Modular Verification of Linearizability with Non-Fixed Linearization Points

Xinyu Feng
USTC

Joint work with Hongjin Liang
Concurrency Verification

• Concurrency: a basic programming skill in multicore era

• Very difficult to ensure correctness
  – Very subtle interleaving

• This talk: correctness of (non-blocking) concurrent obj.
push(int v):
1 int d=0, x, t;
2 x = new Node();
3 x.data = v;
4 while(d==0){
5   t = top;
6   x.next = t;
7   d = cas(&top, t, x);
8 }
Linearizability

• Standard correctness criterion
• All **concurrent** executions are “equivalent” to some **sequential** ones

[Herlihy & Wing’90]
LP Method to Prove Linearizability

• Locate LP in impl code O
• Show it is the single point where the method takes effect

Thread 1:

Thread 2:

is linearized to

push(7), ret pop(), ret(7)
push(v):

1  local d:=0, x, t;
2  x := new Node(v);
3  while (d=0) {
4    t := top;
5    x.next := t;
6    d := cas(&top, t, x);
7  }

Treiber’s Stack

Not update the shared list

Line 6: the only command that changes the list
Problems with LP Method

• Informal
  – Mostly folklore theorem

• Difficult to locate LP
  – LP cannot be statically located – non-fixed LP
  – Common in many wait-free algorithms
Example: HSY Stack

T1: push(v)

T2: pop()

Find T2 is doing pop!
Example: HSY Stack

T1: push(v)

T2: pop()

T1 finishes not only its own opr, but also T2’s
Coordination Pattern in Wait-Free Alg.

T1: push(v)

T2: pop()

Elimination Array

ret v

T1 interrupts T2, and helps T2 to finish its pending opr.

When T2 comes, its job is done.

Difficult to find LP of T2’s opr!
Problems with LP Method

• Informal
  – Mostly folklore theorem

• Difficult to locate LP
  – LP cannot be statically located – non-fixed LP
  – Common in many wait-free algorithms
This Talk

• Simulation-based proof for linearizability
  – Supports compositional verification
  – Formally justifies the folklore LP method

• Extending it for objects with non-fixed LP
  – HSY elimination stack, lazy set, etc.
Linearizability and Program Refinement

**Observation:** Linearizable fine-grained impl. has the same effect as atomic operations.

Thread 1:
- `push(7)`
- `ret`

Thread 2:
- `pop()`
- `ret(7)`

is linearized to
- `push(7), ret` (Thread 1)
- `pop(), ret(7)` (Thread 2)

**Execution of atomic opr**
Linearizability and Program Refinement

Observation: Linearizable fine-grained impl. has the same effect as atomic operations

Reduce linearizability to program equivalence:
push(v):
1  local d:=0, x, t;
2  x := new Node(v);
3  while(d=0){
4    t := top;
5    x.next := t;
6    d := cas(&top, t, x);
7  }

Treiber’s Stack

Abstract representation
stk: n1 :: n2 :: ... :: nk
push(v):
<stk := v::stk>;

Atomic operation

object impl O
atomic opr as spec A
Simulation for Refinement Proof

$(O_0, \sigma_0) \leadsto (O_1, \sigma_1) \leadsto (O_2, \sigma_2) \leadsto \ldots$

$(A_0, \Sigma_0) \leadsto (A_1, \Sigma_1) \leadsto (A_2, \Sigma_2) \leadsto \ldots$

Whatever behavior produced by low-level could be produced by high-level.
push(v):
1  local d:=0, x, t;
2  x := new Node(v);
3  while(d=0){
4    t := top;
5    x.next := t;
6    d := cas(&top, t, x);
7  }

However, needs to consider env., otherwise unsound.
Simulation with Env.

Needs to ensure the env. steps do not break simulation (environment doesn’t do bad things)
Simulation with Env.

- **RGSim**: Rely-Guarantee based simulation
  - General method for concurrent program refinement
  - Adapted for linearizability proof
    - Formally justifies the LP method
    - Used to verify Treiber’s stack

- Needs to know statically LP point
  - Does not support non-fixed LP as in HSY stack
T1: push(v)

T2: pop()

T1 finishes not only its own opr, but also T2’s
HSY Stack Implementation

T1: push(v)

What's the problem?
The LP of pop may not even be inside the method body.

T2: pop()

LP corresponds to env. Step, not supported in previous sim.

T1 finishes not only its own opr, but also T2's
Our Solution

- Parameterize RGSim with **pending threads** that might be helped by others

**T1:**
- `push(v)`

**Elimination Array**

**T2:**
- `pop()`

**pending thread pool U**
- Derived from concrete state

- T1’s step may fulfill opr of T1 or threads in U
- T2 checks U to see if its opr has been done (by env)
Our New Simulation $O \preceq \Theta S$

$\left( U_0 \cup A_0, \Sigma_0 \right) \rightarrow^* \left( U_1 \cup A_1, \Sigma_1 \right) \rightarrow^* \left( U_2 \cup A_2, \Sigma_2 \right) \rightarrow^* \left( U_3 \cup A_3, \Sigma_3 \right)$

$\preceq_\Theta$ $\preceq_\Theta$ $\preceq_\Theta$ $\preceq_\Theta$

$\left( O_0, \sigma_0 \right) \rightarrow \left( O_1, \sigma_1 \right) \rightarrow \left( O_1, \sigma_2 \right) \rightarrow \left( O_2, \sigma_3 \right)$

Env. steps

U maps a thread ID t to a pending opr. A or •

$U_i \cup A_i = \Theta(\sigma_i)$

Support non-fixed LPs: thread’s LP can be in either $\longrightarrow$ or $\longleftarrow$
\[
(U \cup \text{push}(v), \Sigma) \xrightarrow{\cdot} (U \cup \{\cdot\}, \Sigma') \xrightarrow{\cdot} (U' \cup \{\cdot\}, \Sigma'')
\]

\[
\sigma \xrightarrow{T1 \text{ update the slot}} \sigma'
\]

\[
U(T2) = \text{pop}
\]

\[
U'(T2) = \cdot
\]

Elimination Array

T1: push(v)

T2: pop()
Soundness:

If $O \preceq_{\Theta} A$ for some $\Theta$, then $O \leq_{\text{lin}} A$

What’s more:

Hoare-style syntactic logic for lin.

http://home.ustc.edu.cn/~lhj1018/lin.pdf
Conclusion

• A new simulation $O \preceq A$ for linearizability proof
  – Sim $\rightarrow$ refinement $\rightarrow$ linearizability
  – Supports non-fixed LP
    • HSY elimination-based stack, lazy set, etc
    • First refinement-based proof for HSY elimination-based stack
  – A program logic for syntactic verification

• Another dimension of complexity
  – LP depends on future
  – May need backward simulation (leave as future work)
Thank you!