Higher-Order Concurrent Separation Logic: Why and How

Lars Birkedal
Aarhus University

Nijmegen, Netherlands
Dec, 2015

ModuRes

Modular Reasoning about
Higher-Order Concurrent Imperative Programs

Introduction

Goal

- Program logics for modular reasoning about partial correctness of higher-order, concurrent, imperative code programs.

Outline

1. **Why higher-order logic + guarded recursion** is useful for expressing such modular specs
   - via example of layered and recursive abstraction.
2. **Why impredicative protocols** to govern shared state
   - via example verification of lock implementation
   - **invariants** to enforce protocols
   - **monoids** to express protocols (ownership)
3. **How** the logic is modelled and showed sound (key ideas)
   - BI-hyperdoctrine over ultra-metric spaces, Kripke model with recursively defined worlds.
4. Overview of resources to learn more
   - tutorial material
   - papers: iCAP [ESOP-2014] and Iris [POPL-2015]
   - paper on formalization: ModuRes Library [ITP-2015]
5. Overview of features in iCAP and Iris not covered today

Higher-Order Programming

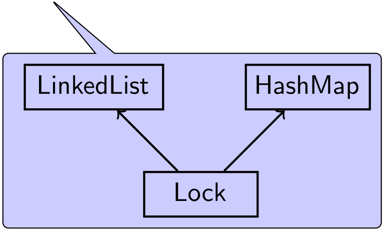
- Programming features
  - HO functions
  - Interfaces in OO languages
  - Function Pointers in low-level languages
- for
  - Building libraries
  - Returning libraries
  - Parameterization
- Point:
  - Features important for modularizing large programs
  - Allow programming relative to unknown code
- Goal: Logics and Models that support correspondingly modular specifications of code.

Example: Layered and Recursive Abstractions

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

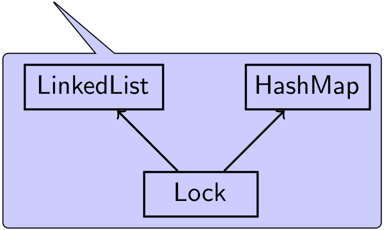
Example: Layered and Recursive Abstractions

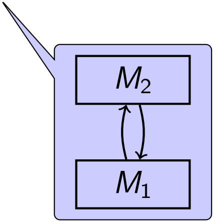
- Modular library specifications that supports **layering** of abstractions and **recursive** abstractions.



Example: Layered and Recursive Abstractions

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





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, ...

A Modular Lock Specification

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

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

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

A Modular Lock Specification

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

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

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

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

A Modular Lock Specification

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

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

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

A Modular Lock Specification

Third-order quantification.

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

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

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

A Modular Lock Specification

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

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

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

A Modular Lock Specification

Second-order quantification.

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

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

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

A Modular Lock Specification

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

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

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

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

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

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

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:

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

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:

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

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:

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

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:

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

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:

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

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:

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

Must be non-expansive!

Summary

- Higher-order logic $+$ guarded recursion useful for specifying layered and recursive abstractions.
- Next: impredicative protocols for verifying module implementations

A Modular Lock Specification

Verifying a spinlock implementation

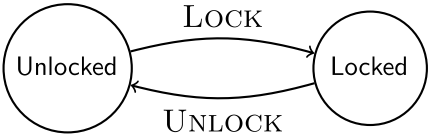
- Upon allocating a lock, we introduce a protocol to govern the sharing **of** the lock and **through** the lock.

A Modular Lock Specification

Verifying a spinlock implementation

- Upon allocating a lock, we introduce a protocol to govern the sharing **of** the lock and **through** the lock.

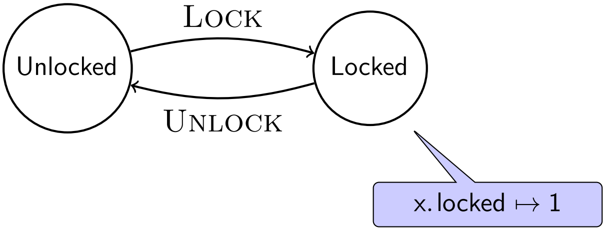
- A lock can be in one of two abstract states:



A Modular Lock Specification

Verifying a spinlock implementation

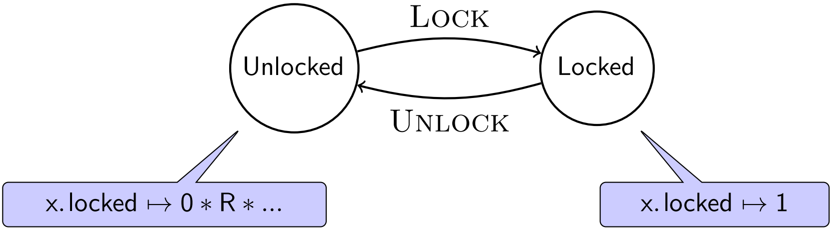
- Upon allocating a lock, we introduce a protocol to govern the sharing **of** the lock and **through** the lock.
- A lock can be in one of two abstract states:



A Modular Lock Specification

Verifying a spinlock implementation

- Upon allocating a lock, we introduce a protocol 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

- Protocol expressed via invariant:

$$
\begin{aligned}
I(\mathsf{R}) = {} & (\mathsf{x.locked} \mapsto 1) \\
& \vee (\mathsf{x.locked} \mapsto 0 * \mathsf{R} * \bullet)
\end{aligned}
$$

A Modular Lock Specification

- Protocol expressed via invariant:

$$I(\mathsf{R}) = (\mathsf{x.locked} \mapsto 1) \quad \text{locked}$$
$$\vee\, (\mathsf{x.locked} \mapsto 0 * \mathsf{R} * \bullet) \quad \text{unlocked}$$

A Modular Lock Specification

- Protocol expressed via invariant:

$$I(\mathsf{R}) = \overbrace{(\mathsf{x.locked} \mapsto 1)}^{\text{locked}}$$
$$\lor \underbrace{(\mathsf{x.locked} \mapsto 0 * \mathsf{R} * \bullet)}_{\text{unlocked}}$$

- Parameterized by R, any predicate, including one referring to protocols, hence **impredicative protocols**.

A Modular Lock Specification

- Protocol expressed via invariant:

locked

$$I(\mathsf{R}) = (\mathsf{x.locked} \mapsto 1)$$
$$\vee\ (\mathsf{x.locked} \mapsto 0 * \mathsf{R} * \bullet)$$

unlocked

- Parameterized by R, any predicate, including one referring to protocols, hence **impredicative protocols**.
- Only the owner of the lock should be able to unlock:
  - Expressed via monoid: the $\bullet$ is an element of a partial commutative monoid (PCM), with one non-neutral element, which is not duplicable.
  - Also uses standard PCM of heaps.

Summary

- Monoids to express protocols on shared state
- Invariants to **enforce** protocol governing shared state
- Protocol parametric in R: impredicative protocols.

Fine-Grained Concurrent Data Structures

- Modular Bag Specification (excerpt)

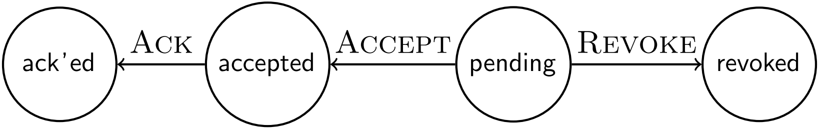
$$\exists \mathsf{bag} : \mathsf{RId} \times \mathsf{Val} \to \mathsf{Prop}.$$

$$\{emp\}\ \textbf{new}\ \mathtt{Bag(-)}\ \{\mathsf{ret}.\ \exists n : \mathsf{RId}.\ \mathsf{bag}(n, \mathsf{ret}) * \mathsf{ret}_{\mathsf{cont}} \overset{0.5}{\mapsto} \emptyset\}$$

$$\forall P, Q : \mathsf{Val} \times \mathsf{Val} \to \mathit{Prop}.\ \forall n : \mathsf{RId}.$$
$$\forall X : \mathcal{P}_m(\mathsf{Val}).\ \forall x, y : \mathsf{Val}.$$
$$x_{\mathsf{cont}} \overset{0.5}{\mapsto} X * P(x, y) \sqsubseteq^{\mathit{RId} \setminus \{n\}} x_{\mathsf{cont}} \overset{0.5}{\mapsto} (X \cup \{y\}) * Q(x, y)) \ \Rightarrow$$
$$\{\mathsf{bag}(n, x) * P(x, y)\}\ \mathtt{x.Push(y)}\ \{\mathsf{bag}(n, x) * Q(x, y)\}$$

- Parameterization to allow clients to reason about what should happen at linearization points (inspired by Jacobs and Piessens):

Verification of Implementation using Helping



- pending: an offer has been made and it is waiting for somebody to accept it,
- accepted: the offer has been accepted,
- ack'ed: we have acknowledged that somebody has accepted the offer,
- revoked: in case we revoke the offer (since no one accepted it and now we will re-attempt to push).

Interpretation of states

$$
\begin{aligned}
\mathsf{I}_{\mathit{offer}(\mathsf{pending})} &= x.\mathsf{state} \mapsto 0 * P(b, y) * \\
&\quad \forall X : \mathcal{P}_m(\mathsf{Val}).\ \forall x, y : \mathsf{Val}. \\
&\quad x_{\mathsf{cont}} \overset{0.5}{\mapsto} X * P(x, y) \sqsubseteq^{\mathit{RId} \setminus \{n\}} x_{\mathsf{cont}} \overset{0.5}{\mapsto} (X \cup \{y\}) * Q(x, y \\
\mathsf{I}_{\mathit{offer}(\mathsf{accepted})} &= x.\mathsf{state} \mapsto 1 * Q(b, y) \\
\mathsf{I}_{\mathit{offer}(\mathsf{revoked})} &= x.\mathsf{state} \mapsto 2 \\
\mathsf{I}_{\mathit{offer}(\mathsf{ack'ed})} &= x.\mathsf{state} \mapsto 1
\end{aligned}
$$

Summary

- Key logical tools: impredicative protocols and view shifts. (the depicted state transition system can be encoded using PCMs, as we indicated for locks earlier)
- iCap was the first logic to allow verification of such FCDs against such general specs.

Model Ideas

- $\mathrm{Prop} = W \rightarrow_{mon} P(M)$
  - M a PCM (a product of all the necessary monoids)
  - W: worlds, keeping track of invariants

Model Ideas

- Prop $= W \rightarrow_{mon} P(M)$
  - M a PCM (a product of all the necessary monoids)
  - W: worlds, keeping track of invariants
- $W = N \rightharpoonup_{fin}$ Prop
  - names of invariants are natural numbers
  - each invariant described by a predicate (recall $I(\mathrm{R})$ for the lock)

Model Ideas

- Prop $= W \to_{mon} P(M)$
  - M a PCM (a product of all the necessary monoids)
  - W: worlds, keeping track of invariants
- $W = N \rightharpoonup_{fin}$ Prop
  - names of invariants are natural numbers
  - each invariant described by a predicate (recall $I(\mathrm{R})$ for the lock)
- So need something like

$$\mathrm{Prop} \cong (N \rightharpoonup_{fin} \mathrm{Prop}) \to_{mon} P(M)$$

Model Ideas

- Our approach: solve in model of guarded type theory:

$$\mathrm{Prop} \cong \blacktriangleright(N \rightharpoonup_{fin} \mathrm{Prop}) \to_{mon} P(M)$$

- iCAP: use topos of trees and construct model synthetically
- Iris: more explicit using subcat U corresponding to ultrametric spaces.

Model in U

- Let U be category of complete bisected ultrametric spaces and non-expansive maps.
- Equivalently described as complete ordered families of equivalence relations and non-expansive maps (cofe).
- Use uniform predicates instead of just predicates:

  $$\mathrm{UPred}(M) = N^{\mathsf{op}} \rightarrow_{mon} P^{\uparrow}(M)$$

  Two such are n-equal if they agree up to level n.
- Can model guarded predicates (the $\triangleright$) by "shifting a uniform predicate to the right".
- Solve

  $$\mathsf{Prop} \cong \blacktriangleright((N \rightharpoonup_{fin} \mathsf{Prop}) \rightarrow_{ne}^{mon} \mathrm{UPred}(M))$$

Model in U

- **Theorem** $U(\_, \mathrm{Prop})$ is a BI-hyperdoctrine, which also
  - models guarded recursive predicates
  - models invariants
  - also models hoare triples and associated proof rules, but omitted from presentation today

Model in U

- **Theorem** $U(\_, \mathrm{Prop})$ is a BI-hyperdoctrine, which also
  - models guarded recursive predicates
  - models invariants
  - also models hoare triples and associated proof rules, but omitted from presentation today
- Explicitly:
  - $\Gamma \vdash \phi : \mathrm{prop}$ is interpreted as a non-expansive function from $[\![\Gamma]\!]$ to Prop.

$$
\begin{aligned}
[\![\Gamma \vdash M =_\tau N]\!]_\gamma w &= \left\{(n, r) \mid [\![\Gamma \vdash M : \tau]\!]_\gamma \overset{n+1}{=} [\![\Gamma \vdash N : \tau]\!]_\gamma\right\} \\
[\![\Gamma \vdash \top]\!]_\gamma w &= N \times M \\
[\![\Gamma \vdash \phi \wedge \psi]\!]_\gamma w &= [\![\Gamma \vdash \phi]\!]_\gamma w \cap [\![\Gamma \vdash \psi]\!]_\gamma w \\
[\![\Gamma \vdash \bot]\!]_\gamma w &= \emptyset \\
[\![\Gamma \vdash \phi \vee \psi]\!]_\gamma w &= [\![\Gamma \vdash \phi]\!]_\gamma w \cup [\![\Gamma \vdash \psi]\!]_\gamma w \\
[\![\Gamma \vdash \phi \implies \psi]\!]_\gamma w &= \forall w' \geq w, \forall n' \leq n, \forall r' \geq r, \\
&\qquad (n', r') \in [\![\Gamma \vdash \phi]\!]_\gamma w' \Rightarrow (n', r') \in [\![\Gamma \vdash \psi]\!]_\gamma w' \\
[\![\Gamma \vdash \forall x : \sigma, \phi]\!]_\gamma w &= \bigcap_{d \in [\![\sigma]\!]} [\![\Gamma, x : \sigma \vdash \phi]\!]_{(\gamma, d)} w \\
[\![\Gamma \vdash \exists x : \sigma, \phi]\!]_\gamma w &= \bigcup_{d \in [\![\sigma]\!]} [\![\Gamma, x : \sigma \vdash \phi]\!]_{(\gamma, d)} w
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash \triangleright\phi]\!]_\gamma w &= \{(0, r) \mid r \in M\} \\
&\quad \cup \{(n+1, r) \mid (n, r) \in [\![\Gamma \vdash \phi]\!]_\gamma w\} \\
[\![\Gamma \vdash \mathrm{emp}]\!]_\gamma w &= N \times M \\
[\![\Gamma \vdash \phi \star \psi]\!]_\gamma w &= \{(n, r) \mid \exists r_1, r_2, r = r_1 \cdot r_2 \wedge \\
&\qquad (n, r_1) \in [\![\Gamma \vdash \phi]\!]_\gamma w \wedge (n, r_2) \in [\![\Gamma \vdash \psi]\!]_\gamma w\} \\
[\![\Gamma \vdash \phi \mathrel{-\!\!\star} \psi]\!]_\gamma w &= \{(n, r) \mid \forall w' \geq w, \forall n' \leq n, \forall r' \# r. \\
&\qquad (n', r') \in [\![\Gamma \vdash \phi]\!]_\gamma w' \wedge (n', r \cdot r') \in [\![\Gamma \vdash \psi]\!]_\gamma w'\}
\end{aligned}
$$

Recursively defined predicates

When p occurs under a $\triangleright$ in ϕ, then

$$\llbracket \Gamma \vdash \mu p.\phi : \mathsf{prop}^\tau \rrbracket_\gamma = \mathit{fix}(\lambda x : \llbracket \mathsf{prop}^\tau \rrbracket. \\ \llbracket \Gamma, p : \mathsf{prop}^\tau \vdash \phi : \mathsf{prop}^\tau \rrbracket_{(\gamma, x)})$$

Here *fix* yields the fixed point of the contractive function,

Invariant predicates

$$\llbracket \Gamma \vdash \boxed{\phi}^{K} \rrbracket_\gamma w = \{(n, r) \mid \\ w(\llbracket \Gamma \vdash K \rrbracket_\gamma) \stackrel{n+1}{=} \iota(\llbracket \Gamma \vdash \phi \rrbracket_\gamma)\}.$$

Model in U

- **Theorem** $U(\_, \mathrm{Prop})$ is a BI-hyperdoctrine, which also
  - models guarded recursive predicates
  - models invariants
- Key Observation (Iris):
  - By choosing monoids appropriately, we can encode many (all ?) kinds of protocols and ownership disciplines and derive rules for other concurrent separation logics

Use of Interactive Theorem Proving

- Verify meta-theory (model construction and soundness of all proof rules, relative to operational semantics of concrete programming language).
- Tool(s) for experimenting with larger program verification.
  - Support for higher-order quantification is important.
- Earlier work: Charge! (with Bengtson, Jensen, Sieczkowski)
  - For sequential OO language, logic without invariants.
  - Shallow embedding of logic, Coq tactics to automate part of the reasoning.
  - Non-trivial to reason efficiently **in** embedded logic.

Use of Interactive Theorem Proving

- ModuRes Coq Library [ITP-2015]
  - Implementation of U and solutions to g.r. domain eqn's.
  - Draft tutorial material online (model of ML references)
  - Used to verify meta-theory of Iris
- Ongoing and Future Work (with Krebbers, Jung, Dreyer, Bengtson, . . .)
  - New tool for concurrency reasoning based on improvement of ModuRes Coq Library
  - Hope to make use of progress by Malecha and Bengtson on reflective tactics.

To Learn More

- Elementary tutorial notes on categorical logic (including above theorem)
- Papers:
  - ModuRes Coq Library [ITP-2015]
  - iCAP [ESOP-2014]
  - Iris [POPL-215]
  - Cover many other aspects, e.g. advanced FCDs, logical atomicity (definable) as in TaDa.
- `http://cs.au.dk/~birke/modures/`

More Ongoing and Future Work

- Defining logical relations in Iris (a la Plokin-Abadi for System F), to prove relational properties, e.g.,
  - security properties such as non-interference
  - optimizations based on type and effect systems
- Liveness properties.

Thank You

Acknowledgements

- Kasper Svendsen
- Aleš Bizjak
- Filip Sieczkowski
- Yannick Zakowski
- Derek Dreyer
- Aaron Turon
- Ralf Jung
- David Swasey