Verifying a higher-order, concurrent, stateful library

Kasper Svendsen, Lars Birkedal and Matthew Parkinson

September 9, 2012
HOPE 2012
A case study ...

- C# Joins library [Russo, Turon & Russo]
  - declarative way of defining synchronization primitives, based on the join calculus [Fournet & Gonthier]
  - combines higher-order features with state, concurrency, recursion through the store and fine-grained synchronization
  - small (150 lines of C#) realistic library
A case study in modularity

- Locks
- …
- Barriers

Joins specification

- Lock-based
- Non-locking

- Lock
- Concurrent bag

Join implementations

Join clients
class RWLock {
    public SyncChannel acqR, acqW, relR, relW;
    private AsyncChannel unused, shared, writer;
    private int readers = 0;

    public RWLock() {
        Join join = new Join();
        // ... initialize channels ...
        join.When(acqR).And(unused).Do(() => { readers++; shared(); });
        join.When(acqR).And(shared).Do(() => { readers++; shared(); });
        join.When(acqW).And(unused).Do(() => { writer(); });
        join.When(relW).And(writer).Do(() => { unused(); });
        join.When(relR).And(shared).Do(() => {
            if (--readers == 0) unused() else shared();
        });
        unused();
    }
}
A reader/writer lock

class RWLock {
public SyncChannel acqR, acqW, relR, relW;
private AsyncChannel unused, shared, writer;
private int readers = 0;

public RWLock() {
    Join join = new Join();
    // ... initialize channels ...
    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    join.When(acqR).And(shared).Do(() => { readers++; shared(); });
    join.When(acqW).And(unused).Do(() => { writer(); });
    join.When(relW).And(writer).Do(() => { unused(); });
    join.When(relR).And(shared).Do(() => {
        if (--readers == 0) unused() else shared();
    });
    unused();
}
}
class RWLock {
    public SyncChannel acqR, acqW, relR, relW;
    private AsyncChannel unused, shared, writer;
    private int readers = 0;

    public RWLock() {
        Join join = new Join();
        // ... initialize channels ...

        join.When(acqR).And(unused).Do(() => { readers++; shared(); });
        join.When(acqR).And(shared).Do(() => { readers++; shared(); });
        join.When(acqW).And(unused).Do(() => { writer(); });
        join.When(relW).And(writer).Do(() => { unused(); });
        join.When(relR).And(shared).Do(() => {
            if (--readers == 0) unused() else shared(); });
        unused();
    }
}

Verification challenges

- reentrant continuation
- state effect
Joins specification

- Locks
- Lock-based
  - Lock
- ... (omitted)
- Barriers
- Non-locking
  - Concurrent bag
Specification

- Requirements:
  - Ownership transfer
  - Stateful reentrant continuations
- Restrict attention to non-self-modifying clients
Ideas

• Let clients pick an ownership protocol for each channel

  • The channel pre-condition describes the resources the sender is required to transfer to the recipient upon sending a message

  • The channel post-condition describes the resources the recipient is required to transfer to the sender upon receiving the message

• The channel post-condition of asynchronous channels must be empty

• Prove chords obey the ownership protocol, assuming channels obey the ownership protocol (to support reentrancy)
Specification

• Send a message on channel c (async or sync)

\{\text{join}(P, Q, j) \ast \text{chan}(c, j) \ast P(c)\}

\text{c()}

\{\text{join}(P, Q, j) \ast \text{chan}(c, j) \ast Q(c)\}

family of channel pre- and post-conditions, indexed by channels

transfer channel pre-condition from client to join instance

if c is an asynchronous channel, then channel post-condition must be emp

transfer channel post-condition from join instance to client
Specification

• Register a new chord with pattern \( p \) and continuation \( b \)

\[
\begin{align*}
\text{join}_{\text{init-pat}}(P, Q, j) \ast \text{pattern}(p, j, X) \\
\ast b \mapsto \{ \bigotimes_{x \in X} P(x) \ast \text{join}(P, Q, j) \} \\
\big\{ \bigotimes_{x \in X} Q(x) \ast \text{join}(P, Q, j) \big\} \\
p.\text{Do}(b) \\
\{ \text{join}_{\text{init-pat}}(P, Q, j) \} \\
\end{align*}
\]

pattern \( p \) matches the multiset of channels \( X \)

resources senders must transfer to recipient

resources recipient must transfer to senders
Specification

• Register a new chord with pattern p and continuation b

\[
\begin{align*}
\text{join}_{\text{init-pat}}(P, Q, j) \ast \text{pattern}(p, j, X) \\
\ast b &\mapsto \big\{ \bigotimes_{x \in X} P(x) \ast \text{join}(P, Q, j) \big\} \\
\big\{ \bigotimes_{x \in X} Q(x) \ast \text{join}(P, Q, j) \big\} \\
p.\text{Do}(b) \\
\text{join}_{\text{init-pat}}(P, Q, j)
\end{align*}
\]

the continuation is allowed to assume channels obey their ownership protocol
Verifying Clients

- Locks
- ... Barriers

Join specification

- Lock-based
  - Lock
- Non-locking
  - Concurrent bag
Reader/Writer lock

• Given resource invariants R and $R_{ro}$ (picked by client) s.t.

$$\forall n \in \mathbb{N}. \; R(n) \Leftrightarrow R_{ro} \ast R(n + 1)$$

• $R_{ro}$ : read permission to underlying resource
• $R(0)$: write permission to underlying resource
• $R(n)$: resource after splitting off $n$ read permissions

• The reader/writer lock satisfies the following specification

\[
\begin{align*}
\{emp\} & \xrightarrow{\text{acqR}()} \{R_{ro}\} \\
\{emp\} & \xrightarrow{\text{acqW}()} \{R(0)\} \\
\{R_{ro}\} & \xrightarrow{\text{relR}()} \{emp\} \\
\{R(0)\} & \xrightarrow{\text{relW}()} \{emp\}
\end{align*}
\]
• Assign pre-conditions to asynchronous channels

\[ P(\text{unused}) = \text{readers} \rightarrow 0 \ast R(0) \]
\[ P(\text{shared}) = \exists n \in \mathbb{N}. \text{readers} \rightarrow n \ast R(n) \ast n > 0 \]
\[ P(\text{writer}) = \text{readers} \rightarrow 0 \]

• Assign pre- and post-conditions to synchronous channels

\[ P(\text{acqR}) = emp \]
\[ Q(\text{acqR}) = R_{ro} \]
\[ P(\text{acqW}) = emp \]
\[ Q(\text{acqW}) = R(0) \]
\[ P(\text{relR}) = R_{ro} \]
\[ Q(\text{relR}) = emp \]
\[ P(\text{relW}) = R(0) \]
\[ Q(\text{relW}) = emp \]
• Prove chords obey channel ownership protocol

class RWLock {
    ...  
    public int readers = 0;
    
    public RWLock() {
        ...
        join.When(acqR).And(unused).Do(() => { readers++; shared(); });
        ...
    }
}

\{P(acqR) \ast P(unused) \ast \text{join}(P, Q, j)\}

readers++

\{Q(acqR) \ast Q(unused) \ast \text{join}(P, Q, j)\}

shared();
• Prove chords obey channel ownership protocol

class RWLock {
    ...
    public int readers = 0;

    public RWLock() {
        ...
        join.When(acqR).And(unused).Do(() => { readers++; shared(); });
        ...
    }

    {readers \mapsto 0 \ast R(0) \ast \text{join}(P,Q,j)}
    readers++
    {readers \mapsto 1 \ast R(1) \ast R_{ro} \ast \text{join}(P,Q,j)}
    shared();
    {R_{ro} \ast \text{join}(P,Q,j)}
    \quad P(\text{shared}) = \exists n \in \mathbb{N}_{+}.
    \quad \text{readers} \mapsto n \ast R(n)
Verifying an Implementation

Locks

...  

Barriers

Joins specification

Lock-based

Non-locking

Lock

Concurrent bag
Verifying an Implementation

- Challenges:
  - High-level join primitives implemented using shared mutable state
  - Definition of recursive representation predicates

Guarded recursion & step-indexed model
class Message {
    public int state;

    public Message() {
        state = 0;
    }

    public void Receive() {
        state = 1;
    }
}

Messages

- Assume channel pre- and post-conditions P and Q
- Imagine a message on channel c

\[
\text{state } \rightarrow 0
\]
\[\ast P(c)\]

\[
\text{state } \rightarrow 0
\]

\[
\text{state } \rightarrow 1
\]
\[\ast Q(c)\]

anybody can perform this transition

only message sender can perform this transition
• Use Concurrent Abstract Predicates [Dinsdale-Young et. al.] to impose this low-level protocol on messages

```
state \rightarrow 0
\ast P(c)
```

```
state \rightarrow 0
```

```
state \rightarrow 1
\ast Q(c)
```

```
state \rightarrow 1
```

higher-order protocol

anybody can perform this transition

only message sender can perform this transition
HOCAP

- Higher-order protocols are difficult; the previous proposal [Dodds et. al.] from POPL11 is unsound!

- We restrict attention to state-independent higher-order protocols. An assertion $P$ is expressible using state-independent protocols (SIPs) iff

  \[ \exists R, S : \text{Prop. valid} \ (P \Leftrightarrow R \ast S) \land \text{noprotocol}(R) \land \text{nostate}(S) \]

- We require all channel pre- and post-conditions to be expressible using SIPs
Summary

• Verified the lock-based joins implementation against the high-level joins specification

• Verified a couple of classic synchronization primitives using the high-level joins specification

• Given a logic and model for HO CAP with support for state-independent higher-order protocols

• TRs available at www.itu.dk/~kasv
Questions?
Higher-order protocols in CAP

Let

\[ P \overset{\text{def}}{=} (x \mapsto 0 \ast (y \mapsto 0)_I^r \lor y \mapsto 0)_J^r)) \lor (x \mapsto 1 \ast y \mapsto 0)_J^r \]

where

\[ I[\alpha] : y \mapsto 1 \leadsto y \mapsto 2 \]
\[ J[\alpha] : y \mapsto 1 \leadsto y \mapsto 3 \]
\[ K[\alpha] : P \leadsto P \]

then \( P \) is stable, but \( P_K^{r'} \) is not