HIGHER-ORDER LINEARISABILITY

Andrzej Murawski
University of Warwick

Nikos Tzevelekos
Queen Mary
University of London
Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

A concurrent object is a data object shared by concurrent processes. Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object’s operations can be given by pre- and post-conditions. This paper defines linearizability, compares it to other correctness conditions, presents and demonstrates a method for proving the correctness of implementations, and shows how to reason about concurrent objects, given they are linearizable.

\[ m : \text{unit} \rightarrow \text{unit} \quad m : \text{int} \rightarrow \text{int} \]
LINEARISABILITY

- correctness criterion for concurrent libraries
- enables proofs of conformance to (sequential) specifications
- based on restricted rearrangements of actions in histories
Concurrent library behaviour

Consider a queue library \( L \), invoked concurrently, with methods:

- \( \text{enqueue}: \text{int} \rightarrow \text{void} \)
- \( \text{dequeue}: \text{void} \rightarrow \text{int} \)

A (correct) library behaviour is the following:

1. \( \text{call } \text{dq}() \)
2. \( \text{ret } \text{dq}(1) \)
3. \( \text{call } \text{nq}(1) \)
4. \( \text{ret } \text{nq}() \)
5. \( \text{call } \text{nq}(2) \)
6. \( \text{ret } \text{nq}() \)
7. \( \text{call } \text{deq}() \)
8. \( \text{ret } \text{deq}(1) \)

Threads:

- library \( L \) / client \( K \)

- \( \text{m} : \text{unit} \rightarrow \text{unit} \)
- \( \text{m} : \text{int} \rightarrow \text{int} \)

- \( \text{call } \text{m}(\text{m}_1, \text{m}_2) \)
- \( \text{ret } \text{m}_1(\text{m}_3) \)
- \( \text{call } \text{m}_1(\text{···}) \)

Def. History \( h_1 \) linearises to \( h_2 \) if \( h_2 \) can be obtained from \( h_1 \) by a series of /-transformations.

Def. Library \( L \) linearises to specification \( S \) if every history \( h_1 \) of \( L \) linearises to some \( h_2 \) from \( S \).
Consider a queue library \( L \), invoked concurrently, with methods:

- **enqueue**: \( \text{int} \rightarrow \text{void} \)
- **dequeue**: \( \text{void} \rightarrow \text{int} \)

A (correct) library behaviour is the following:

- **thread 1**: \( \text{call } nq(1) \quad \text{ret } nq() \)
- **thread 2**: \( \text{call } nq(2) \quad \text{ret } nq() \)
- **thread 3**: \( \text{call } dq() \quad \text{ret } dq(1) \)

The library \( L \) is a client of \( K \).
Consider a queue library $L$, invoked concurrently, with methods:

- **enqueue**: int $\rightarrow$ void
- **dequeue**: void $\rightarrow$ int

A (correct) library behaviour is the following:

1. Call $dq()$; return $dq(1)$
2. Call $nq(2)$; return $nq()$
3. Call $nq(1)$; return $nq()$
FIRST-ORDER LINEARISABILITY

$t \neq t'$

\[ \cdots (t, \text{call } m(v)) (t', x') \cdots \triangleleft \cdots (t', x') (t, \text{call } m(v)) \cdots \]

\[ \cdots (t', x') (t, \text{ret } m(v)) \cdots \triangleleft \cdots (t, \text{ret } m(v)) (t', x') \cdots \]
INGREDIENTS

• histories
• sequential histories
• linearisability
• correctness
HIGHER-ORDER LIBRARIES

• higher-order routines (public methods), e.g. HO queue

• higher-order parameters (abstract methods), e.g. HO queue with parameter
Hi gher-order library

Parameter library

Client

$\Theta$, $\Theta''$, $\Theta'$ are sets of method names with their types

types can be of higher order (i.e. general function types)

$M_1, \ldots, M_N$ are terms/programs running in parallel

write

$L : \Theta \rightarrow \Theta'$
Quarantining Weakness
Compositional Reasoning under Relaxed Memory Models
(Extended Abstract)

Radha Jagadeesan\(^1\), Gustavo Petri\(^2\), Corin Pitcher\(^1\), and James Riely\(^1\)

\(^1\) DePaul University
\(^2\) Purdue University
Parameterised Linearisability

Andrea Cerone\textsuperscript{1}, Alexey Gotsman\textsuperscript{1}, and Hongseok Yang\textsuperscript{2}

\textsuperscript{1} IMDEA Software Institute
\textsuperscript{2} University of Oxford
We now look at the concrete syntax of libraries and clients. Libraries comprise collections of typed
where each function call in
typed, i.e. names for methods of type
type respectively variables, methods and references.
must guarantee mutual exclusion) and only one lock acquisition will suffice to process one array of
combiner of all registered requests. Note that the requests will be attended to one after another (thus
Tests Blocks Terms Clients

Libraries $L ::= B \mid \text{abstract } m; L \mid \text{public } m; L$

Blocks $B ::= \epsilon \mid m = \lambda x.M; B \mid r := \lambda x.M; B \mid r := i; B$

Terms $M ::= (\mid i \mid t_{id} \mid x \mid m \mid M \oplus M \mid \langle M, M \rangle \mid \pi_1 M \mid \pi_2 M \mid \text{if } M \text{ then } M \text{ else } M$

$\mid \lambda x^\theta.M \mid xM \mid mM \mid \text{let } x = M \text{ in } M \mid r := M \mid !r$

Values $v ::= (\mid i \mid m \mid \langle v, v \rangle$

Higher-order libraries
Parameter library

\[\Theta, \Theta', \Theta''\]

Library

\[L, L', \Theta, \Theta', \Theta''\]

Client

\[K, M_1, \ldots, M_N\]

- \(\Theta, \Theta', \Theta''\) are sets of method names with their types.
- Types can be of higher order (i.e., general function types).
- \(M_1, \ldots, M_N\) are terms/programs running in parallel.
HIGHER-ORDER TRACES

• (abstract) interactions of the library with its context
• design guided by game semantics (O-context, P-library)
• example history

(1, call m(m_1, m_2))_O \quad (2, call m(\cdots))_O \quad (2, call m_1(\cdots))_P \quad (2, ret m_1(m_3))_O

sequential histories = alternating histories
(INT) \((E, M, R, P, A, S) \rightarrow_t (E, M', R', P, A, S')\), given that \((M, R, S) \rightarrow_t (M', R', S')\) and \(\text{dom}(R' \setminus R)\) consists of names that do not occur in \(E, A\).

(PQY) \((E, E[mv], R, P, A, S) \xrightarrow{\text{call } m(v')_PY} (m :: E :: E, -, R', P, A, S)\), given \(m \in A_Y\) and (PC).

(OQY) \((E, -, R, P, A, S) \xrightarrow{\text{call } m(v)_{OY}} (m :: E, M[v/x], R, P, A', S)\), given \(m \in P_Y, R(m) = \lambda x. M\) and (OC).

(PAY) \((m :: E, v, R, P, A, S) \xrightarrow{\text{ret } m(v')_PY} (E, -, R', P, A, S)\), given \(m \in P_Y\) and (PC).

(OAY) \((m :: E :: E, -, R, P, A, S) \xrightarrow{\text{ret } m(v)_{OY}} (E, E[v], R, P, A', S)\), given \(m \in A_Y\) and (OC).

(PC) If \(v\) contains the names \(m_1, \ldots, m_k\) then \(v' = v\{m'_i/m_i \mid 1 \leq i \leq k\}\) with each \(m'_i\) being a fresh name. Moreover, \(R' = R \cup \{m'_i \mapsto \lambda x.m_ix \mid 1 \leq i \leq k\}\) and \(P' = P \cup Y \{m'_1, \ldots, m'_k\}\).

(OC) If \(v\) contains names \(m_1, \ldots, m_k\) then \(m_i \in \phi(P, A)\), for each \(i\), and \(A' = A \cup Y \{m_1, \ldots, m_k\}\).

**Figure 6** Trace semantics rules. The rule (INT) is for embedding internal rules. In the rule (PQY), the library \((P)\) calls one of its abstract methods (either the original ones or those acquired via interaction), while in (PAY) it returns from such a call. The rules (OQY) and (OAY) are dual and represent actions of the context. In all of the rules, whenever we write \(m(v)\) or \(m(v')\), we assume that the type of \(v\) matches the argument type of \(m\).
HIGHER-ORDER LINEARISABILITY

$\text{def.} \quad \text{History } h_1 \text{ linearises to } h_2 \text{ if } h_2 \text{ can be obtained from } h_1 \text{ by a series of } \prec \text{-transformations.}$

Def. A library $L$ linearises to specification $S$ if every history $h_1$ of $L$ linearises to some $h_2$ from $S$. 
higher-order linearisability

We consider a library written in ML-like syntax which implements a multiset data structure with
the shape :

\[
\begin{align*}
\text{count} & : \text{int} \rightarrow \text{int}, & \quad \text{update} & : (\text{int} \times (\text{int} \rightarrow \text{int})) \rightarrow \text{int}
\end{align*}
\]

```plaintext
public count, update;
Lock lock;
F := \lambda x.0;

\text{count} = \lambda i. (\!F)i
\text{update} = \lambda(i, g). \text{aux}(i, g, \text{count} i)

\text{aux} = \lambda(i, g, j).
  let y = |g j| in
  lock.acquire();
  let f = \!F in
  if (j == (f i)) then {
    F := \lambda x. if (x == i) then y
      else (f x);
    lock.release();
    y }
  else {
    lock.release();
    aux(i, g, f i) }
```

http://c-cube.github.io/ocaml-containers/0.21/CCMultiSet.S.html
ENCAPSULATED CASE

$\Theta, \Theta'$ are sets of method names with their types ($L : \Theta \rightarrow \Theta'$).

Types can be of higher order (i.e. general function types).

$M_1, \ldots, M_N$ are terms/programs running in parallel.

[Cerone, Gotsman, Yang '14]
Encapsulation: Splitting the Opponent

- $\Theta, \Theta'$ are sets of method names with their types ($\Theta : \Theta' \rightarrow \Theta$)
- Types can be of higher order (i.e. general function types)
- $M_1, \ldots, M_N$ are terms/programs running in parallel
- Separate Opponent into $O_K$ and $O_L$ (dually for Proponent)

$\Theta$ is more constrained + more transpositions are legal

---

Parameter library

Library

Client
LINEARISABILITY
(ENCAPSULATED)

\[ t \neq t' \]

\[ \cdots (t, x)_K (t', x')_L \cdots \diamond \cdots (t', x')_L (t, x)_K \cdots \]

\[ \cdots (t', x')_L (t, x)_K \cdots \diamond \cdots (t, x)_K (t', x')_L \cdots \]

**Def.** History \( h_1 \) linearises to \( h_2 \) if \( h_2 \) can be obtained from \( h_1 \) by a series of \( \leftarrow \) and \( \diamond \)-transformations.
RELATIONAL LINEARISABILITY

- linearisability under additional assumptions about clients
- the extra constraints are expressed as closure under a relation $R$
- correctness with respect to clients whose behaviour (trace set) is $R$-closed
We now look at the concrete syntax of libraries and clients. Libraries comprise collections of typed
where each function call in
method declarations (public or abstract) followed by a block
names for references to methods of type
names for methods of type
typed, i.e.
to range over
capturing thread-blind client behaviour (see Appendix
of thread identifiers. This can be expressed through
because clients may call library methods with functional arguments that recognise thread identity.
developed by the library
guaranteeing mutual exclusion) and only one lock acquisition will suffice to process one array of
Terms

The syntax for libraries and clients is given in Figure 4. Each library

\[
\begin{align*}
\text{run} & = \lambda (f,x). \ (lock\ .\ acquire\ ();\ \ \text{let}\ \ result = f(x)\ \text{in}\ \ lock\ .\ release\ ();\ \ result)
\end{align*}
\]
public run; 

Lock lock;

struct {fun, arg, wait, retv} requests [N];

run = λ (f, x).

requests [t_id].fun := f;
requests [t_id].arg := x;
requests [t_id].wait := 1;

while (requests [t_id].wait)
    if (lock . tryacquire ()) {
        for (t=0; t<N; t++)
            if (requests [t].wait) {
                requests [t].retv :=
                requests [t].fun (requests [t].arg);
                requests [t].wait := 0;
            }
        lock . release ()
    }
requests [t_id].retv;
SUMMARY

- new framework for higher-order linearisability in various cases (general, encapsulated, relational)

- soundness and compositionality

- case studies

- main target for future work:
  proof techniques for higher-order linearisability