SteelCore & Steel: An Extensible Concurrent Separation Logic for Effectful Dependent Programs

Nik Swamy

OPLSS 2021

Thanks to **Aymeric Fromherz**

Verifying Concurrent Programs

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

$$\textsc{Write}\quad \{r \mapsto \_\}\ r := v\ \{r \mapsto v\}$$

$$\textsc{Frame}\quad \frac{\{P\}\ c\ \{Q\}}{\{P * F\}\ c\ \{Q * F\}}$$

$$\textsc{Par}\quad \frac{\forall i. \{P_i\}\ c_i\ \{Q_i\}}{\{P_0 * P_1\}\ c_0 || c_1\ \{Q_0 * Q_1\}}$$

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

```
Definition swap : val := λ: "x" "y",
  let: "tmp" := !"x" in
  "x" <- !"y";;
  "y" <- "tmp".
```

```
Lemma swap_spec x y v1 v2 :
  {{{ x ↦ v1 ∗ y ↦ v2 }}} swap #x #y {{{ RET #(); x ↦ v2 ∗ y ↦ v1 }}}.
```

```
Proof.
  iIntros (Φ) "[Hx Hy] Post".
  unfold swap.
  wp_lam. wp_let.
  wp_load. wp_let.
  wp_load. wp_store.
  wp_store.
  iApply "Post".
  iSplitL "Hx".
  - iApply "Hx".
  - iApply "Hy".
Qed.
```

Verifying Concurrent Programs

How to get a CSL for a dependently-typed language? Through a shallow embedding?

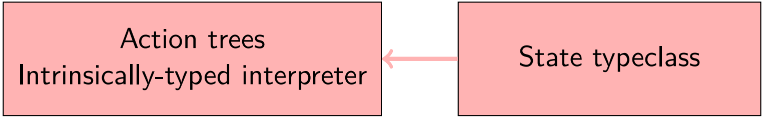
```
let swap (r0 r1:ref a)
  : ST unit
    (requires r0 ↦ v0 ∗ r1 ↦ v1)
    (ensures λ_ → r0 ↦ v1 ∗ r1 ↦ v0)
  = let v0 = !r0 in
    r0 := !r1;
    r1 := v0
```

Challenges

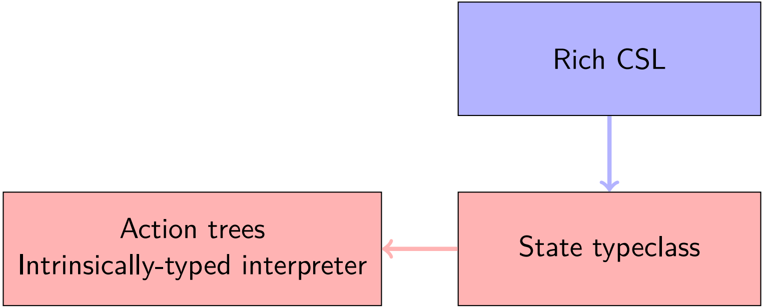
- How to reflect the effect of concurrency in the language?
- How to support partial correctness?
- How to enable dynamically allocated invariants?

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

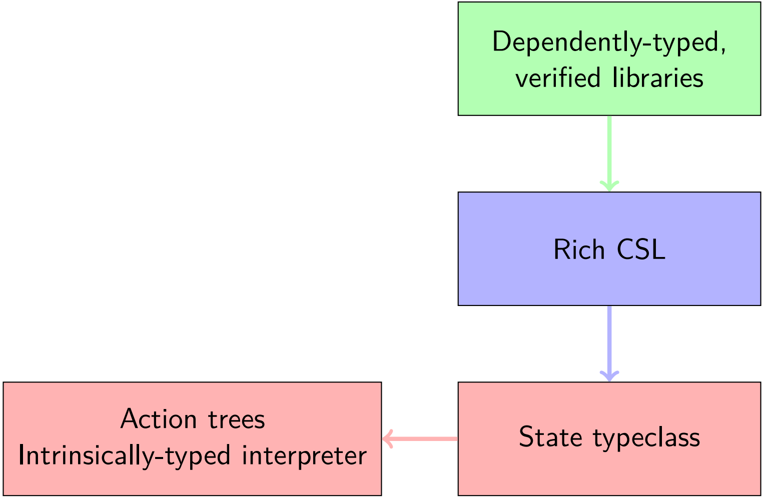
Steel: A Concurrent Separation Logic (CSL) for F*



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



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



Encoding Computations through Effectful Indexed Action Trees

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

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' → (x:a → Dv (ctree st b (post' x) post)) → ctree st a pre post
  | Par : ctree st unit p q → ctree st unit p' q' → ctree st a (q ‘st.star‘ q') post → ctree st a (
```

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 → SteelAtomic a (i ⊎ u) g (p ⋆ q) (λ y → p ⋆ 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
- 28 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
- 28 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
- 28 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