Code</th>
</tr>
</thead>
</table>
| $\Phi_1$: true | $n \leftarrow \perp$
| $\Phi_2$: head $\not\Rightarrow n$ | next $\leftarrow \text{read}(\text{head.next})$
| $\Phi_3$: head $\not\Rightarrow n$ | while next $\neq \perp$
| | $n \leftarrow \text{next}$

**Function: readLast()**

- $n \leftarrow \perp$
- next $\leftarrow \text{read}(\text{head.next})$
- while next $\neq \perp$
- $n \leftarrow \text{next}$
- return $n$

**Function: remove(n)**

- $p \leftarrow \perp$
- next $\leftarrow \text{read}(\text{head.next})$
- while next $\neq n$
- $p \leftarrow \text{next}$
- next $\leftarrow \text{read}(p.next)$
- write(p.next, n.next)

Maintains base conditions

After removal, $n$ is detached. However, it was reachable from *head* in some shared state that is a legal base point for *readLast* that reads $n$. 
### Linked List Example

<table>
<thead>
<tr>
<th>Base conditions:</th>
<th>\textbf{readLast()}</th>
<th>\textbf{remove}_2(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Phi_1): true</td>
<td>(n \leftarrow \bot)</td>
<td>(p \leftarrow \bot)</td>
</tr>
<tr>
<td>(\Phi_2): head (\not\Rightarrow) n</td>
<td>(\text{next} \leftarrow \text{read}(\text{head}.\text{next}))</td>
<td>(\text{next} \leftarrow \text{read}(\text{head}.\text{next}))</td>
</tr>
<tr>
<td>(\Phi_3): head (\not\Rightarrow) n</td>
<td>while (\text{next} \neq \bot)</td>
<td>while (\text{next} \neq n)</td>
</tr>
<tr>
<td>\null</td>
<td>(n \leftarrow \text{next})</td>
<td>(p \leftarrow \text{next})</td>
</tr>
<tr>
<td>\null</td>
<td>return(n)</td>
<td>next (\leftarrow \text{read}(p.\text{next}))</td>
</tr>
<tr>
<td>\null</td>
<td>\null</td>
<td>write(p.next, n.next)</td>
</tr>
<tr>
<td>\null</td>
<td>\null</td>
<td>invalid(n)</td>
</tr>
</tbody>
</table>

\textit{remove}_2 does not maintain \(\Phi_2\) or \(\Phi_3\), since \textit{readLast} might read garbage that was never reachable from the head of the list.
Conclusions and Future Directions

- New framework for reasoning about correctness of data structures.
- Identifying base conditions in sequential code, and ensuring base points under concurrency.
Conclusions and Future Directions

- New framework for reasoning about correctness of data structures.
- Identifying base conditions in sequential code, and ensuring base points under concurrency.

It’s only the tip of the iceberg!
- Write-Write concurrency.
- Create tools for suggesting base conditions.
- A synchronization mechanism that is both general purpose and fine-grained, e.g., combine with TM.
Appendix A - Single Visible Mutation Point

Does not suffice for linearizability by itself.

Here every update operation has a single visible mutation point, but \textit{ro} counts only 1 element in the list:

The initial shared state.

The post-state of \textit{uo}_2.

The post-state of \textit{uo}_1.