Logical relations for fine-grained concurrency

Aaron Turon

with
Jacob Thamsborg, Amal Ahmed, Lars Birkedal, Derek Dreyer
Concurrency is overlapped execution:

- Process A
- Process B
Concurrency is overlapped execution:

Process A

Process B
Concurrency is overlapped execution:

Process A

Process B

Atomicity is its antidote:

Process A

Process B
Granularity (in time & space)

**Coarse-grained** atomicity:
- easier reasoning

**Fine-grained** atomicity:
- lower latency
- higher throughput (if parallel)
Atomicity abstraction

Fine-grained implementation

Coarse-grained specification
Atomicity abstraction

Fine-grained implementation ≤ Coarse-grained specification

Refines, i.e., contextually approximates
Contributions

*Direct, insightful* refinement method
(no linearizability!)

Handles *higher-order, polymorphic* languages

Scales to *sophisticated* algorithms
Key Idea

Atomicity abstraction, like data abstraction, relies on hiding
Approach

Transition systems for hidden state

} State of the art: Kripke Logical Relations, ICFP10
Approach

Transition systems for hidden state → Local protocols

State of the art: Kripke Logical Relations, ICFP'10

Fine granularity
Approach

Transition systems for hidden state

Local protocols

Tokens

Computations as resources

Speculation

State of the art: Kripke Logical Relations, ICFP10

Fine granularity

Sophisticated algorithms
Approach

Transition systems for hidden state

Local protocols

Tokens

Computations as resources

Speculation

State of the art: Kripke Logical Relations, ICFP10

Fine granularity

Sophisticated algorithms
Compare and set

\[
\text{cas(cell, old, new)} = \text{atomic}\ {\{\\}
  \text{if } \neg \text{cell} := \text{old} \\
  \text{then } \text{cell} := \text{new}; \text{true} \\
  \text{else } \text{false}
\}\}
\]
A Lock-free “Queue”
A Lock-free “Queue”

Head

2
A Lock-free “Queue”

Head  

2

\text{CAS(Head, null, } n)\text{)
A Lock-free “Queue”

Head 🗔️
A Lock-free “Queue”
A Lock-free “Queue”
A Lock-free “Queue”
A Lock-free "Queue"
A Lock-free "Queue"
A Lock-free “Queue”
A Lock-free “Queue”
A Lock-free “Queue”
A Lock-free “Queue”
A Lock-free "Queue"
A Lock-free “Queue”
A Lock-free “Queue”

Head

7  →  11
A Lock-free “Queue”
A Lock-free “Queue”

Head

7 → 11 → 13
A Lock-free “Queue”

Head

3 X
A Lock-free “Queue”
A Lock-free “Queue”

Head  

3  →  2  

[Diagram of a ladybug]
The Michael-Scott Queue

Head \[ \rightarrow \] Sentinel
The Michael-Scott Queue

Head → Sentinel

2 → CAS
The Michael-Scott Queue

Head → Sentinel

2 → X
The Michael-Scott Queue
The Michael-Scott Queue

Head → 2 → Sentinel → 3
The life story of a node
The life story of a node

Global constraints:
Exactly one sentinel
Live iff reachable from sentinel
Nonnull next pointers are born
The life story of a node

Global constraints:
Exactly one sentinel
Live iff reachable from sentinel
Nonnull next pointers are born

linkNode(p, n) = case !(p.next) of null ⇒
   if cas(p.next, null, n)
   then () else linkNode(p, n)
| p' ⇒ linkNode(p', n)
The life story of a node

Global constraints:
Exactly one sentinel
Live iff reachable from sentinel
Nonnull next pointers are born

\[
\text{linkNode}(p, n) = \begin{cases} 
\text{case } !\!(p.\text{next}) \text{ of null } \Rightarrow \\
\quad \text{if } \text{cas}(p.\text{next}, \text{null}, n) \\
\quad \text{then } () \text{ else } \text{linkNode}(p, n) \\
\quad | \quad p' \Rightarrow \text{linkNode}(p', n)
\end{cases}
\]
The life story of a node

Global constraints:
Exactly one sentinel
Live iff reachable from sentinel
Nonnull next pointers are born

```
linkNode(p, n) = case !(p.next)
of null ⇒
  if cas(p.next, null, n)
  then () else linkNode(p, n)
| p' ⇒ linkNode(p', n)
```
The life story of a node

**Global constraints:**
- Exactly one sentinel
- Live iff reachable from sentinel
- Nonnull next pointers are born

```plaintext
linkNode(p, n) = case !(p.next) of null =>
  if cas(p.next, null, n)
  then () else linkNode(p, n)
  p' => linkNode(p', n)
```
The life story of a node

Global constraints:
Exactly one sentinel
Live iff reachable from sentinel
Nonnull next pointers are born

linkNode(p, n) = case !(p.next) of null ⇒
  if cas(p.next, null, n)
  then () else linkNode(p, n)
  | p' ⇒ linkNode(p', n)
Key Idea

Local protocols capture abstract knowledge/interference at the same granularity as the algorithm operates
Approach

Transition systems for hidden state

Local protocols

Tokens

Computations as resources

Speculation

State of the art: Kripke Logical Relations, ICFP10

Fine granularity

Sophisticated algorithms
Approach

Transition systems for hidden state

Local protocols

Tokens

Computations as resources

Speculation

State of the art: Kripke Logical Relations, ICFP 10

Fine granularity

Sophisticated algorithms
Approach

Transition systems for hidden state

Local protocols

Tokens

Computations as resources

Speculation

State of the art: Kripke Logical Relations, ICFP10

Fine granularity

Sophisticated algorithms
Conservation Law

Before
\[ T \oplus P = T' \oplus P' \]

After

Thread’s tokens

Protocol’s tokens

Unlocked \{ ♠ \}

Locked \∅
Conservation Law

Before
\[ T \cup P = T' \cup P' \]

Thread’s tokens
Protocol’s tokens

After

Unlocked \{♦\} → Locked: \[ \emptyset \cup \{♦\} = ? \cup \emptyset \]
Conservation Law

Before

\[ T \cup P = T' \cup P' \]

Thread's tokens

Protocol's tokens

After

Unlocked \{ \diamond \} \rightarrow \text{Locked } \emptyset

\[ \emptyset \cup \{ \diamond \} = T' \cup P' \]

Unlocked \rightarrow \text{Locked: } \emptyset \cup \{ \diamond \} = T' \cup P' \]
Conservation Law

Before: \[ T \cup P = T' \cup P' \]

Thread's tokens \hspace{0.5cm} Protocol's tokens

Unlocked \[ \{ \Diamond \} \] \hspace{1cm} Locked \[ \emptyset \]

Unlocked \rightarrow Locked: \[ \emptyset \cup \{ \Diamond \} = \{ \Diamond \} \cup \emptyset \]
Conservation Law

Before  
\[ T \cup P = T' \cup P' \]

After

Unlocked  \{ \diamond \}  Locked  \emptyset

Thread’s tokens  Protocol’s tokens

Unlocked \rightarrow\: Locked:  \emptyset \cup \{ \diamond \} = \{ \diamond \} \cup \emptyset

Locked \rightarrow\: Unlocked:  \emptyset \cup \emptyset = ? \cup \{ \diamond \}
Conservation Law

Before

\[ T \cup P = T' \cup P' \]

Thread’s tokens

Protocol’s tokens

After

Unlocked \{ \diamond \} → Locked: \( \emptyset \cup \{ \diamond \} = \{ \diamond \} \cup \emptyset \)

Locked → Unlocked: \( \emptyset \cup \emptyset = \emptyset \cup \{ \diamond \} \) Impossible
Key Idea

Tokens enable threads to gain and lose roles *dynamically*, via transitions.
Key Idea

**Tokens** enable threads to gain and lose roles *dynamically*, via transitions.

*Rely:* transitions with environment tokens

*Guarantee:* transitions with thread’s tokens
Approach

Transition systems for hidden state

Local protocols

Tokens

Computations as resources

Speculation

✓

State of the art: Kripke Logical Relations, ICFP10

Fine granularity

Sophisticated algorithms
Approach

Transition systems for hidden state

Local protocols

Tokens

Computations as resources

Speculation

✓

✓

✓

State of the art: Kripke Logical Relations, ICFP10

Fine granularity

Sophisticated algorithms
Approach

Transition systems for hidden state

Local protocols

Tokens

Computations as resources

Speculation

State of the art: Kripke Logical Relations, ICFP 10

Fine granularity

Sophisticated algorithms
**Problem:**
Cooperation is inherently non-thread-local
e.g. elimination stacks

**Solution:**
Make thread’s specification a *shareable resource*
**Problem:**
Cooperation is inherently non-thread-local
e.g. elimination stacks

**Solution:**
Make thread’s specification a *shareable resource*
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Local protocols</th>
<th>Tokens</th>
<th>Computational resources</th>
<th>Speculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Treiber’s Stack</td>
<td>X</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Michael-Scott Queue</td>
<td>X</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Hand-over-hand set</td>
<td>X</td>
<td>X</td>
<td></td>
<td>★</td>
</tr>
<tr>
<td>Elimination Flags</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Lazy set</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>★</td>
</tr>
<tr>
<td>Conditional CAS</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>K-CAS</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>★</td>
</tr>
</tbody>
</table>
The Takeaway

Analyze fine-grained algorithms with fine-grained protocols!

To scale up, must address:
- varying roles
- cooperation
- nondeterminism

Our contributions:
- tokens
- computational resources
- speculation