

On Correctness of Concurrent Data Structures under Reads-Write Concurrency



Kfir Lev-Ari¹



Gregory Chockler²



Idit Keidar¹

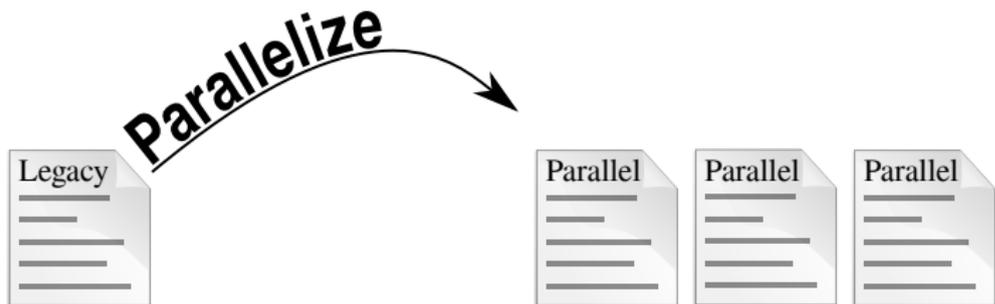
¹Technion, Israel

²Royal 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.

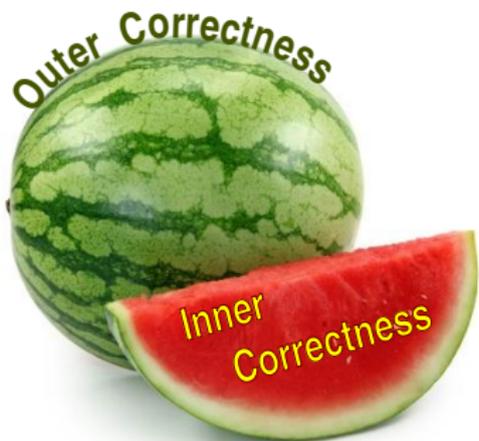
From Sequential to Concurrent



Key challenge: is it correct?

What is a Correct Concurrent Behaviour?

Two aspects:



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.

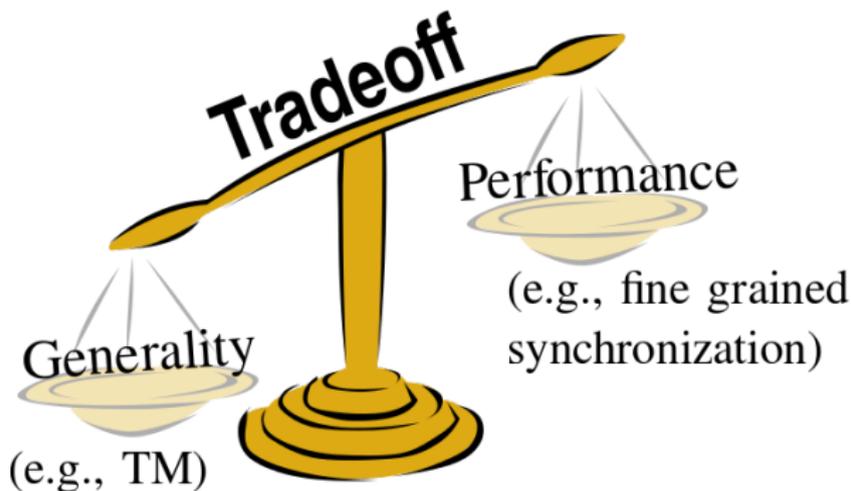
Focus on Reads-Write Concurrency

- Concurrent modifications are hard.



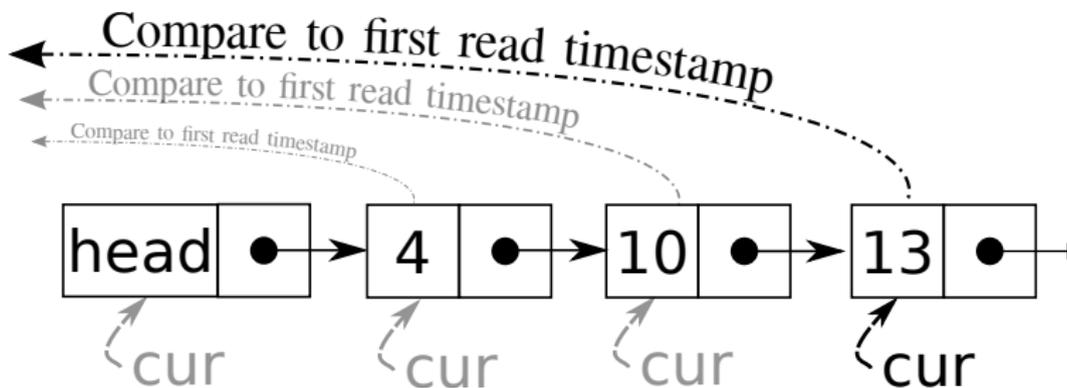
- Contention among writers can hamper performance.
- Like many approaches, **do not allow write-write concurrency.**
 - e.g., flat combining, RCU, coarse grain r/w locks

Generality VS Performance



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

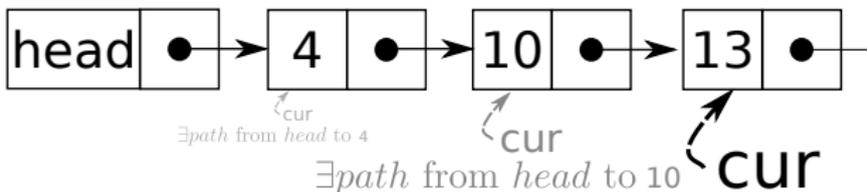
Read-set Validation



All read values must belong to the same initial shared state.

New Concept: Base Conditions

Defined per
local state of
r/o operation



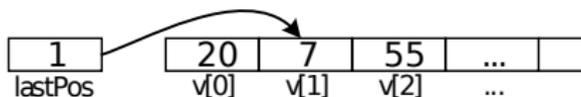
Base condition: $\exists path$ from head to 13.

Forget other values in readset.

Consistent Snapshot:

some shared state satisfies a base condition.

Base Conditions - Example



local state

```
{ }
{ tmp = 1 }
{ tmp = 1, res = 7 }
```

base condition

```
 $\Phi_1 : true$ 
 $\Phi_2 : lastPos = 1$ 
 $\Phi_3 : lastPos = 1 \wedge v[1] = 7$ 
```

Operation readLast()

```
tmp ← read(lastPos)
res ← read(v[tmp])
return (res)
```

Base Conditions - Example

local state

```
{
  tmp = 1
  tmp = 1, res = 7
}
```

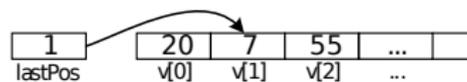
base condition

```
 $\Phi_1 : true$ 
 $\Phi_2 : lastPos = 1$ 
 $\Phi_3 : lastPos = 1 \wedge v[1] = 7$ 
```

Operation readLast()

```
tmp ← read(lastPos)
res ← read(v[tmp])
return (res)
```

Two shared states satisfying Φ_3



Every sequential execution of `readLast` from these shared states reaches the same final local state.

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)
```

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.

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.

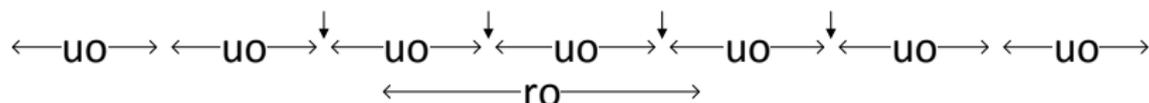
With *writeUnsafe*, they do not.



Base Points

For validity - every local state l has a shared state that satisfies a base condition of l . We call that shared state a *base point* of l .

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



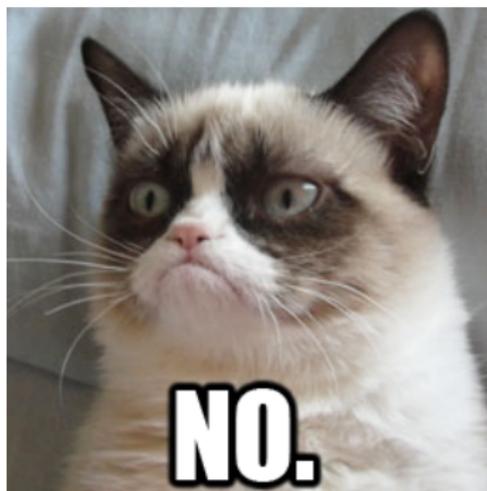
(More options for sequential consistency.)

Base Points and Linearizability

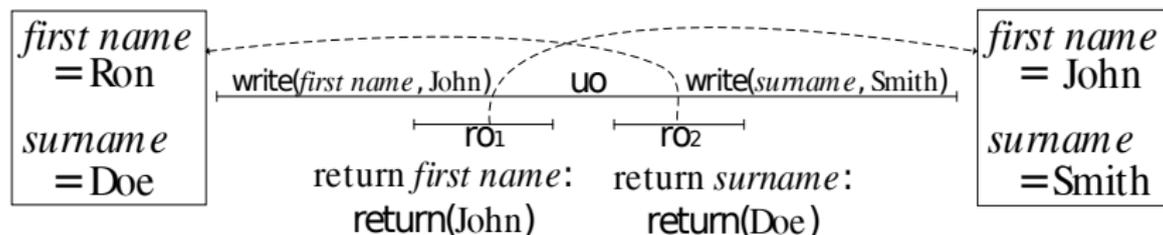
Wait, are base points enough for linearizability?

Base Points and Linearizability

Wait, are base points enough for linearizability? well..



Base Points and Linearizability



There is no sequential execution of the operations where ro_1 returns John and ro_2 returns Doe.

Regularity

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.

Single Visible Mutation Point

Every update operation should appear to other threads as if it has a single write step.



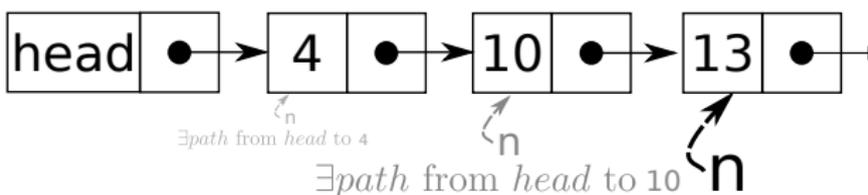
Combined with base conditions \Rightarrow linearizability!

Methodology

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: $\exists \text{path from head to } 13.$

Base conditions:

```

readLast()
n ← ⊥
Φ1 : true  next ← read(head.next)
           while next ≠ ⊥
             n ← next
Φ2 : head ≍* n      next ← read(n.next)
Φ3 : head ≍* n      return (n)
  
```

Linked List Example

Base conditions:

	<i>readLast()</i>	<i>remove(n)</i>
	$n \leftarrow \perp$	$p \leftarrow \perp$
$\Phi_1 : \text{true}$	$\text{next} \leftarrow \mathbf{read}(\text{head.next})$	$\text{next} \leftarrow \text{read}(\text{head.next})$
	$\text{while next} \neq \perp$	$\text{while next} \neq n$
	$n \leftarrow \text{next}$	$p \leftarrow \text{next}$
$\Phi_2 : \text{head} \rightsquigarrow n$	$\text{next} \leftarrow \mathbf{read}(n.\text{next})$	$\text{next} \leftarrow \text{read}(p.\text{next})$
$\Phi_3 : \text{head} \rightsquigarrow n$	$\mathbf{return}(n)$	$\mathbf{write}(p.\text{next}, n.\text{next})$
		 <i>Maintains base conditions</i>

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

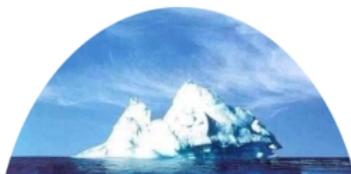
Base conditions:

	<i>readLast()</i>	<i>remove₂(n)</i>
	$n \leftarrow \perp$	$p \leftarrow \perp$
$\Phi_1 : \text{true}$	$\text{next} \leftarrow \mathbf{read}(\text{head.next})$	$\text{next} \leftarrow \text{read}(\text{head.next})$
	$\text{while next} \neq \perp$	$\text{while next} \neq n$
	$n \leftarrow \text{next}$	$p \leftarrow \text{next}$
$\Phi_2 : \text{head} \stackrel{*}{\Rightarrow} n$	$\text{next} \leftarrow \mathbf{read}(n.\text{next})$	$\text{next} \leftarrow \text{read}(p.\text{next})$
$\Phi_3 : \text{head} \stackrel{*}{\Rightarrow} n$	$\mathbf{return}(n)$	$\mathbf{write}(p.\text{next}, n.\text{next})$
		$\mathbf{invalid}(n)$

remove₂ does not maintain Φ_2 or Φ_3 , since *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!

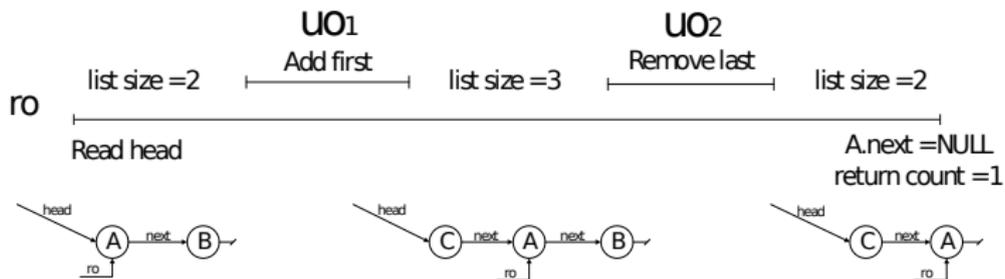
- Write-Write concurrency.
- Create tools for suggesting base conditions.
- A synchronization mechanism that is both general purpose and fine-grained, e.g., combine with TM.



Appendix A - Single Visible Mutation Point

Does not suffice for linearizability by itself.

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



The initial shared state.

The post-state of *uo1*.

The post-state of *uo2*.