SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependent Programs

Nikhil Swamy, Aseem Rastogi, **Aymeric Fromherz**,
Denis Merigoux, Danel Ahman, Guido Martinez

ICFP 2020

Verifying Concurrent Programs

- Lots of recent work on using Concurrent Separation Logic (CSL) for verification

Verifying Concurrent Programs

- Lots of recent work on using Concurrent Separation Logic (CSL) for verification
  - Iris: Comprehensive, expressive logic. But applies to deeply embedded, simply-typed languages

Verifying Concurrent Programs

- Lots of recent work on using Concurrent Separation Logic (CSL) for verification
  - Iris: Comprehensive, expressive logic. But applies to deeply embedded, simply-typed languages
- How to get a CSL for a dependently-typed language? Through a shallow embedding?

Verifying Concurrent Programs

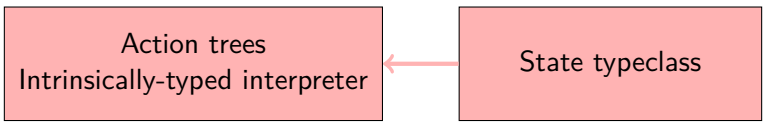
- Lots of recent work on using Concurrent Separation Logic (CSL) for verification
  - Iris: Comprehensive, expressive logic. But applies to deeply embedded, simply-typed languages
- How to get a CSL for a dependently-typed language? Through a shallow embedding?

Challenges

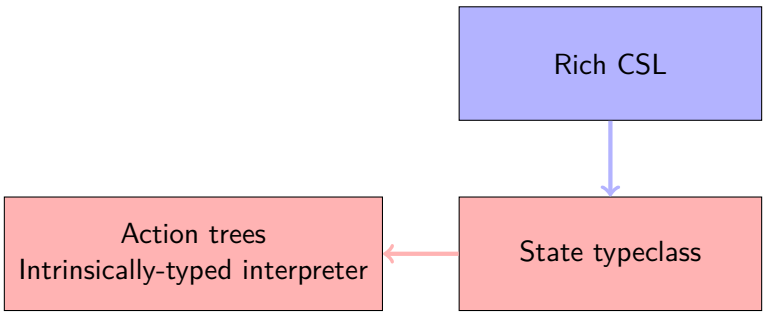
- How to reflect the effect of concurrency in the language?
- How to support partial correctness?
- How to enable dynamically allocated invariants?

Steel: A Concurrent Separation Logic (CSL) for F*

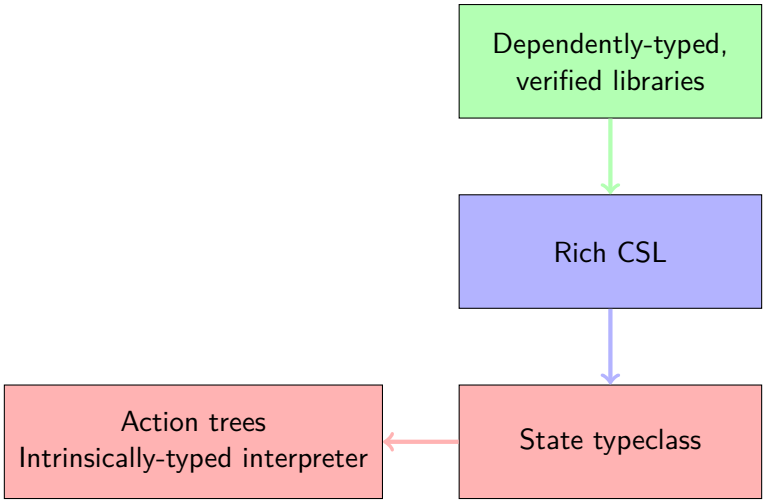
Steel: A Concurrent Separation Logic (CSL) for F*



Steel: A Concurrent Separation Logic (CSL) for F*



Steel: A Concurrent Separation Logic (CSL) for F*



Encoding Computations through Effectful Indexed Action Trees

$$\begin{aligned}\texttt{type state} = \{&\texttt{mem: Type;}\\ &\texttt{slprop:Type; equals; emp; star;}\\ &\texttt{interp: slprop} \to \texttt{mem} \to \texttt{prop}\}\end{aligned}$$

Encoding Computations through Effectful Indexed Action Trees

$$\texttt{type state} = \{\texttt{mem: Type};\ \texttt{slprop:Type; equals; emp; star;}\ \texttt{interp: slprop} \to \texttt{mem} \to \texttt{prop}\}$$

$$\texttt{type ctree (st:state) : a:Type} \to \texttt{pre:st.slprop} \to \texttt{post:(a} \to \texttt{st.slprop)} \to \texttt{Type} =$$

Encoding Computations through Effectful Indexed Action Trees

```
type state = {mem: Type;
              slprop:Type; equals; emp; star;
              interp: slprop → mem → prop}

type ctree (st:state) : a:Type → pre:st.slprop → post:(a → st.slprop) → Type =
 | Ret : y:a → ctree st a (post y) post
 | Act : action a pre post → ctree st a pre post
```

Encoding Computations through Effectful Indexed Action Trees

```
type state = {mem: Type;
              slprop:Type; equals; emp; star;
              interp: slprop → mem → prop}

type ctree (st:state) : a:Type → pre:st.slprop → post:(a → st.slprop) → Type =
 | Ret : y:a → ctree st a (post y) post
 | Act : action a pre post → ctree st a pre post
 | Bind : ctree st a p q → ((y:a) → Dv (ctree st b (q y) r)) → ctree st b p r
```

Encoding Computations through Effectful Indexed Action Trees

```
type state = {mem: Type;
              slprop:Type; equals; emp; star;
              interp: slprop → mem → prop}

type ctree (st:state) : a:Type → pre:st.slprop → post:(a → st.slprop) → Type =
  | Ret : y:a → ctree st a (post y) post
  | Act : action a pre post → ctree st a pre post
  | Bind : ctree st a p q → ((y:a) → Dv (ctree st b (q y) r)) → ctree st b p r
  | Par : ctree st a p q → ctree st a' p' q' → ctree st (a & a') (p `st.star` p') (λ (y, y') →
q y `st.star` q' y')
```

Proving Soundness of the Semantics

- We propose an intrinsically-typed definitional interpreter
- Atomic actions are non-deterministically interleaved
- The type of the interpreter states its soundness

```
val run (e:ctree st a p q) : NST a
  (requires λm → st.interp p m)
  (ensures λm0 y m1 → st.interp (q y) m1)
```

Instantiating the Program Logic

- **Memory:** Map from abstract addresses to typed references

Instantiating the Program Logic

- **Memory:** Map from abstract addresses to typed references
- Standard SL connectives: $\star$, $-\!\!*$, $\wedge$, $\vee$, $\exists$, $\forall$

Instantiating the Program Logic

- **Memory:** Map from abstract addresses to typed references
- Standard SL connectives: $\star$, $-\!\!*$, $\wedge$, $\vee$, $\exists$, $\forall$
- Partial Commutative Monoid (PCM)-indexed pts_to assertion

Instantiating the Program Logic

- **Memory:** Map from abstract addresses to typed references
- Standard SL connectives: $\star$, $-\!\!*$, $\wedge$, $\vee$, $\exists$, $\forall$
- Partial Commutative Monoid (PCM)-indexed pts_to assertion
- Invariants

Invariants in Steel

```
let inv_name = nat
val (⇝) (i:inv_name) (p:slprop) : prop
let ival (p:slprop) = i:inv_name{i ⇝p}
```

Invariants in Steel

```
let inv_name = nat
val (⇝) (i:inv_name) (p:slprop) : prop
let ival (p:slprop) = i:inv_name{i ⇝p}

val new_invariant (p:slprop) : Steel (ival p) p emp
```

Using Invariants

Atomic commands

- Atomic actions
- Possibly composed with ghost computations

Using Invariants

Atomic commands

- Atomic actions
- Possibly composed with ghost computations
- New effect: SteelAtomic a (...) is_ghost p q

Using Invariants

Atomic commands

- Atomic actions
- Possibly composed with ghost computations
- New effect: SteelAtomic a (...) is_ghost p q

```
val with_invariant (i:ival p) (f:unit → SteelAtomic a g (p ⋆ q) (λ y → p ⋆ r y))
  : SteelAtomic a g q r
```

Using Invariants

Atomic commands

- Atomic actions
- Possibly composed with ghost computations
- New effect: SteelAtomic a (...) is_ghost p q

val with_invariant (i:ival p) (f:unit $\rightarrow$ SteelAtomic a $(i \uplus u)$ g (p $\star$ q) (λ y $\rightarrow$ p $\star$ r y))
  : SteelAtomic a u g q r

Stacking Abstractions in Steel

```
module Steel.Effect
module Steel.Effect.Atomic
```

Stacking Abstractions in Steel

```
module Steel.Effect
module Steel.Effect.Atomic

module Steel.Memory
module Steel.Actions
```

Stacking Abstractions in Steel

```
module Steel.Effect
module Steel.Effect.Atomic

module Steel.Memory
module Steel.Actions

module Steel.SpinLock
```

Stacking Abstractions in Steel

```
module Steel.Effect
module Steel.Effect.Atomic

module Steel.Memory
module Steel.Actions

module Steel.SpinLock

module Steel.ForkJoin
module Steel.Channels
```

Steel Example: Channel Types

```
val chan (p:prot) : Type
val sender #p (c:chan p) (cur:prot) : slprop
val receiver #p (c:chan p) (cur:prot) : slprop
```

Steel Example: Channel Types

```
val chan (p:prot) : Type
val sender #p (c:chan p) (cur:prot) : slprop
val receiver #p (c:chan p) (cur:prot) : slprop

val send #p (#cur:prot{more cur}) (c:chan p) (x:msg_t cur)
  : Steel unit (sender c cur) (λ _ → sender c (step cur x))
```

Steel Example: Channel Types

```
val chan (p:prot) : Type
val sender #p (c:chan p) (cur:prot) : slprop
val receiver #p (c:chan p) (cur:prot) : slprop

val send #p (#cur:prot{more cur}) (c:chan p) (x:msg_t cur)
  : Steel unit (sender c cur) (λ _ → sender c (step cur x))

val recv ... : Steel (msg_t cur) (receiver c cur) (λ x → receiver c (step cur x))
```

Steel Example: PingPong Protocol

```
let pingpong : prot =
  x ← Protocol.send int;
  y ← Protocol.recv (y:int{y > x});
  Protocol.done
```

Steel Example: PingPong Protocol

```
let pingpong : prot =
  x ← Protocol.send int;
  y ← Protocol.recv (y:int{y > x});
  Protocol.done

let client (c:chan pingpong) =
  send c 17;
  let y = recv c in
  assert (y > 17);
  return ()
```

Conclusion

Steel

- A shallow embedding of CSL in a dependently-typed language
- A PCM-based memory model
- Concurrency reasoning through dynamically allocated invariants
- 11 kLoC in F*, and a growing stack of verified libraries

Conclusion

Steel

- A shallow embedding of CSL in a dependently-typed language
- A PCM-based memory model
- Concurrency reasoning through dynamically allocated invariants
- 11 kLoC in F*, and a growing stack of verified libraries

Also in the paper

- Implicit Dynamic Frames
- Monotonicity and Preorders for References
- More libraries: Lock-coupling Lists, Counters with local state, ...

Conclusion

Steel

- A shallow embedding of CSL in a dependently-typed language
- A PCM-based memory model
- Concurrency reasoning through dynamically allocated invariants
- 11 kLoC in F*, and a growing stack of verified libraries

Also in the paper

- Implicit Dynamic Frames
- Monotonicity and Preorders for References
- More libraries: Lock-coupling Lists, Counters with local state, ...

fromherz@cmu.edu