Impredicative Concurrent Abstract Predicates

Kasper Svendsen, Lars Birkedal
Aarhus University

September 28, 2013

Introduction

Goal

- A logic for modular reasoning about partial correctness of concurrent, higher-order, reentrant, imperative code.

Introduction

Goal

- A logic for modular reasoning about partial correctness of concurrent, higher-order, reentrant, imperative code.

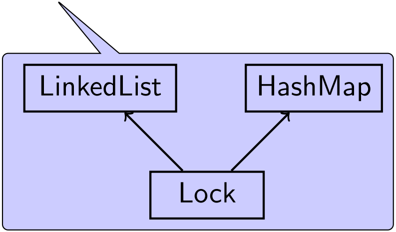
- Modular library specifications that supports **layering** of abstractions and **recursive** abstractions.

Introduction

Goal

- A logic for modular reasoning about partial correctness of concurrent, higher-order, reentrant, imperative code.

- Modular library specifications that supports **layering** of abstractions and **recursive** abstractions.

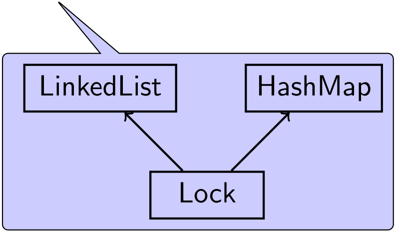


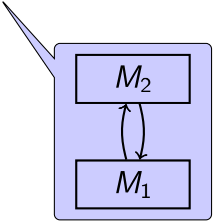
Introduction

Goal

- A logic for modular reasoning about partial correctness of concurrent, higher-order, reentrant, imperative code.

- Modular library specifications that supports **layering** of abstractions and **recursive** abstractions.





A Modular Lock Specification

$$\exists \mathsf{isLock}, \mathsf{locked} : \mathsf{Val} \times \mathsf{Prop} \to \mathsf{Prop}.\ \forall \mathsf{R} : \mathsf{Prop}.\ \mathsf{stable}(\mathsf{R}) \Rightarrow$$

$$\begin{array}{rcl} \{\mathsf{R}\} & \texttt{new Lock()} & \{\mathsf{isLock}(\mathsf{ret}, \mathsf{R})\} \\ \{\mathsf{isLock}(\mathsf{x}, \mathsf{R})\} & \texttt{x.Acquire()} & \{\mathsf{locked}(\mathsf{x}, \mathsf{R}) * \mathsf{R}\} \\ \{\mathsf{locked}(\mathsf{x}, \mathsf{R}) * \mathsf{R}\} & \texttt{x.Release()} & \{\mathsf{isLock}(\mathsf{x}, \mathsf{R})\} \end{array}$$

$$\forall \mathsf{x} : \mathsf{Val}.\ \mathsf{isLock}(\mathsf{x}, \mathsf{R}) \Leftrightarrow \mathsf{isLock}(\mathsf{x}, \mathsf{R}) * \mathsf{isLock}(\mathsf{x}, \mathsf{R})$$

$$\forall \mathsf{x} : \mathsf{Val}.\ \mathsf{stable}(\mathsf{isLock}(\mathsf{x}, \mathsf{R})) \wedge \mathsf{stable}(\mathsf{locked}(\mathsf{x}, \mathsf{R}))$$

A Modular Lock Specification

Standard sep. logic lock specification
The resource invariant R describes the resources protected by the lock.

$$\exists \mathsf{isLock}, \mathsf{locked} : \mathsf{Val} \times \mathsf{Prop} \rightarrow \mathsf{Prop}.\ \forall \mathsf{R} : \mathsf{Prop}.\ \mathsf{stable}(\mathsf{R}) \Rightarrow$$

$$\begin{array}{rcl} \{\mathsf{R}\} & \texttt{new Lock()} & \{\mathsf{isLock}(\mathsf{ret}, \mathsf{R})\} \\ \{\mathsf{isLock}(\mathsf{x}, \mathsf{R})\} & \texttt{x.Acquire()} & \{\mathsf{locked}(\mathsf{x}, \mathsf{R}) * \mathsf{R}\} \\ \{\mathsf{locked}(\mathsf{x}, \mathsf{R}) * \mathsf{R}\} & \texttt{x.Release()} & \{\mathsf{isLock}(\mathsf{x}, \mathsf{R})\} \end{array}$$

$$\forall \mathsf{x} : \mathsf{Val}.\ \mathsf{isLock}(\mathsf{x}, \mathsf{R}) \Leftrightarrow \mathsf{isLock}(\mathsf{x}, \mathsf{R}) * \mathsf{isLock}(\mathsf{x}, \mathsf{R})$$

$$\forall \mathsf{x} : \mathsf{Val}.\ \mathsf{stable}(\mathsf{isLock}(\mathsf{x}, \mathsf{R})) \wedge \mathsf{stable}(\mathsf{locked}(\mathsf{x}, \mathsf{R}))$$

A Modular Lock Specification

$$\exists \mathsf{isLock}, \mathsf{locked} : \mathsf{Val} \times \mathsf{Prop} \to \mathsf{Prop}.\ \forall R : \mathsf{Prop}.\ \mathsf{stable}(R) \Rightarrow$$

$$\begin{array}{rcl} \{R\} & \texttt{new Lock()} & \{\mathsf{isLock}(\mathsf{ret}, R)\} \\ \{\mathsf{isLock}(x, R)\} & \texttt{x.Acquire()} & \{\mathsf{locked}(x, R) * R\} \\ \{\mathsf{locked}(x, R) * R\} & \texttt{x.Release()} & \{\mathsf{isLock}(x, R)\} \end{array}$$

$$\forall x : \mathsf{Val}.\ \mathsf{isLock}(x, R) \Leftrightarrow \mathsf{isLock}(x, R) * \mathsf{isLock}(x, R)$$

$$\forall x : \mathsf{Val}.\ \mathsf{stable}(\mathsf{isLock}(x, R)) \wedge \mathsf{stable}(\mathsf{locked}(x, R))$$

A Modular Lock Specification

Third-order quantification.

$$\exists \mathsf{isLock}, \mathsf{locked} : \mathsf{Val} \times \mathsf{Prop} \to \mathsf{Prop}.\ \forall R : \mathsf{Prop}.\ \mathsf{stable}(R) \Rightarrow$$

$$\begin{array}{rcl}
\{R\} & \texttt{new Lock()} & \{\mathsf{isLock}(\mathsf{ret}, R)\} \\
\{\mathsf{isLock}(x, R)\} & \texttt{x.Acquire()} & \{\mathsf{locked}(x, R) * R\} \\
\{\mathsf{locked}(x, R) * R\} & \texttt{x.Release()} & \{\mathsf{isLock}(x, R)\}
\end{array}$$

$$\forall x : \mathsf{Val}.\ \mathsf{isLock}(x, R) \Leftrightarrow \mathsf{isLock}(x, R) * \mathsf{isLock}(x, R)$$

$$\forall x : \mathsf{Val}.\ \mathsf{stable}(\mathsf{isLock}(x, R)) \wedge \mathsf{stable}(\mathsf{locked}(x, R))$$

A Modular Lock Specification

$$\forall \mathsf{R} : \mathsf{Prop}.\ \exists \mathsf{isLock}, \mathsf{locked} : \mathsf{Val} \to \mathsf{Prop}.\ \mathsf{stable}(\mathsf{R}) \Rightarrow$$

$$\begin{array}{rll} \{\mathsf{R}\} & \texttt{new Lock()} & \{\mathsf{isLock}(\mathsf{ret})\} \\ \{\mathsf{isLock}(\mathsf{x})\} & \texttt{x.Acquire()} & \{\mathsf{locked}(\mathsf{x}) * \mathsf{R}\} \\ \{\mathsf{locked}(\mathsf{x}) * \mathsf{R}\} & \texttt{x.Release()} & \{\mathsf{isLock}(\mathsf{x})\} \end{array}$$

$$\forall \mathsf{x} : \mathsf{Val}.\ \mathsf{isLock}(\mathsf{x}) \Leftrightarrow \mathsf{isLock}(\mathsf{x}) * \mathsf{isLock}(\mathsf{x})$$

$$\forall \mathsf{x} : \mathsf{Val}.\ \mathsf{stable}(\mathsf{isLock}(\mathsf{x})) \wedge \mathsf{stable}(\mathsf{locked}(\mathsf{x}))$$

A Modular Lock Specification

Second-order quantification.

$$\forall \mathsf{R} : \mathsf{Prop}.\ \exists \mathsf{isLock}, \mathsf{locked} : \mathsf{Val} \to \mathsf{Prop}.\ \mathsf{stable}(\mathsf{R}) \Rightarrow$$

$$\begin{array}{rcl} \{\mathsf{R}\} & \texttt{new Lock()} & \{\mathsf{isLock}(\mathsf{ret})\} \\ \{\mathsf{isLock}(\mathsf{x})\} & \texttt{x.Acquire()} & \{\mathsf{locked}(\mathsf{x}) * \mathsf{R}\} \\ \{\mathsf{locked}(\mathsf{x}) * \mathsf{R}\} & \texttt{x.Release()} & \{\mathsf{isLock}(\mathsf{x})\} \end{array}$$

$$\forall \mathsf{x} : \mathsf{Val}.\ \mathsf{isLock}(\mathsf{x}) \Leftrightarrow \mathsf{isLock}(\mathsf{x}) * \mathsf{isLock}(\mathsf{x})$$

$$\forall \mathsf{x} : \mathsf{Val}.\ \mathsf{stable}(\mathsf{isLock}(\mathsf{x})) \land \mathsf{stable}(\mathsf{locked}(\mathsf{x}))$$

A Modular Lock Specification

This specification might suffice for layering of abstractions, but **not** for all **recursive** abstractions.

$$\forall R : \mathsf{Prop}.\ \exists \mathsf{isLock}, \mathsf{locked} : \mathsf{Val} \to \mathsf{Prop}.\ \mathsf{stable}(R) \Rightarrow$$

$$\begin{array}{rcl} \{R\} & \texttt{new Lock()} & \{\mathsf{isLock}(\mathsf{ret})\} \\ \{\mathsf{isLock}(x)\} & \texttt{x.Acquire()} & \{\mathsf{locked}(x) * R\} \\ \{\mathsf{locked}(x) * R\} & \texttt{x.Release()} & \{\mathsf{isLock}(x)\} \end{array}$$

$$\forall x : \mathsf{Val}.\ \mathsf{isLock}(x) \Leftrightarrow \mathsf{isLock}(x) * \mathsf{isLock}(x)$$

$$\forall x : \mathsf{Val}.\ \mathsf{stable}(\mathsf{isLock}(x)) \wedge \mathsf{stable}(\mathsf{locked}(x))$$

Recursive Abstractions

Reentrant Event Loop Library

```
delegate void handler();

interface IEventLoop {
  void loop();
  void signal();
  void when(handler f);
}
```

Recursive Abstractions

Reentrant Event Loop Library

```
delegate void handler();

interface IEventLoop {
  void loop();
  void signal();
  void when(handler f);
}
```

Event handlers are allowed to emit events!

Recursive Abstractions

Reentrant Event Loop Library

A library that allows us to close Landin's Knot / perform recursion through the store.

```
delegate void handler();

interface IEventLoop {
  void loop();
  void signal();
  void when(handler f);
}
```

Event handlers are allowed to emit events!

Recursive Abstractions

Reentrant Event Loop Library

A library that allows us to close Landin's Knot / perform recursion through the store.

```
delegate void handler();

interface IEventLoop {
  void loop();
  void signal();
  void when(handler f);
}
```

Event handlers are allowed to emit events!

Realistic examples of this form:
libevent, Node.js, Twisted, ...
C5, GUI libraries, Joins library, ...

Recursive Abstractions

Event Loop Memory Safety Specification

$\exists \mathsf{eloop} : \mathsf{Val} \rightarrow \mathsf{Prop}.$

$$\begin{array}{rcl}
\{\mathsf{emp}\} & \texttt{new EventLoop()} & \{\mathsf{eloop}(\mathsf{ret})\} \\
\{\mathsf{eloop}(\mathsf{x})\} & \texttt{x.loop()} & \{\mathsf{eloop}(\mathsf{x})\} \\
\{\mathsf{eloop}(\mathsf{x})\} & \texttt{x.signal()} & \{\mathsf{eloop}(\mathsf{x})\} \\
\{\mathsf{eloop}(\mathsf{x}) * P\} & \texttt{x.when(f)} & \{\mathsf{eloop}(\mathsf{x})\}
\end{array}$$

$$\forall \mathsf{x} : \mathsf{Val}.\ \mathsf{eloop}(\mathsf{x}) \Leftrightarrow \mathsf{eloop}(\mathsf{x}) * \mathsf{eloop}(\mathsf{x})$$

where $P = f \mapsto \{\mathsf{emp}\}\{\mathsf{emp}\}$

Recursive Abstractions

Event Loop Memory Safety Specification

$\exists \mathsf{eloop} : \mathsf{Val} \to \mathsf{Prop}.$

$$
\begin{array}{rcl}
\{\mathsf{emp}\} & \texttt{new EventLoop()} & \{\mathsf{eloop}(\mathsf{ret})\} \\
\{\mathsf{eloop}(\mathsf{x})\} & \texttt{x.loop()} & \{\mathsf{eloop}(\mathsf{x})\} \\
\{\mathsf{eloop}(\mathsf{x})\} & \texttt{x.signal()} & \{\mathsf{eloop}(\mathsf{x})\} \\
\{\mathsf{eloop}(\mathsf{x}) * P\} & \texttt{x.when(f)} & \{\mathsf{eloop}(\mathsf{x})\}
\end{array}
$$

$$\forall \mathsf{x} : \mathsf{Val}.\ \mathsf{eloop}(\mathsf{x}) \Leftrightarrow \mathsf{eloop}(\mathsf{x}) * \mathsf{eloop}(\mathsf{x})$$

where $P = f \mapsto \{\mathsf{emp}\}\{\mathsf{emp}\}$

Event handler must run without any resources and emitting an event requires an eloop(x) resource!

Recursive Abstractions

Reentrant Event Loop Memory Safety Specification

$$\exists \mathsf{eloop} : \mathsf{Val} \to \mathsf{Prop}.$$

$$\begin{array}{rcl}
\{\mathsf{emp}\} & \texttt{new EventLoop()} & \{\mathsf{eloop}(\mathsf{ret})\} \\
\{\mathsf{eloop}(\mathsf{x})\} & \texttt{x.loop()} & \{\mathsf{eloop}(\mathsf{x})\} \\
\{\mathsf{eloop}(\mathsf{x})\} & \texttt{x.signal()} & \{\mathsf{eloop}(\mathsf{x})\} \\
\{\mathsf{eloop}(\mathsf{x}) * P\} & \texttt{x.when(f)} & \{\mathsf{eloop}(\mathsf{x})\}
\end{array}$$

$$\forall \mathsf{x} : \mathsf{Val}.\ \mathsf{eloop}(\mathsf{x}) \Leftrightarrow \mathsf{eloop}(\mathsf{x}) * \mathsf{eloop}(\mathsf{x})$$

where $P = f \mapsto \{\mathsf{eloop}(\mathsf{x})\}\{\mathsf{emp}\}$

Recursive Abstractions

Verifying a lock-based event loop implementation

- Since we are interested in memory safety, eloop has to specify the memory footprint of the event loop.

Recursive Abstractions

Verifying a lock-based event loop implementation

- Since we are interested in memory safety, eloop has to specify the memory footprint of the event loop.
- Since an event loop can call handlers, its footprint include the footprint of its handlers.
- Since handlers can signal events, the footprint of handlers include the footprint of their event loop.

Recursive Abstractions

Verifying a lock-based event loop implementation

- Since we are interested in memory safety, eloop has to specify the memory footprint of the event loop.
- Since an event loop can call handlers, its footprint include the footprint of its handlers.
- Since handlers can signal events, the footprint of handlers include the footprint of their event loop.
- The footprint of an event loop is thus recursively defined.

Recursive Abstractions

Verifying a lock-based event loop implementation

- Imagine an implementation that maintains a set of signal handlers and a set of pending signals, protected by a lock:

```
class EventLoop : IEventLoop {
  private Lock lock;
  private Set<handler> handlers;
  private Set<signal> signals;

  ...
}
```

- Tying Landin's Knot using a reference protected by a lock.

Recursive Abstractions

Verifying a lock-based event loop implementation

- The footprint of an event loop is thus recursively defined and the recursion **"goes through the lock"**.

Recursive Abstractions

Verifying a lock-based event loop implementation

- The footprint of an event loop is thus recursively defined and the recursion **“goes through the lock”**.

- We define eloop using **guarded recursion** and the **third-order** isLock representation predicate:

$$
\begin{aligned}
\mathrm{eloop} = & \mathit{fix}(\lambda \mathrm{eloop} : \mathrm{Val} \to \mathrm{Prop}.\ \lambda x : \mathrm{Val}. \\
& \quad \exists l.\ \mathrm{x}.\mathit{lock} \overset{-}{\mapsto} l * \\
& \quad\quad \mathrm{isLock}(l, \exists \mathrm{y}, \mathrm{z}, A, B.\ \mathit{set}(\mathrm{y}, A) * \mathit{set}(\mathrm{z}, B) \\
& \quad\quad\quad * \ \mathrm{x}.\mathit{handlers} \mapsto y * \mathrm{x}.\mathit{signals} \mapsto z \\
& \quad\quad\quad * \ \forall a \in A.\ \triangleright a \mapsto \{\mathrm{eloop}(\mathrm{x})\}\{\mathrm{emp}\}))
\end{aligned}
$$

Recursive Abstractions

Verifying a lock-based event loop implementation

- The footprint of an event loop is thus recursively defined and the recursion **"goes through the lock"**.
- We define eloop using **guarded recursion** and the **third-order** isLock representation predicate:

$$\begin{aligned}\mathrm{eloop} = &\mathit{fix}(\lambda \mathrm{eloop} : \mathrm{Val} \to \mathrm{Prop}.\ \lambda x : \mathrm{Val}.\\ &\quad \exists l.\ \mathrm{x}.\mathit{lock} \overset{-}{\mapsto} l \ast\\ &\qquad \mathrm{isLock}(l, \exists \mathrm{y}, \mathrm{z}, A, B.\ \mathit{set}(\mathrm{y}, A) \ast \mathit{set}(\mathrm{z}, B)\\ &\qquad\quad \ast\ \mathrm{x}.\mathit{handlers} \mapsto y \ast \mathrm{x}.\mathit{signals} \mapsto z\\ &\qquad\quad \ast\ \forall a \in A.\ \triangleright a \mapsto \{\mathrm{eloop}(\mathrm{x})\}\{\mathrm{emp}\}))\end{aligned}$$

Recursive Abstractions

Verifying a lock-based event loop implementation

- The footprint of an event loop is thus recursively defined and the recursion **“goes through the lock”**.

- We define eloop using **guarded recursion** and the **third-order** isLock representation predicate:

$$
\begin{aligned}
\mathrm{eloop} = &\mathit{fix}(\lambda \mathrm{eloop} : \mathrm{Val} \to \mathrm{Prop}.\ \lambda x : \mathrm{Val}.\\
&\quad \exists l.\ \mathrm{x}.\mathit{lock} \overset{-}{\mapsto} l \ast\\
&\qquad \mathrm{isLock}(l, \exists \mathrm{y}, \mathrm{z}, A, B.\ \mathit{set}(\mathrm{y}, A) \ast \mathit{set}(\mathrm{z}, B)\\
&\qquad\qquad \ast\ \mathrm{x}.\mathit{handlers} \mapsto y \ast \mathrm{x}.\mathit{signals} \mapsto z\\
&\qquad\qquad \ast\ \forall a \in A.\ \triangleright a \mapsto \{\mathrm{eloop}(\mathrm{x})\}\{\mathrm{emp}\}))
\end{aligned}
$$

Recursive Abstractions

Verifying a lock-based event loop implementation

- The footprint of an event loop is thus recursively defined and the recursion **“goes through the lock”**.
- We define eloop using **guarded recursion** and the **third-order** isLock representation predicate:

$$
\begin{aligned}
\mathrm{eloop} = & \mathit{fix}(\lambda \mathrm{eloop} : \mathrm{Val} \to \mathrm{Prop}.\ \lambda x : \mathrm{Val}.\\
& \quad \exists l.\ \mathrm{x}.\mathit{lock} \overset{-}{\mapsto} l \ast \\
& \quad\quad \mathrm{isLock}(l,\ \exists \mathrm{y}, \mathrm{z}, A, B.\ \mathit{set}(\mathrm{y}, A) \ast \mathit{set}(\mathrm{z}, B) \\
& \quad\quad\quad \ast\ \mathrm{x}.\mathit{handlers} \mapsto y \ast \mathrm{x}.\mathit{signals} \mapsto z \\
& \quad\quad\quad \ast\ \forall a \in A.\ \triangleright a \mapsto \{\mathrm{eloop}(\mathrm{x})\}\{\mathrm{emp}\}))
\end{aligned}
$$

Recursive Abstractions

Verifying a lock-based event loop implementation

- The footprint of an event loop is thus recursively defined and the recursion **“goes through the lock”**.

- We define eloop using **guarded recursion** and the **third-order** isLock representation predicate:

$$
\begin{aligned}
\mathrm{eloop} = & \, \mathit{fix}(\lambda \mathrm{eloop} : \mathrm{Val} \to \mathrm{Prop}.\ \lambda x : \mathrm{Val}. \\
& \quad \exists l.\ \mathrm{x}.\mathit{lock} \overset{-}{\mapsto} l \ast \\
& \qquad \mathrm{isLock}(l, \exists \mathrm{y}, \mathrm{z}, A, B.\ \mathit{set}(\mathrm{y}, A) \ast \mathit{set}(\mathrm{z}, B) \\
& \qquad\qquad \ast\ \mathrm{x}.\mathit{handlers} \mapsto y \ast \mathrm{x}.\mathit{signals} \mapsto z \\
& \qquad\qquad \ast\ \forall a \in A.\ \triangleright a \mapsto \{\mathrm{eloop}(\mathrm{x})\}\{\mathrm{emp}\}))
\end{aligned}
$$

Recursive Abstractions

Verifying a lock-based event loop implementation

- The footprint of an event loop is thus recursively defined and the recursion **"goes through the lock"**.
- We define eloop using **guarded recursion** and the **third-order** isLock representation predicate:

$$
\begin{aligned}
\mathrm{eloop} = & \mathit{fix}(\lambda \mathrm{eloop} : \mathrm{Val} \to \mathrm{Prop}.\ \lambda x : \mathrm{Val}. \\
& \exists l.\ \mathrm{x}.\mathit{lock} \stackrel{-}{\mapsto} l * \\
& \quad \textcolor{red}{\mathrm{isLock}}(l, \exists \mathrm{y}, \mathrm{z}, A, B.\ \mathit{set}(\mathrm{y}, A) * \mathit{set}(\mathrm{z}, B) \\
& \qquad * \ \mathrm{x}.\mathit{handlers} \mapsto y * \mathrm{x}.\mathit{signals} \mapsto z \\
& \qquad * \ \forall a \in A.\ \triangleright a \mapsto \{\mathrm{eloop}(\mathrm{x})\}\{\mathrm{emp}\}))
\end{aligned}
$$

Must be non-expansive!

iCAP

Reasoning about shared state

- Following CAP, iCAP extends separation logic with shared regions and protocols to govern shared state.

- The state is split into a local part and shared regions.

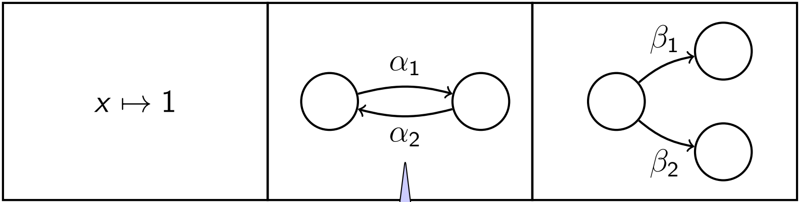
local	region 1	region 2

iCAP

Reasoning about shared state

- Following CAP, iCAP extends separation logic with shared regions and protocols to govern shared state.

- The state is split into a local part and shared regions.



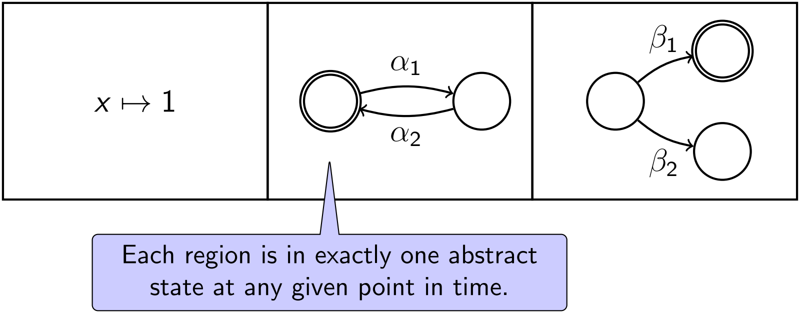
A labelled transition system specifies the possible **abstract states** of the shared region.

iCAP

Reasoning about shared state

- Following CAP, iCAP extends separation logic with shared regions and protocols to govern shared state.

- The state is split into a local part and shared regions.

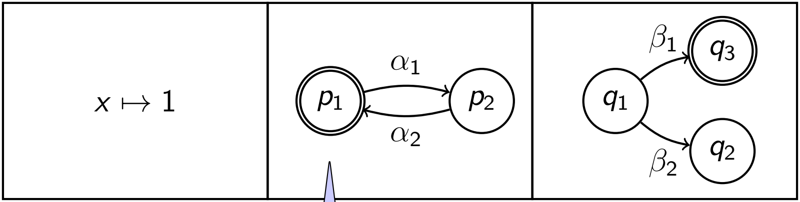


iCAP

Reasoning about shared state

- Following CAP, iCAP extends separation logic with shared regions and protocols to govern shared state.

- The state is split into a local part and shared regions.



A predicate that defines the resources owned by the shared region in the given abstract state.

A Modular Lock Specification

Verifying a spinlock implementation

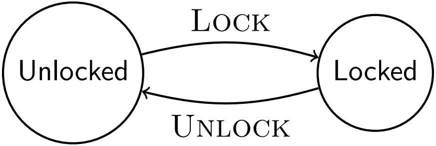
- Upon allocating a lock, we allocate a shared region to govern the sharing **of** the lock and **through** the lock.

A Modular Lock Specification

Verifying a spinlock implementation

- Upon allocating a lock, we allocate a shared region to govern the sharing **of** the lock and **through** the lock.

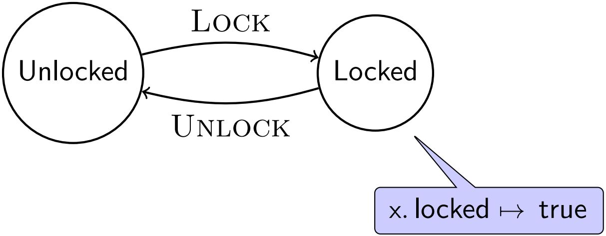
- A lock can be in one of two abstract states:



A Modular Lock Specification

Verifying a spinlock implementation

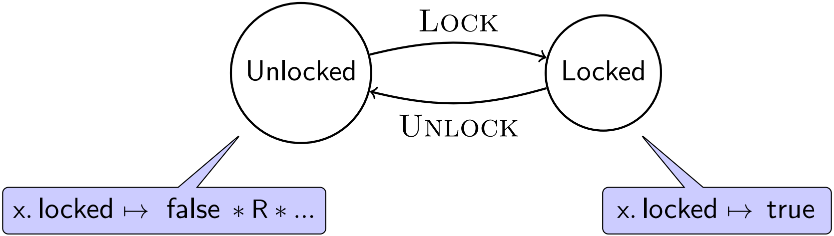
- Upon allocating a lock, we allocate a shared region to govern the sharing **of** the lock and **through** the lock.
- A lock can be in one of two abstract states:



A Modular Lock Specification

Verifying a spinlock implementation

- Upon allocating a lock, we allocate a shared region to govern the sharing **of** the lock and **through** the lock.
- A lock can be in one of two abstract states:

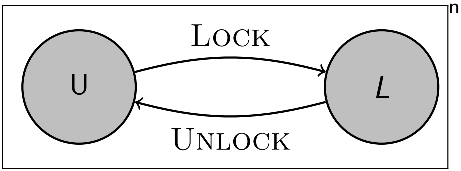


A Modular Lock Specification

Verifying a spinlock implementation

Formally

$$\mathsf{isLock}(x, R) = \exists n : \mathsf{RId}.\ [\textsc{Lock}]^{n}_{-} * \mathsf{rintr}(I(x, R, n), n)$$

$*$ 

where

$$I(x, R, n)(s) = \begin{cases} x.\mathsf{locked} \mapsto \mathsf{true} & \text{if } s = \mathsf{L} \\ x.\mathsf{locked} \mapsto \mathsf{false} * R * [\textsc{Unlock}]^{n}_{1} & \text{if } s = \mathsf{U} \end{cases}$$

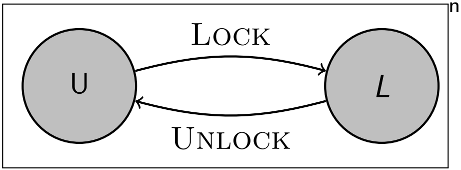
A Modular Lock Specification

Verifying a spinlock implementation

Formally

$\mathsf{rintr} : (\mathsf{SId} \to \mathsf{Prop}) \times \mathsf{RId} \to \mathsf{Prop}$

$$\mathsf{isLock}(\mathsf{x}, \mathsf{R}) = \exists \mathsf{n} : \mathsf{RId}.\ [\textsc{Lock}]^{\mathsf{n}}_{\_} * \mathsf{rintr}(\mathsf{I}(\mathsf{x}, \mathsf{R}, \mathsf{n}), \mathsf{n})$$

$*$ 

where

$$I(\mathsf{x}, \mathsf{R}, \mathsf{n})(s) = \begin{cases} \mathsf{x.locked} \mapsto \mathsf{true} & \text{if } s = \mathsf{L} \\ \mathsf{x.locked} \mapsto \mathsf{false} * \mathsf{R} * [\textsc{Unlock}]^{n}_{1} & \text{if } s = \mathsf{U} \end{cases}$$

Model

Impredicative protocols introduce a circularity

- rintr(I, n) : Prop, asserts that the interpretation of the abstract states of region n are given by I : SId $\to$ Prop

$$\mathit{Prop} \cong \mathcal{P}^{\uparrow}(\ldots \times (\mathit{RId} \times \mathit{SId} \rightharpoonup_{\mathit{fin}} \mathit{Prop}))$$

Model

Impredicative protocols introduce a circularity

- $\mathrm{rintr}(I, n) : \mathrm{Prop}$, asserts that the interpretation of the abstract states of region n are given by $I : \mathrm{SId} \to \mathrm{Prop}$

$$\mathit{Prop} \cong \mathcal{P}^{\uparrow}(\ldots \times (\mathit{RId} \times \mathit{SId} \rightharpoonup_{\mathit{fin}} \mathit{Prop}))$$

- We (implicitly) use step-indexing to solve the circularity. We define the model using the internal lang. of the topos of trees.

Model

Impredicative protocols introduce a circularity

- $\mathrm{rintr}(I, n)$: Prop, asserts that the interpretation of the abstract states of region n are given by I : SId $\rightarrow$ Prop

$$\mathit{Prop} \cong \mathcal{P}^{\uparrow}(\ldots \times (\mathit{RId} \times \mathit{SId} \rightharpoonup_{\mathit{fin}} \mathit{Prop}))$$

- We (implicitly) use step-indexing to solve the circularity. We define the model using the internal lang. of the topos of trees.
- iCAP function space is the topos of trees function space:

$$[\![\tau \rightarrow \sigma]\!] = [\![\tau]\!] \rightarrow [\![\sigma]\!]$$

iCAP function space

Topos of trees function space

Related work

CAP

- [Dodds et al., POPL 2011] was unsound, because the authors broke the circularity introduced by impredicative protocols, but reasoned as if they had solved it.
- In HOCAP [ESOP 2013] we broke the circularity and introduced a predicative stratification to ensure soundness.

CaReSL [Turon et al., ICFP 2013]

- Model related to iCAP model, but logic is only second-order, so types of CaReSL can be interpreted as constant sets (in iCAP they are variable sets, objects in topos of trees).

Conclusion

Recursive abstractions

- Recursive abstractions are **useful** and ubiquitous in higher-order code with effects!
- We can reason about recursive abstractions using higher-order specifications and guarded recursion.

iCAP

- A logic for modular reasoning about partial correctness of concurrent, higher-order, reentrant, imperative code.