Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity

Ilya Sergey      Aleks Nanevski      Anindya Banerjee

ESOP 2015
A logic-based approach for Specifying and Verifying Concurrent Algorithms
An approach, which is
An approach, which is

- Natural
  - captures intuition behind realistic algorithms
An approach, which is

- **Natural**
  - captures intuition behind realistic algorithms

- **Powerful**
  - enables compositional verification of concurrency
An approach, which is

- Natural
  - captures intuition behind realistic algorithms

- Powerful
  - enables compositional verification of concurrency

- Lightweight
  - does not require to engineer a new logical framework
Key ideas

• Subjectivity

• Partial Commutative Monoids (PCMs)

• Histories
Key ideas

- Subjectivity
- Partial Commutative Monoids (PCMs)

Nanevski et al. [ESOP’14]

- Histories
Key ideas

• Subjectivity

• Partial Commutative Monoids (PCMs)

• Histories
Hoare-style program specifications
Hoare-style program specifications

\{ P \} \ c \ \{ Q \}
Hoare-style program specifications

\{ P \} \ c \ \{ Q \}

precondition
Hoare-style program specifications

\{ P \} \ c \ \{ Q \}

precondition  postcondition
Hoare-style program specifications

If the initial state satisfies $P$, then, after $c$ terminates, the final state satisfies $Q$. 

$\{ P \} \ c \ \{ Q \}$

precondition  postcondition
Abstract specifications for a stack

\texttt{push(x)}

\texttt{pop()}
Abstract specifications for a stack

\texttt{push(x)}

\texttt{pop()}
Abstract specifications for a stack

\[
\{ S = xs \} \quad \text{push}(x) \quad \{ S' = x :: xs \}
\]

\[
\text{pop}()
\]
Abstract specifications for a stack

\[
\{ S = xs \} \quad \text{push}(x) \quad \{ S' = x :: xs \}
\]

\[
\{ S = xs \} \quad \text{pop}( ) \quad \{ 
\begin{align*}
res &= \text{None} \land S = \text{Nil} \\
\lor \quad &\exists x, xs'. res = \text{Some } x \land \\
xs &= x :: xs' \land S' = xs'
\end{align*}
\}
\]
Abstract specifications for a stack

\{ S = xs \} \quad \textbf{push}(x) \quad \{ S' = x :: xs \} \\

\{ S = xs \} \quad \textbf{pop}( ) \quad \{ \begin{array}{l}
res = \text{None} \land S = \text{Nil} \\
\lor \exists x, xs'. res = \text{Some } x \land \\
x = x :: xs' \land S' = xs'
\end{array} \}

Suitable for sequential case
Abstract specifications for a stack

\[
\{ S = xs \} \quad \text{push}(x) \quad \{ S' = x :: xs \}
\]

\[
\{ S = xs \} \quad \text{pop}() \quad \{ \begin{align*}
res &= \text{None} \land S = \text{Nil} \\
\lor \exists x, xs'. \; res &= \text{Some } x \land \\
x &= x :: xs' \land S' = xs'
\end{align*} \}
\]

Not so good for concurrent use:
useless in the presence of interference
y := pop();
\{ S = \textbf{Nil} \} \\
y := \text{pop}();
\{ S = \texttt{Nil} \}

\texttt{y := pop();}

\{ \texttt{y = ???} \}
$y := \text{pop}()$; \hspace{1cm} \begin{align*}
\{ S = \text{Nil} \} & \quad \text{push}(1); \\
& \quad \text{push}(2); 
\end{align*}
\[
\begin{align*}
\{ \text{S = } \textbf{Nil} \} \\
y := \text{pop}(); \\
\{ y = 1 \lor y = 2 \lor y = \text{None} \} \\
\end{align*}
\]

\[
\begin{align*}
push(1); \\
push(2); \\
\end{align*}
\]
\[
\begin{align*}
    y & := \text{pop}() ; \\
    \begin{cases}
        \{ S = \text{Nil} \} \\
        \text{push(1)} ; \\
        \text{push(2)} ; \\
        \text{push(3)} ;
    \end{cases}
\end{align*}
\]
\[
\{ S = \text{Nil} \} \\
\]

\[
y := \text{pop}(); \\
\{ y = 1 \lor y = 2 \lor y = 3 \lor y = \text{None} \} \\
\]

\[
\text{push}(1); \quad \text{push}(2); \quad \text{push}(3); \\
\]

Thread-modular spec for pop?

\[
\{ S = \textbf{Nil} \}
\]

\[
y := \text{pop}();
\]

\[
\{ y = ??? \}
\]
Idea
Idea

Capture the effect of self, abstract over the others.
Idea

Capture the effect of *self*, abstract over the *others*.

*(subjective specification)*
Subjective stack specifications

\[ y := \text{pop}(); \]
Subjective stack specifications

- $H_s$ — pushes/pops to the stack by this thread

\[ y := \text{pop}(); \]
Subjective stack specifications

- $H_s$ — pushes/pops to the stack by this thread
- $H_o$ — pushes/pops by all other threads

\[
y := \text{pop}();
\]
Subjective stack specifications

- $H_s$ — pushes/pops to the stack by this thread
- $H_o$ — pushes/pops by all other threads

\[
\{ H_s = \emptyset \}
\]

\[
y := \text{pop}() ;
\]
Subjective stack specifications

• $H_s$ — pushes/pops to the stack by this thread
• $H_o$ — pushes/pops by all other threads

$$\{ H_s = \emptyset \}$$

$$y := \text{pop}();$$

$$\{ y = \text{None} \lor y = \text{Some}(v), \text{ where } v \in H_o \}$$
Subjective stack specifications

- $H_s$ — pushes/pops to the stack by this thread
- $H_o$ — pushes/pops by all other threads

\[
\{ H_s = \emptyset \}
\]

\[
y := \text{pop()};
\]

\[
\{ y = \text{None} \lor y = \text{Some}(v), \text{ where } v \in H_o \}
\]

what I popped depends on what the others have pushed
Subjective stack specifications

Valid only if the stack is changed only by push/pops.

\[ H_s = \emptyset \]

\[
\begin{align*}
y &:= \text{pop}() ; \\
\{ y = \text{None} \lor y = \text{Some}(v), \text{where } v \in H_o \} \\
\end{align*}
\]

what I popped depends on what the others have pushed
\{P\} y := \text{pop}(); \{Q\}
C \vdash \{ P \} \ y \ := \ \text{pop}() ; \ \{ Q \}
$C \vdash \{P\} \ y := \text{pop}(); \ \{Q\}$

Specifies expected thread interference
Concurrent Resources
Concurrent Resources

Shared state
Concurrent Resources

Owicki, Gries [CACM'77]

Shared state

Auxiliary state
Subjective Concurrent Resources

Ley-Wild, Nanevski [POPL’13]
Subjective Concurrent Resources

Ley-Wild, Nanevski [POPL’13]

Shared state

Auxiliary state, controlled by this thread
Subjective Concurrent Resources

Ley-Wild, Nanevski [POPL’13]

Auxiliary state, controlled by *others*

Shared state

Auxiliary state, controlled by *this* thread
Subjective Concurrent Resources

Jones [TOPLAS’83]
Subjective Concurrent Resources

Changes (transitions) allowed to myself (Guarantee)

Jones [TOPLAS'83]
Transitions, allowed to the *others* *(Rely)*

Changes (transitions) allowed to *myself* *(Guarantee)*

Subjective Concurrent Resources

Jones [TOPLAS’83]
Subjective Concurrent Resources

Transitions, allowed to the others (Rely)

Changes (transitions) allowed to myself (Guarantee)

What I have = what I can do and what I have done.
Concurrent Resources

= State Transition Systems with Subjective Auxiliary State
Concurrent Resources = State Transition Systems with Subjective Auxiliary State (Concurroids)
Specifications with concurroids
Specifications with concurroids

- *Self* — state controlled by *me*
Specifications with concurroids

- **Self** — state controlled by *me*
- **Other** — state controlled by *all other threads*
Specifications with concurroids

- **Self** — state controlled by *me*
- **Other** — state controlled by *all other threads*
- **Joint** — modified by everyone, as *allowed by transitions*
Specifications with concurroids

\[ C = \text{Self} \cup \text{Joint} \cup \text{Other} \]
Specifications with concurroids

C = \{ P \} \; c \; \{ Q \}

C \vdash \{ P \} \; c \; \{ Q \}
Specifications with concurroids

\[
C = \{ \text{Self} \} \circ \{ \text{Joint} \} \circ \{ \text{Other} \}
\]

\[
\{ P \} \circ \text{c} \circ \{ Q \} \circ \text{at} \circ C
\]
Specifications with concurroid\textsc{s}

C = \begin{array}{c}
\text{Self} \\
\text{Joint} \\
\text{Other}
\end{array}

\{ P \} c \{ Q \} @ C

defines resources, touched by c, their transitions and invariants
Specifications with concurroids

\[ C = \{ P \} c \{ Q \} @ C \]

specify self/other/joint parts
FCSL: Fine-grained Concurrent Separation Logic

Nanevski, Ley-Wild, Sergey, Delbianco [ESOP’14]
FCSL: Fine-grained Concurrent Separation Logic

Nanevski, Ley-Wild, Sergey, Delbianco [ESOP’14]

• Logic for reasoning with concurroids
FCSL: Fine-grained Concurrent Separation Logic

Nanevski, Ley-Wild, Sergey, Delbianco [ESOP’14]

• Logic for reasoning with concurroids

• Emphasis on subjective specifications
Key ideas

• Subjectivity

• PCMs

• Histories
Key ideas

• Subjectivity — reasoning with self and other

• PCMs

• Histories
Key ideas

- Subjectivity — reasoning with self and other

- PCMs

- Histories
Partial Commutative Monoids

\[(S, \oplus, 0)\]

- A set \(S\) of elements
- **Join** \((\oplus)\): commutative, associative, partial
- **Unit** element \(0\): \(\forall e \in S, e \oplus 0 = 0 \oplus e = e\)
Parallel composition
Parallel composition

parent

child_1

child_2
Parallel composition

- commutative
- associative
- unit — *idle* thread
- partial
Logical state split

\[
\begin{array}{c}
\text{parent} \\
\{ \ s_1 \oplus s_2 \ \}
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{child}_1 & \text{child}_2
\end{array}
\]
Logical state split

\[
\text{parent} \{ s_1 \oplus s_2 \}
\]

\[
s_1 \oplus s_2
\]

child_1 | | child_2
Logical state split

State that belongs to child$_1$

\[
\begin{align*}
&\{ s_1 \oplus s_2 \} \\
&\{ s_1 \} \\
&\text{parent} \\
\text{child}_1 & \text{||} & \text{child}_2
\end{align*}
\]
Logical state split

State that belongs to child$_2$
Logical state split

\[
\begin{align*}
z_1 \oplus z_3 \\
z_2
\end{align*}
\]

parent
\[
\{ s_1 \oplus s_2 \}
\]

\[
\begin{align*}
\{ s_1 \} & \quad \| \quad \{ s_2 \} \\
\{ z_1 \} & \quad \| \quad \{ z_2 \}
\end{align*}
\]
Logical state split

New state that belongs to parent’
Key ideas

- Subjectivity — reasoning with self and other
- PCMs
- Histories
Key ideas

• Subjectivity — reasoning with *self* and *other*

• PCMs — uniform way to logically *split* state

• Histories
Familiar PCM: finite heaps
Familiar PCM: finite heaps

- Heaps are partial finite maps \( \text{nat} \rightarrow \text{Val} \)
Familiar PCM: finite heaps

• Heaps are partial finite maps $\text{nat} \rightarrow \text{Val}$

• Join operation $\oplus$ is disjoint union
Familiar PCM: finite heaps

• Heaps are partial finite maps $\text{nat} \rightarrow \text{Val}$

• Join operation $\oplus$ is disjoint union

• Unit element $0$ is the empty heap $\emptyset$
Concurroid for thread-local state
Concurroid for thread-local state

- $h_s$ — heap, logically owned by this thread
Concurroid for thread-local state

- $h_s$ — heap, logically owned by this thread
- $h_o$ — heap, owned by others
Concurroid for thread-local state

- $h_s$ — heap, logically owned by this thread
- $h_o$ — heap, owned by others
\[*x := 5;\] \[\quad\] \[\quad\] \[\quad\] \[*y := 7;\]
\{ h_s = x \mapsto - \oplus y \mapsto - \wedge h_o = h \}
disjoint by resource definition

\{ h_s = x \mapsto - \oplus y \mapsto - \land h_o = h \}
disjoint by resource definition

\[
\begin{align*}
\{ h_s &= x \mapsto - \oplus y \mapsto - \land h_o = h \} \\
\{ h_s &= x \mapsto - \land h_o = y \mapsto ? \oplus h \} \\
* x & := 5 ; \\
* y & := 7 ;
\end{align*}
\]
\[
\begin{align*}
\{ h_s = x & \mapsto - \oplus y \mapsto - \land h_o = h \} \\
\{ h_s = x & \mapsto - \land h_o = y \mapsto ? \oplus h \} \\
\{ h_s = y & \mapsto - \land h_o = x \mapsto ? \oplus h \}
\end{align*}
\]

*\( x := 5 \); 
\*\( y := 7 \);

disjoint by resource definition
disjoint by resource definition

\[
\{ h_s = x \mapsto - \oplus y \mapsto - \land h_o = h \}
\]

\[
\{ h_s = x \mapsto - \land h_o = y \mapsto ? \oplus h \}
\]

\[
\{ h_s = y \mapsto - \land h_o = x \mapsto ? \oplus h \}
\]

\[
\{ h_s = x \mapsto 5 \land h_o = y \mapsto ? \oplus h \}
\]

\[
\{ h_s = y \mapsto 7 \land h_o = x \mapsto ? \oplus h \}
\]
disjoint by resource definition

\{ h_s = x \mapsto - \oplus y \mapsto - \wedge h_o = h \}

\{ h_s = x \mapsto - \wedge h_o = y \mapsto ? \oplus h \}
\{ h_s = y \mapsto - \wedge h_o = x \mapsto ? \oplus h \}

\{ h_s = x \mapsto 5 \wedge h_o = y \mapsto ? \oplus h \}
\{ h_s = y \mapsto 7 \wedge h_o = x \mapsto ? \oplus h \}

\{ h_s = x \mapsto 5 \oplus y \mapsto 7 \wedge h_o = h \}
Key ideas

- Subjectivity — reasoning with self and other
- PCMs — uniform way to logically split state
- Histories
Key ideas

- Subjectivity — reasoning with self and other

- PCMs — uniform way to logically split state

- Histories
Key ideas

• Subjectivity — reasoning with self and other

• PCMs — uniform way to logically split state

• Histories

Sergey et al. [ESOP’15]
Atomic stack specifications

push(x)
Atomic stack specifications

\{ S = xs \} \quad \text{push}(x) \quad \{ S' = x :: xs \}
Atomic stack specifications

t_k \rightarrow \begin{array}{|c|c|}
  \hline
  XS & x :: xs \\
  \hline
\end{array}
Atomic stack specifications

$t_k \rightarrow XS \quad x :: XS$

“timestamp”
$t_k \rightarrow$
Changes by *this* thread

\[ t_k \rightarrow \]

\[ t_{k+2} \rightarrow \]

\[ \ldots \]

Changes by *other* threads

\[ t_{k+1} \rightarrow \]

\[ t_{k+3} \rightarrow \]

\[ t_{k+4} \rightarrow \]

\[ t_{k+n} \rightarrow \]
$H_s, H_o — \text{self/other} \text{ contributions to the resource history}$
Histories are like heaps!
Histories are like heaps!

- Histories are partial finite maps \( \text{nat} \to \text{AbsOp} \)
Histories are like heaps!

- Histories are partial finite maps $\text{nat} \rightarrow \text{AbsOp}$
- Join operation $\oplus$ is disjoint union
Histories are like heaps!

- Histories are partial finite maps $\text{nat} \rightarrow \text{AbsOp}$
- Join operation $\oplus$ is disjoint union
- Unit element $0$ is the empty history $\emptyset$
Specifying stacks with histories

\[ C_{\text{stack}} = H_S \cup H_0 \]
Specifying stacks with histories

\[ C_{stack} = \]

- \( H_s, H_o = \{ t_k \mapsto (xs, x::xs), t_n \mapsto (x::xs, xs), \ldots \} \)
Specifying stacks with histories

\[ C_{\text{stack}} = \]

\[ H_s, H_o = \{ t_k \mapsto (xs, x::xs), t_n \mapsto (x::xs, xs), \ldots \} \]

\[ \text{Joint part is specific for each implementation} \]
Specifying stacks with histories

\[ C_{\text{stack}} = \]

\[ \begin{align*}
H_s &= \{ t_k \mapsto (xs, x::xs), t_n \mapsto (x::xs, xs), \ldots \} \\
H_o &= \{ \}
\end{align*} \]

- \( H_s, H_o = \{ t_k \mapsto (xs, x::xs), t_n \mapsto (x::xs, xs), \ldots \} \)
- *Joint* part is specific for each implementation
- Adjacent history entries agree on overlapping abstract states
Stack specification

\{ H_s = \emptyset \land H \subseteq H_0 \}\}

push(x)

\{ \exists t, xs. H_s = t \mapsto (xs, x::xs) \land H \subseteq H_0 \land H < t \}\}
Stack specification

self-contribution is a single entry

\[
\{ \exists t, xs. H_s = t \mapsto (xs, x::xs) \land H \subseteq H_o \land H < t \} \]

\[
\{ H_s = \emptyset \land H \subseteq H_o \} \]

\text{push}(x)
Stack specification

\[
\begin{align*}
\{ & H_s = \emptyset \land H \subseteq H_0 \} \\
& \text{push(} x \text{)} \\
\{ & \exists t, xs. \ H_s = t \mapsto (xs, x::xs) \land H \subseteq H_0 \land H < t \} @ C_{\text{stack}}
\end{align*}
\]
Stack specification

\[
\{ H_s = \emptyset \land H \subseteq H_0 \} \]

\[
\text{pop( )}
\]

\[
\{ \text{res. if (res = Some x)} \}
\]

\[
\text{then } \exists t, xs. H \subseteq H_0 \land H < t \land H_s = t \mapsto (x::xs, xs)
\]

\[
\text{else } \exists t. H \subseteq H_0 \land H \leq t
\]

\[
\land H_s = \emptyset \land t \mapsto (_, \text{Nil}) \subseteq H_0 \}
\]
Stack specification

\[
\{ H_s = \emptyset \land H \subseteq H_0 \}
\]

\text{pop( )}

\[
\{ \text{res. if } (\text{res} = \text{Some } x) \}
\]

\text{then } \exists t, xs. H \subseteq H_0 \land H < t \land H_s = t \mapsto (x::xs, xs)

\text{else } \exists t. H \subseteq H_0 \land H \leq t

\land H_s = \emptyset \land t \mapsto (\_, \text{Nil}) \subseteq H_0 \}

• \text{pop has hit Nil during its execution at the moment } t
Stack specification

no self-contributions initially?

\{ H_s = \emptyset \land H \subseteq H_o \}\

\text{pop( )}

\{ \text{res. if (res = Some x)} \}

\begin{align*}
\text{then } & \exists t, xs. H \subseteq H_o \land H < t \land H_s = t \mapsto (x::xs, xs) \\
\text{else } & \exists t. H \subseteq H_o \land H \leq t \\
& \land H_s = \emptyset \land t \mapsto (\_, \text{Nil}) \subseteq H_o \} @ C_{\text{stack}}
\end{align*}
Framing in FCSL
Framing in FCSL

my_program
Framing in FCSL

my_program
Framing in FCSL

my_program

{ }

{ }

{ }

{ }
Framing in FCSL

my_program

```python
{ my_program }
```
Framing in FCSL

Works for *any* PCM, not just heaps!
Framing histories

\{ \exists t, xs. H \subseteq H_0 \}

\text{push}(x)

\{ \exists t, xs. H \subseteq H_0 \land H < t \land H_s = t \mapsto (xs, x::xs) \} @ C_{\text{stack}}
Framing histories

\{ \exists t, xs. H_2 \subseteq H_o \}

\textbf{push}(x)

\{ \exists t, xs. H_2 \subseteq H_o \land H_1 \oplus H_2 < t \land H_s = H_1 \oplus t \mapsto (xs, x::xs) \} @ C_{\text{stack}}
Framing histories

\{ H_s = H_1 \land H_2 \subseteq H_0 \}

push(x)

\{ \exists t, xs. H_2 \subseteq H_0 \land H_1 \oplus H_2 < t \land H_s = H_1 \oplus t \mapsto (xs, x::xs) \} \mathcal{C}_{\text{stack}}
Framing histories

\[
\{ H_s = H_1 \land H_2 \subseteq H_0 \}
\]

push(\(x\))

\[
\{ \exists t, xs. H_2 \subseteq H_0 \land H_1 \oplus H_2 < t \land H_s = H_1 \oplus t \mapsto (xs, x::xs) \}
\]
Key ideas

• Subjectivity — reasoning with self and other

• PCMs — uniform way to logically split state

• Histories
Key ideas

• Subjectivity — reasoning with self and other

• PCMs — uniform way to logically split state

• Histories — logical updates via auxiliary state
How useful are histories for clients?
A stack client program

• Two threads: producer and consumer

• Ap — an n-element producer array

• Ac — an n-element consumer array

• A shared concurrent stack $S$ is used as a buffer

• The goal: prove the exchange correct
Auxiliary Predicates

- **Pushed** $HE$ iff $E$ is a multiset of elements, *pushed* in $H$

- **Popped** $HE$ iff $E$ is a multiset of elements, *popped* in $H$
letrec produce(i : nat) = {
  if (i == n)
    then return;
  else {
    S.push(Ap[i]);
    produce(i+1);
  }
}
letrec produce(i : nat) = {
    if (i == n)
    then return;
    else {
        S.push(Ap[i]);
        produce(i+1);
    }
}

{ Ap \rightarrow L \land \text{Pushed } H_s \, L[< i] \land \text{Popped } H_s \emptyset }
letrec produce(i : nat) = {
  if (i == n)
  then return
  else {
    S.push(Ap[i]);
    produce(i+1);
  }
}

{ Ap \leftrightarrow L \land \textbf{Pushed } H_s L[< i] \land \text{Popped } H_s \emptyset }

{ Ap \leftrightarrow L \land \textbf{Pushed } H_s L[< n] \land \text{Popped } H_s \emptyset }
letrec consume(i : nat) = {
  if (i == n)
  then return;
  else {
    t ← S.pop();
    if t == Some v
    then {
      Ac[i] := v;
      consume(i+1);
    }
    else consume(i);
  }
}

\[
\{ \exists L, \text{Ac} \mapsto L \land \text{Pushed } H_s \emptyset \land \text{Popped } H_s L[< i] \}
\]

```ocaml
letrec consume(i : nat) = {
  if (i == n)
  then return;
  else {
    t ← S.pop();
    if t == Some v
    then {
      Ac[i] := v;
      consume(i+1);
    }
    else consume(i);
  }
}
```

\[
\{ \exists L, \text{Ac} \mapsto L \land \text{Pushed } H_s \emptyset \land \text{Popped } H_s L[< n] \}
\]
letrec consume(i : nat) = {
  if (i == n)
    then return;
  else {
    t ← S.pop();
    if t == Some v
      then {
        Ac[i] := v;
        consume(i+1);
      } 
    else consume(i);
  }
}

{∃L, Ac ↦ L ∧ Pushed Hₛ ∅ ∧ Popped Hₛ L[< i]}
produce(0) | consume(0)
\[ \text{hide } C_{\text{stack}}(h_S) \text{ in } \{ \begin{align*} \text{produce}(0) \quad & \mid \quad \text{consume}(0) \end{align*} \} \]
No other threads can interfere on $S$

$\text{hide } C_{\text{stack}}(h_S) \text{ in }$

\{
  \begin{align*}
    \text{produce}(0) & \quad \| \quad \text{consume}(0)
  \end{align*}
\}
\[ \{ \text{Ap} \mapsto L \oplus \text{Ac} \mapsto L' \oplus h_S \} \]

hide \( C_{\text{stack}}(h_S) \) in

\[
\{ \text{produce}(0) \quad \text{consume}(0) \}
\]
\[
\{ \text{Ap} \leftrightarrow L \oplus \text{Ac} \leftrightarrow L' \oplus h_S \}
\]

\text{hide } C_{\text{stack}}(h_S) \text{ in }

\{
\begin{align*}
\{ \text{Ap} \leftrightarrow L \\
\text{produce}(0)
\end{align*}
\} \quad \text{||} \quad \{
\begin{align*}
\{ \text{Ac} \leftrightarrow L' \\
\text{consume}(0)
\end{align*}
\}
\[ \{ \text{Ap} \mapsto L \oplus \text{Ac} \mapsto L' \oplus h_S \} \]

given \( h_S \)

\[ \begin{align*}
\text{produce}(0) \quad \text{consume}(0)
\end{align*} \]
\[
\begin{align*}
\{ \text{Ap} \mapsto L \oplus \text{Ac} \mapsto L' \oplus h_S \} \\
\text{hide } C_{\text{stack}}(h_S) \text{ in }
\end{align*}
\]

\[
\begin{align*}
\{ \text{Ap} \mapsto L \land \text{Pushed } H_s \emptyset \land \text{Popped } H_s \emptyset \} & \quad \{ \text{Ac} \mapsto L' \land \text{Pushed } H_s \emptyset \land \text{Popped } H_s \emptyset \} \\
\text{produce}(0) & \quad \text{consume}(0) \\
\{ \text{Ap} \mapsto L \land \text{Pushed } H_s \text{L}[< n] \land \text{Popped } H_s \emptyset \} & \quad \{ \text{Ac} \mapsto L'' \land \text{Pushed } H_s \emptyset \land \text{Popped } H_s \text{L''}[<n] \} \\
\end{align*}
\]
These are the only changes in the stack's history.
\[
\{ \text{produce}(0) \} \quad \quad \{ \text{consume}(0) \}
\]

\[
\{ \text{Ap} \mapsto L \land \text{Pushed } H_s \emptyset \land \text{Popped } H_s \emptyset \} \quad \{ \text{Ac} \mapsto L' \land \text{Pushed } H_s \emptyset \land \text{Popped } H_s \emptyset \}
\]

\[
\{ \text{Ap} \mapsto L \land \text{Pushed } H_s L[<n] \land \text{Popped } H_s \emptyset \} \quad \{ \text{Ac} \mapsto L'' \land \text{Pushed } H_s \emptyset \land \text{Popped } H_s L''[<n] \}
\]

\[
\{ \text{Ap} \mapsto L \oplus \text{Ac} \mapsto L' \oplus h_s \land L =_{\text{set}} L'' \}
\]
\[
\{ \text{Ap} \mapsto L \oplus \text{Ac} \mapsto L' \oplus h_s \} \\
\text{hide } \mathcal{C}_{\text{stack}}(h_s) \text{ in } \\
\{ \text{Ap} \mapsto L \land \text{Pushed } H_s \emptyset \land \text{Popped } H_s \emptyset \} \quad \text{produce}(0) \quad \{ \text{Ac} \mapsto L' \land \text{Pushed } H_s \emptyset \land \text{Popped } H_s \emptyset \} \\
\{ \text{Ap} \mapsto L \land \text{Pushed } H_s L[<n] \land \text{Popped } H_s \emptyset \} \\
\{ \text{Ac} \mapsto L'' \land \text{Pushed } H_s \emptyset \land \text{Popped } H_s L''[<n] \} \\
\{ \text{Ap} \mapsto L \oplus \text{Ac} \mapsto L'' \oplus h_s' \land L =_{\text{set}} L'' \} \]
More use for histories
(see the paper)
More use for histories

- Verifying atomic snapshots

(see the paper)
More use for histories

(see the paper)

• Verifying **atomic snapshots**

• Instantiating **higher-order** concurrent structures
More use for histories
(see the paper)

• Verifying atomic snapshots

• Instantiating higher-order concurrent structures

• Deriving sequential specifications via hiding
More use for histories

(see the paper)

- Verifying *atomic snapshots*
- Instantiating *higher-order* concurrent structures
- Deriving *sequential* specifications via hiding
To take away
To take away

• **Histories** as auxiliary state
  - Expressive abstraction for concurrent specs
To take away

• **Histories** as auxiliary state
  - Expressive abstraction for concurrent specs

• **Histories are a PCM**
  - They are subject of the same rules as heaps
To take away

- **Histories as auxiliary state**
  - Expressive abstraction for concurrent specs

- **Histories are a PCM**
  - They are subject of the same rules as heaps

- **Historical reasoning requires subjectivity**
  - History-based specs often talk about the effect of *other* threads
To take away

- **Histories** as auxiliary state
  - Expressive abstraction for concurrent specs

- **Histories are a PCM**
  - They are subject of the same rules as heaps

- **Historical reasoning requires subjectivity**
  - History-based specs often talk about the effect of other threads

software.imdea.org/fcs1
To take away

• **Histories** as auxiliary state
  - Expressive abstraction for concurrent specs

• **Histories are a PCM**
  - They are subject of the same rules as heaps

• **Historical reasoning requires subjectivity**
  - History-based specs often talk about the effect of other threads

software.imdea.org/fcs1

Thanks!
Q&A slides
How is your stuff different from other existing concurrent logics?

- [Owicki-Gries: CACM76] - reasoning about parallel composition is not compositional; subjectivity fixes that;
- [O'Hearn: CONCUR04] - only one type of resources — critical sections; FCSL allows one to define arbitrary resources;
- [Feng-al: ESOP07, Vafeiadis-Parkinson: CONCUR07] - framing over Rely/Guarantee, but only one shared resource: FCSL allows multiple ones;
- [Feng: POPL09] - introduced local Rely/Guarantee; FCSL improves on it by introducing a subjective state and explicitly identifying resources as STS;
- [Dinsdale-Young-al: ECOOP10] - first introduced concurred protocols; FCSL generalises permissions - self-state defines what a thread is allowed to do with a resource;
- [Dinsdale-Young-al: POPL13] - general framework for concurrency logic; FCSL is a particular logic, not clear whether it is an instance of Views;
- [Turon-al: ICFP13] - CaReSL and reasoning about contextual refinement; FCSL doesn’t address CR, in our experience it’s never required for Hoare-style reasoning;
- [Svendsen-al: ESOP13, ESOP14] - use much richer semantic domain, FCSL uses transitions and communication instead of view-shifts for changes in state and composition of resources;
- [Raad-al: ESOP15] - different notion of subjectivity, no self/other dichotomy, no observation made about PCMs.

FCSL’s assertions work explicitly with state variables.
How is your stuff different from Iris?

• Iris makes the same observations as FCSL did in 2014 (PCMs, Invariants);

• Iris doesn’t have hiding and self/other dichotomy;

• It considers more primitive “building blocks” and encodes protocols as STSs + interpretation;
  
  • This encoding is made default in FCSL, and so far it suffices;

• Currently, FCSL doesn’t support abstract atomicity in Iris/iCAP sense (however, it can recover most of it through the choice of PCMs).
Encoding verification in FCSL
Encoding verification in FCSL

Program Definition my_prog: STSep (p, q) :=
Do c.
Encoding verification in FCSL

Program Definition  my_prog: STSep (p, q) :=
Do c

has type STSep (p*, q*)

• Program c’s weakest pre- and strongest postconditions (p*, q*) wrt. safety, inferred from the types of basic commands (ret, par, bind);
Encoding verification in FCSL

**Program Definition**

```
my_prog: STSep (p, q) :=
  Do c
```

- Program c’s *weakest pre-* and *strongest postconditions* \((p^*, q^*)\) wrt. safety, inferred from the types of basic commands \((\text{ret}, \text{par}, \text{bind})\);

- `Do` encodes the application of the rule of consequence \((p^*, q^*) \sqsubseteq (p, q)\);
Encoding verification in FCSL

Program Definition

\begin{verbatim}
my_prog: STSep (p, q) :=
  Do c
\end{verbatim}

Notation for \texttt{do (_ : (p*, q*) \sqsubseteq (p, q)) c}

- Program c’s \textit{weakest pre-} and \textit{strongest postconditions} \((p*, q*)\) \textit{wrt. safety}, inferred from the types of basic commands \((\text{ret, par, bind})\);
- \texttt{Do} encodes the application of the rule of consequence \((p*, q*) \sqsubseteq (p, q)\);
- The client constructs the proof of \((p*, q*) \sqsubseteq (p, q)\) interactively;
Encoding verification in FCSL

**Program Definition** my_prog: STSep (p, q) :=

\[ \text{Do } c \]

Notation for \( \text{do } (_ : (p^*, q^*) \sqsubseteq (p, q)) \ c \)

- Program c’s *weakest pre- and strongest postconditions* \((p^*, q^*)\) wrt. safety, inferred from the types of basic commands (ret, par, bind);
- Do encodes the application of the rule of consequence \((p^*, q^*) \sqsubseteq (p, q)\);
- The client constructs the proof of \((p^*, q^*) \sqsubseteq (p, q)\) interactively;
- The obligations are reduced via *structural lemmas* (inference rules).
### Implementation and evaluation

In principle, we could implement an abstract lock interface, which is used to implement and verify the lock interface, which is then employed by a Treiber stack, used as a basis for sequential stack and producer/consumer implementations. In the spirit of linearizability [Herlihy and Wing 1985], as well as stability-related lemmas, while the sizes of proofs of properties of transitions and actions, such as several client programs: a sequential stack (obtained from the Treiber stack), a large fraction of an implementor's commits, and build times (without any advanced proof automation in the proof scripts, as is done only for libraries, so library clients can reason with consistent use of the concurroid for thread-local state and locks). With no specific focus on abstractions for fine-grained concurrency, such as protocols and auxiliary state, although, with no specific focus on abstractions for fine-grained concurrency, such as protocols and auxiliary state, although, we didn’t carry out this exercise.

### Table 1: Statistics for implemented programs: lines of code

<table>
<thead>
<tr>
<th>Program</th>
<th>Libs</th>
<th>Conc</th>
<th>Acts</th>
<th>Stab</th>
<th>Main</th>
<th>Total</th>
<th>Build</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAS-lock</td>
<td>63</td>
<td>291</td>
<td>509</td>
<td>358</td>
<td>27</td>
<td>1248</td>
<td>1m 1s</td>
</tr>
<tr>
<td>Ticketed lock</td>
<td>58</td>
<td>310</td>
<td>706</td>
<td>457</td>
<td>116</td>
<td>1647</td>
<td>2m 46s</td>
</tr>
<tr>
<td>Increment</td>
<td>26</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>44</td>
<td>70</td>
<td>8s</td>
</tr>
<tr>
<td>Allocator</td>
<td>82</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>192</td>
<td>274</td>
<td>14s</td>
</tr>
<tr>
<td>Pair snapshot</td>
<td>167</td>
<td>233</td>
<td>107</td>
<td>80</td>
<td>51</td>
<td>638</td>
<td>4m 7s</td>
</tr>
<tr>
<td>Treiber stack</td>
<td>56</td>
<td>323</td>
<td>313</td>
<td>133</td>
<td>155</td>
<td>980</td>
<td>2m 41s</td>
</tr>
<tr>
<td>Spanning tree</td>
<td>348</td>
<td>215</td>
<td>162</td>
<td>217</td>
<td>305</td>
<td>1247</td>
<td>1m 11s</td>
</tr>
<tr>
<td>Flat combiner</td>
<td>92</td>
<td>442</td>
<td>672</td>
<td>538</td>
<td>281</td>
<td>2025</td>
<td>10m 55s</td>
</tr>
<tr>
<td>Seq. stack</td>
<td>65</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>125</td>
<td>190</td>
<td>1m 21s</td>
</tr>
<tr>
<td>FC-stack</td>
<td>50</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>114</td>
<td>164</td>
<td>44s</td>
</tr>
<tr>
<td>Prod/Cons</td>
<td>365</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>243</td>
<td>608</td>
<td>2m 43s</td>
</tr>
</tbody>
</table>
Proof of push specification
Proof of push specification

Next Obligation.

apply: gh=>i [h hS] B[v] X C.
case: C (C) X =>_ [x][xl][>][x][xt][->] Cp Ct Ca C [P S H].
rewrite (getC Cp C) !(getC Ct C) /= in P S H.
rewrite -!joinA joinCA in C *.
apply: step; apply: val_extend=>//; apply: (gh_ex h); apply: val_do=>//.
move=>p' {xt S H C Ct Ca} xt [t][K] S H _ Ct s C M.
case: {M} (menvs_coh M) (M) C=>_ [x][xl][xl][->][s] Cpl Ca M C.
case/menvs_split (injLE _ (erefl _)) Cpl C: M=>/= M _.
have {M P} P : pv_self xpl = p -> v \+ hS by rewrite -(menvs_loc M).
rewrite -joinCA in C *.
apply: step; apply: val_extend=>//.
apply: (gh_ex hS); apply: val_do; first by exists B, v.
case=>[xp xpl xa C Cp Cp1 Ca P] xp P Cp s /= C M.
case: {M} (menvs_coh M) (M) C=>_ [x][xl][xl][->][s] Cpl Ca M C.
case/menvs_split (injLA _ (erefl _)) Ct Ct1: M=>/= {xal} M _.
have S S : tb_self xt1 = Unit by rewrite -(menvs_loc M).
have {xt Ct M H} H : [h <<= tb_other xt1] by apply: hist_trans H (hist_other M).
rewrite joinA in C *.
apply: step; apply: val_extend; first by apply/(star_coh_prec C).
apply: (gh_ex h); apply: val_do.
- move=>C'; rewrite (getC Cp C') !(getC Ct C').
move=>b s X Y; case: Y (Y) X=>_ [x][x][xt][->][s] {Cp} Cpl Ct Y X.
moves=xal /= {C} C M; case: {menvs_coh M}=>_ {M xal Ca} Cal.
rewrite (getC Cp1 Y) !(getC Ct Y) in X.
case: b X; last first.
- case=>(P S H) P S H.
apply: (gh_ex h); apply: (gh_ex hS); apply: val_do=>//.
move=>(C) C; exists (prod A ptr), {e, p'}.
by rewrite (getC Cp1 C) !(getC Ct C).
case=>[xp xt1 Ct1 P K S H] t' [ls][P K S H].
apply: val_ret=>//= m M; rewrite -(menvs_loc M).
rewrite (getC Cp1 C) !(getC Ct C).
exists t', ls; split=>//.
case: {M} (menvs_coh M) (M)=/= X'.
case: X' (X') =>_ [x][xl][->][m] Y'; case: Y' (Y') =>_ [x][xt][->].
moves=Cp Ct1 Y' {Cal} Ca {C} C.
case/menvs_split (injLE _ (erefl _)) Y Y'.
case/menvs_split (injLE _ (erefl _)) Cp1 Cp= _ M _.
by rewrite (getC Ct1 C); apply: hist_trans H (hist_other M).
Proof of push specification

apply: gh=>i [h hS] B[v] X C.

```plaintext
apply: val_extend; apply: (gh_ex h); apply: (gh_ex hS); apply: val_do; first by exists B, v.
```

```plaintext
case: {M} (menvs_coh M) (M) =>_ [{m} {xp} {xt}]-> C M C = S H C T C a x t = t [K] S H _ Ct s C M.
```

```plaintext
case: X (X) Y=>_ [{xp} {xt}]-> /= C Y X.
```

```plaintext
move=>p' {xt S H C Ct Ca} xt [t] S H C Ct Ca C M. 
```

```plaintext
apply: step; apply: val_extend; apply: (gh_ex h); apply: (gh_ex hS); apply: val_do.
```

```plaintext
move=>x m [B[v] Y X C Ca C].
```

```plaintext
apply: (gh_ex hS); apply: val_do; first by exists B, v.
```

```plaintext
case=>{xp xp1 xa C Cp Cp1 Ct Y X} xp P Cp s /= C M.
```

```plaintext
case: {M} (menvs_coh M) (M) =>_ [{m} {xp1} {xt1}]-> /= C M C.
```

```plaintext
rewrite (getC Cp C) !(getC Ct C) /= in P S H.
```

```plaintext
rewrite -(menvs_loc M). 
```

```plaintext
apply: step; apply: val_extend; first by apply/(star_coh_prec C).
```

```plaintext
apply: (gh_ex (pv_self xp)).
```

```plaintext
case=>{P S H} P S H.
```

```plaintext
apply: (gh_ex h); apply: (gh_ex (pv_self xp)).
```

```plaintext
apply: val_do; first by exists B, v; rewrite!(getC Ct C).
```

```plaintext
case=>m [t][ls][P2 S2 H2 K2]; exists t, ls; rewrite P2; split=>//=.
```

```plaintext
Qed.
```

```plaintext
rewrit
```
Proof of push specification

apply: gh=>i [h hS] X C.
case: C (C) X =>_ _ [x]a [->i] _ [xp] [xt] [->] Cp Ct Ca C [P S H].
rewrite !:(getC Ct C) /= in P S H.
rewrite A joinCA in C *.
apply: val_extend=>//; apply: (gh_ex h);
apply: X S H C Ct Ca Xt [t] [K] S H _ Ct S C M.
case: (menvs_split (injLE _ (erefl _)) C) Cp1 Ca M C.
have {M P} P : pv_self xp1 = p -> v \+ hS by rewrite -(menvs_loc M).
rewrite joinCA in C *.
apply: val_extend=>//;
apply: (gh_ex hS);
appliy: (gh_ex (pv_self xp)); first by exists B, v.
case=> {xp1} _ Ct1 Ct S H C X.
case: (menvs_sp spl (injLE _ (erefl _)) C) Cp1 Ct1 C Ct Ca C M.
have {S} S : tb_self xt1 = Unit by rewrite -(menvs_loc M).
have {xt Ct M H} H : [h <<= tb_other xt1] by apply: hist_trans H (hist_o M).
apply: (gh_ex h);
appliy: gh_ex hS;
appliy: val_do; first by exists B, v; rewrite !(getC Ct C).
case=> m [t] [ls] [P2 S2 H2 K2] ; exists t, ls; rewrite P2; split=>//.
Qed.
Why do you need the explicit other? Other logics don’t have it!

- *Other* makes it possible to state *open-world assumptions* in a straightforward way (e.g., in `push`);

- It allows us to use *hiding* for uniformly cancelling the interference;

- Some algorithms are given more natural specs via *other*-contributions (e.g., stack’s `pop` and atomic snapshots).
Composing concurrent resources
Composing concurrent resources

Connect ownership-transferring transitions with right polarity
Composing concurrent resources

Connect ownership-transferring transitions with right polarity

- Some channels might be left loose
- Some channels might be shut down
- *Same* channels might be connected several times
Can you extract the verified program from your Coq implementation and run it?
Can you extract the verified program from your Coq implementation and run it?

Not yet.
Can you extract the verified program from your Coq implementation and run it?

Not yet.

• Imperative programs are composed and verified (i.e., type-checked) by means of Coq;

• They cannot be run by means of Gallina’s operational semantics;

• The reason for that is the necessity to reason about concurrent computations and potentially diverging programs;

• Extraction will require proving operationally of arbitrary atomic actions.