Higher-Order Linearisability

Andrzej Murawski
University of Warwick

Nikos Tzevelekos
Queen Mary U. of London

[ paper appeared at CONCUR’17 ]

Highlights of Logic, Games and Automata, September 2017
Concurrent libraries

Concurrent *queue* library \( L \) with methods:

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

Library \( L : \Theta \), client \( K \) (multi-threaded)

\[ \Theta = \{ \text{\textbf{nq}: int } \to \text{ void, \textbf{dq}: void } \to \text{ int} \} \]
Concurrent queue behaviour

Concurrent *queue* library $L$ with methods:

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

A (correct) library behaviour:

threads

<table>
<thead>
<tr>
<th>Time</th>
<th>Library Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>call <strong>nq(1)</strong></td>
</tr>
<tr>
<td></td>
<td>ret <strong>nq()</strong></td>
</tr>
<tr>
<td>2</td>
<td>call <strong>nq(2)</strong></td>
</tr>
<tr>
<td></td>
<td>ret <strong>nq()</strong></td>
</tr>
<tr>
<td>3</td>
<td>call <strong>dq()</strong></td>
</tr>
<tr>
<td></td>
<td>ret <strong>dq(1)</strong></td>
</tr>
</tbody>
</table>

library $L / \text{client } K$
Concurrent queue behaviour

Concurrent queue library $L$ with methods:

- **enqueue**: int → void
- **dequeue**: void → int

A (correct) library behaviour:

<table>
<thead>
<tr>
<th>threads</th>
<th>library $L$ / client $K$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>call dq() ret dq(1)</td>
</tr>
<tr>
<td>2</td>
<td>call nq(2) ret nq()</td>
</tr>
<tr>
<td>1</td>
<td>call nq(1) ret nq()</td>
</tr>
</tbody>
</table>
Concurrent queue behaviour

Concurrent *queue* library $L$ with methods:

- **enqueue**: int → void
- **dequeue**: void → int

A (correct) library behaviour:

```
threads

1. call nq(1)  ret nq()
2. call nq(2)  ret nq()
3. call dq()    ret dq(1)
```

library $L$ / client $K$
Concurrent queue behaviour

Concurrent *queue* library $L$ with methods:

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

A (correct) library behaviour:

 threads

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>call $\text{nq}(1)$</td>
<td>call $\text{nq}(2)$</td>
<td>call $\text{dq}()$</td>
<td>ret $\text{nq}()$</td>
</tr>
</tbody>
</table>
Concurrent queue behaviour

Concurrent queue library $L$ with methods:

$\text{enqueue}: \text{int} \rightarrow \text{void}$  \hspace{1cm} $\text{dequeue}: \text{void} \rightarrow \text{int}$

A (correct) library behaviour:

threads  \hspace{1cm} library $L$ / client $K$

3  

\begin{align*}
\text{call } dq() & \hspace{1cm} \text{ret } dq(1) \\
\text{call } nq(2) & \hspace{1cm} \text{ret } nq() \\
\end{align*}

2

\begin{align*}
\text{call } nq(1) & \hspace{1cm} \text{ret } nq() \\
\end{align*}

1
Correctness formally

How can we characterise “correct” behaviours?
→ use linearisability:

*a behaviour is correct if it follows a given specification, modulo legal reordering of actions:*

\[
(t, \text{call}) (t', x) \rightarrow (t', x) (t, \text{call}) \\
(t', x) (t, \text{ret}) \rightarrow (t, \text{ret}) (t', x)
\]

Correctness formally

How can we characterise “correct” behaviours?

→ use linearisability:

*a behaviour is correct if it follows a given specification, modulo legal reordering of actions:*

\[(t, \text{call}) (t', x) \rightarrow (t', x) (t, \text{call})\]

\[(t', x) (t, \text{ret}) \rightarrow (t, \text{ret}) (t', x)\]

→ introduced in the 90’s, by now standard

→ soundness: \(L\) linearises into \(L’\) \(\Rightarrow\) \(L\) obs. approximates \(L’\)

[Herlihy & Wing '90]

[Filipovic, O’Hearn, Rinetzky, Yang '10]
Higher-order libraries

Library $L$ is open (depends on abstract methods $\Theta$)

Methods are higher order (e.g. $\text{int} \rightarrow \text{int} \rightarrow \text{int}$)

Write $L : \Theta \rightarrow \Theta'$

Previously only $1^{\text{st}}$ order methods examined

[Jagadeesan, Petri, Pitcher, Riely ’13] [Cerone, Gotsman & Yang ’14]
E.g. multiset library:

```plaintext
public count, update;
Lock lock;
F := λx.0;

count = λi. (!F)i
update = λ(i, g). aux(i, g, count i)

aux = λ(i, g, j).
    let y = |g j| in
    lock.acquire();
    let f = !F in
    if (j == (f i)) then {
        F := λx. if (x == i) then y
        else (f x);
        lock.release();
y    }
else {
    lock.release();
aux(i, g, f i )
```
Higher-order difficulties

The HO situation is much richer as traces are higher-order:

\[(1, \text{call } \overline{m}_1(\overline{m}_1, \overline{m}_2)) \ (2, \text{call } \overline{m}_2(...)) \ (2, \text{call } \overline{m}_1(...)) \ (2, \text{ret } \overline{m}_1(\overline{m}_3)) \ ...\]

- a method call need not be followed by its return
- method calls/returns can be issued by everyone
  (the client, the library, the parameter library)
- new methods can appear on-the-fly
- sequential histories?

The setting is great for game semantics!
What game semantics

Computation modelled as a 2-player game between

- **Opponent** (the environment), aka $O$
- **Proponent** (the program), aka $P$

Moves are method calls and returns

Programs = strategies for $P$

\[
\begin{align*}
\text{int } m \ (f : \text{int } \to \text{int}, \ x: \text{int}) &= \text{return } f(x) + 1 \\
\Rightarrow \quad \text{call } m(f, 5)_{O} \quad \text{call } f(5)_{P} \quad \text{ret } f(41)_{O} \quad \text{ret } m(42)_{P}
\end{align*}
\]

(formulated operationally as “HO open trace semantics”)
Higher-order linearisability

Legal reorderings now defined by:

\[(t, a_O) (t', a') \prec_{PO} (t', a') (t, a_O)\]

\[(t', a') (t, a_P) \prec_{PO} (t, a_P) (t', a')\]

\[P (Proponent: L) \quad O (Opponent: K & L')\]
Higher-order linearisability

Legal reorderings now defined by:

\[
(t, a_O) (t', a') \prec_{PO} (t', a') (t, a_O) \\
(t', a') (t, a_P) \prec_{PO} (t, a_P) (t', a')
\]

**Def:** History \( h_1 \) **linearises** to \( h_2 \) if we can get \( h_2 \) from \( h_1 \) by a series of \( \prec_{PO} \)-reorderings.

A library \( L \) linearises to some **specification** \( A \) of sequential histories if every history \( h_1 \) of \( L \) linearises into some \( h_2 \) in \( A \).

A history \( h \) is **sequential** if it is of the form:

\[
h = (t_1, a_{1(O)}) (t_1, a_{2(P)}) (t_2, a_{3(O)}) (t_2, a_{4(P)}) \ldots
\]
Results

Soundness

\[ L_1 \text{ linearises into } L_2 \implies L_1 \text{ obs. approximates } L_2 \]

i.e. for all \((L', K)\):

\((\text{link } L'; L_1 \text{ in } K) \text{ terminates} \implies (\text{link } L'; L_2 \text{ in } K) \text{ terminates}\)

Compositionality

\[ L_1 \text{ linearises into } L_2 \implies L_1 + L \text{ linearises into } L_2 + L \]
Encapsulation

Environment more restricted

\[ (t, a)_\kappa (t', a)_\ell \quad \Diamond \quad (t', a)_\ell (t, a)_\kappa \]
\[ (t, a)_\ell (t', a)_\kappa \quad \Diamond \quad (t', a)_\kappa (t, a)_\ell \]

[Cerone, Gotsman, Yang '14]
Summary

Introduce extension of linearisability for higher-order libraries

Prove it is sound and compositional

Encapsulation, relational linearisability

Examples

Further on:

- weaker notions, allowing for context-specific linearisations (cf. encapsulation)
- completeness?
- proof methods for linearisability
Flat combining (relational linearisability)

```java
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;
            }
    } else
        lock.release ()
    requests [t_id].retv;
```