Steel: Scaling up verification in F*
Verified low-level programming is hard!

- Efficient memory reasoning is challenging
  1. Heap updates with aliasing
  2. Invariants on private state
  3. Interference among components (e.g. threads)
Verified low-level programming is hard!

• Efficient memory reasoning is challenging
  1. Heap updates with aliasing
  2. Invariants on private state
  3. Interference among components (e.g. threads)

• Verified low-level programming is now viable, but requires lots of effort
Verified low-level programming is hard!

- Efficient memory reasoning is challenging
  1. Heap updates with aliasing
  2. Invariants on private state
  3. Interference among components (e.g. threads)

- Verified low-level programming is now viable, but requires lots of effort
Verified low-level programming is hard!

- Efficient memory reasoning is challenging
  1. Heap updates with aliasing
  2. Invariants on private state
  3. Interference among components (e.g. threads)

- Verified low-level programming is now viable, but requires lots of effort

- $F^*$: Classical Hoare logic and select/update reasoning. How to scale?
Type-based Ownership to the Rescue?
Type-based Ownership to the Rescue?

The Rust example:
- Memory safety
- Data-race freedom

By virtue of typing!
Type-based Ownership to the Rescue?

The Rust example:
- Memory safety
- Data-race freedom

By virtue of typing!

What about verification?
- Rust programs aren’t proven correct
- Rust programs have unsafe blocks
Steel: Ownership and Verification via Resource Typing

Steel: A domain-specific language (DSL) shallowly embedded into F*

- Targets general-purpose concurrent, systems programming
Steel: Ownership and Verification via Resource Typing

Steel: A domain-specific language (DSL) shallowly embedded into F*
- Targets general-purpose concurrent, systems programming
- Always safe, with user-controlled verification
- Core theory proven sound within F*'s logic
Steel: Ownership and Verification via Resource Typing

Steel: A domain-specific language (DSL) shallowly embedded into F*

- Targets general-purpose concurrent, systems programming
- Always safe, with user-controlled verification
- Core theory proven sound within F**’s logic
- Extensible with new constructs, expressed as verified libraries
Steel: Ownership and Verification via Resource Typing

Steel: A domain-specific language (DSL) shallowly embedded into F*
- Targets general-purpose concurrent, systems programming
- Always safe, with user-controlled verification
- Core theory proven sound within F***’s logic
- Extensible with new constructs, expressed as verified libraries

Main ideas:
Steel: Ownership and Verification via Resource Typing

Steel: A domain-specific language (DSL) shallowly embedded into F*
- Targets general-purpose concurrent, systems programming
- Always safe, with user-controlled verification
- Core theory proven sound within F*'s logic
- Extensible with new constructs, expressed as verified libraries

Main ideas:
1. Separated resources and framing for interference control
Steel: Ownership and Verification via Resource Typing

Steel: A domain-specific language (DSL) shallowly embedded into F*
- Targets general-purpose concurrent, systems programming
- Always safe, with user-controlled verification
- Core theory proven sound within F*'s logic
- Extensible with new constructs, expressed as verified libraries

Main ideas:
1. Separated resources and framing for interference control
2. Permissions: Exclusive mutable or shared immutable access to resources
Steel: Ownership and Verification via Resource Typing

Steel: A domain-specific language (DSL) shallowly embedded into F*
- Targets general-purpose concurrent, systems programming
- Always safe, with user-controlled verification
- Core theory proven sound within F**’s logic
- Extensible with new constructs, expressed as verified libraries

Main ideas:
1. Separated resources and framing for interference control
2. Permissions: Exclusive mutable or shared immutable access to resources
3. Fork/join concurrency with locks
Steel: Current status

- Core memory model
- Resource separation
- Permissions
- Framing
- Concurrency
Steel: Current status

- Core memory model
- Resource separation
- Permissions
- Framing
- Concurrency

- Case studies: Singly and doubly-linked lists
Steel: Current status

- Core memory model
- Resource separation
- Permissions
- Framing
- Concurrency

- Case studies: Singly and doubly-linked lists

We can already write Steel programs, but...
Steel: Current status

- Core memory model
- Resource separation
- Permissions
- Framing
- Concurrency

- Case studies: Singly and doubly-linked lists

We can already write Steel programs, but...
Steel: Current status

- Core memory model
- Resource separation
- Permissions
- Framing
- Concurrency

- Case studies: Singly and doubly-linked lists

We can already write Steel programs, but...
Main idea 1: Resources
Main idea 1: Resources

1. Footprint
Main idea 1: Resources

1. Footprint

2. Invariant
Main idea 1: Resources

1. Footprint

2. Invariant
Main idea 1: Resources

1. Footprint

2. Invariant

emp: resource

ptr $p$

arr $b$
Main idea 1: Resources

1. Footprint

2. Invariant

\[
\text{ptr } p \quad \text{arr } b
\]

\[
\text{ptr } p \mapsto v \\
\text{arr } b \mapsto [> 0; \_]
\]
Main idea 1: Resources

1. Footprint
   emp: resource

2. Invariant
   ptr $p$
   arr $b$

3. View
   ptr $p \mapsto v$
   arr $b \mapsto [> 0; _]$
Resources : separation
Resources: separation

\[ R_1 \star R_2 \]
Resources: separation

$R_1 \star R_2$

$(\text{ptr } p_1 \rightarrow v_1) \star (\text{ptr } p_2 \rightarrow v_2)$
Resources: separation

\[ R_1 \ast R_2 \]

\( (\text{ptr } p_1 \mapsto v_1) \ast (\text{ptr } p_2 \mapsto v_2) \)

\[ \text{slist } x \mapsto \ell = \]
Resources: separation

$$R_1 \star R_2$$

$$(\text{ptr } p_1 \mapsto v_1) \star (\text{ptr } p_2 \mapsto v_2)$$

\[
s\text{list } x \mapsto \ell = \text{match } \ell \text{ with} \\
\quad | \quad [] \rightarrow \\
\quad | \quad \text{hd} :: \text{tl} \rightarrow
\]
Resources : separation

\[ R_1 \star R_2 \]

\((\text{ptr } p_1 \mapsto v_1) \star (\text{ptr } p_2 \mapsto v_2)\)

\[
\text{slist } x \mapsto l = \text{match } l \text{ with }
| [] \to \text{emp}
| \text{hd} :: \text{tl} \to
\]
Resources: separation

\[ R_1 \star R_2 \]

\[(\text{ptr } p_1 \leftrightarrow v_1) \star (\text{ptr } p_2 \leftrightarrow v_2)\]

\[
\text{slist } x \leftrightarrow \ell = \text{match } \ell \text{ with}
\]

\[
| [] \rightarrow \text{emp}
\]

\[
| \text{hd} :: \text{tl} \rightarrow (\text{ptr } x \leftrightarrow \text{hd}) \star (\text{slist } \text{hd}. \text{next} \leftrightarrow \text{tl})
\]
Resources: typing
Resources: typing

Computations on resources:

\[ \text{RST (} \alpha : \text{Type) } \]
Resources: typing

Computations on resources:

\[
\text{RST } (\alpha : \text{Type})
\]

(expects resource)
Resources: typing

Computations on resources:

RST (α: Type)
(expects resource)
(provides (α → resource))
Resources: typing

Allocating a pointer:

Computations on resources:

\[
\text{RST} \ (\alpha: \text{Type}) \\
\quad (\text{expects resource}) \\
\quad (\text{provides } (\alpha \rightarrow \text{resource}))
\]
Resources: typing

Allocating a pointer:

val alloc : (v : α) → RST (pointer α)

Computations on resources:

RST (α : Type)
(expects resource)
(provides (α → resource))
Resources: typing

Allocating a pointer:

\[
\text{val alloc : (v: } \alpha) \rightarrow \\
\text{RST (pointer } \alpha) \\
\text{(expects emp)}
\]

Computations on resources:

\[
\text{RST (} \alpha: \text{ Type)} \\
\text{(expects resource)} \\
\text{(provides (} \alpha \rightarrow \text{ resource))}
\]
Resources: typing

Allocating a pointer:

```plaintext
val alloc : (v: α) →
  RST (pointer α)
    (expects emp)
    (provides (λ p → (ptr p ← v)))
```

Computations on resources:

```plaintext
RST (α: Type)
  (expects resource)
  (provides (α → resource))
```
Resources: typing

Allocating a pointer:

```plaintext
val alloc : (ν: α) →
RST (pointer α)
  (expects emp)
  (provides (λ p → (ptr p ← ν)))
```

Updating a pointer:

Computations on resources:

```plaintext
RST (α: Type)
  (expects resource)
  (provides (α → resource))
```
Resources: typing

**Allocating a pointer:**

\[\text{val alloc : } (v : \alpha) \to RST \text{ (pointer } \alpha) \]
\[\text{ (expects emp)} \]
\[\text{ (provides } (\lambda p \to (\text{ptr } p \leftrightarrow v)))\]

**Updating a pointer:**

\[\text{val } (\vdash) : (p : \text{ pointer } \alpha) \to (v : \alpha) \to RST \text{ unit}\]
Resources: typing

**Allocating a pointer:**

```plaintext
val alloc : (v: α) →
  RST (pointer α)
    (expects emp)
    (provides (λ p → (ptr p ↦ v)))
```

**Updating a pointer:**

```plaintext
val (=:): (p: pointer α) → (v: α) →
  RST unit
    (expects (ptr p))
```
Resources: typing

Computations on resources:

RST ($\alpha$: Type)
(expects resource)
(provides ($\alpha \rightarrow$ resource))

Allocating a pointer:

```haskell
val alloc : (v: $\alpha$) →
RST (pointer $\alpha$)
(expects emp)
(provides ($\lambda\ p \rightarrow (ptr\ p \mapsto v)$))
```

Updating a pointer:

```haskell
val (\_): (p: pointer $\alpha$) → (v: $\alpha$) →
RST unit
(expects (ptr p))
(provides ($\lambda\_ \rightarrow (ptr\ p \mapsto v)$))
```
Main idea 1: Resources

1. Footprint
   emp: resource

2. Invariant
   ptr \( p \)  
   arr \( b \)

3. View
   ptr \( p \) \( \mapsto \) \( v \)  
   arr \( b \) \( \mapsto \) \( \geq 0; \_ \)
Resources: typing

Computations on resources:

**RST** ($\alpha$: **Type**)  
(expects resource)  
(provides ($\alpha \to \text{resource}$))

Allocating a pointer:

```
val alloc : ($v$: $\alpha$) $\rightarrow$
  RST (pointer $\alpha$)
  (expects emp)
  (provides ($\lambda p \rightarrow (\text{ptr } p \mapsto v)$))
```

Updating a pointer:

```
val (=:): ($p$: pointer $\alpha$) $\rightarrow$ ($v$: $\alpha$) $\rightarrow$
  RST unit
  (expects (ptr $p$))
  (provides ($\lambda _\rightarrow (\text{ptr } p \mapsto v)$))
```
Resources: view specification
Resources: view specification

Separation-logic style specification:
Resources: view specification

Separation-logic style specification:

```haskell
val inc : (p : pointer int) → (v : ghost int) →
    RST unit
```
Resources: view specification

Separation-logic style specification:

\( \text{val inc : (p: pointer int) \rightarrow (v: ghost int) \rightarrow} \)

\( \text{RST unit} \)

\( \text{(expects (ptr } p \rightarrow v)) \)

\( \text{(provides (λ_ \rightarrow (ptr } p \rightarrow v + 1))) \)
Resources: view specification

Separation-logic style specification:

\[
\text{val inc : } (p: \text{ pointer int}) \rightarrow (v: \text{ ghost int}) \rightarrow \\
\text{RST unit} \\
\quad (\text{expects } (\text{ptr } p \rightarrow v)) \\
\quad (\text{provides } (\lambda_\_ \rightarrow (\text{ptr } p \rightarrow v + 1)))
\]
Resources: view specification

Separation-logic style specification:

```haskell
val inc : (p: pointer int) → (v: ghost int) →
   RST unit
   (expects (ptr p ↦ v))
   (provides (λ_ → (ptr p ↦ v + 1)))
```

More stateful specification:

```haskell
val inc : (p: pointer int) →
   RST unit
```
Resources: view specification

Separation-logic style specification:

```plaintext
val inc : (p: pointer int) → (v: ghost int) →
  RST unit
  (expects (ptr p → v))
  (provides (λ_→(ptr p → v + 1)))
```

More stateful specification:

```plaintext
val inc : (p: pointer int) →
  RST unit
  (expects (ptr p))
```
Resources: view specification

Separation-logic style specification:

```haskell
val inc : (p: pointer int) → (v: ghost int) →
  RST unit
  ( expects (ptr p → v))
  ( provides (λ_ → (ptr p → v + 1)))
```

More stateful specification:

```haskell
val inc : (p: pointer int) →
  RST unit
  ( expects (ptr p))
  ( provides (λ_ → ptr p))
```
Resources: view specification

Separation-logic style specification:

\[
\text{val inc : (} p : \text{ pointer int) } \rightarrow \text{(} v : \text{ ghost int) } \rightarrow \\
\text{RST unit} \\
\quad (\text{expects (} \text{ptr } p \rightarrow v)) \\
\quad (\text{provides (} \lambda _\_ \rightarrow (\text{ptr } p \rightarrow v + 1)))
\]

More stateful specification:

\[
\text{val inc : (} p : \text{ pointer int) } \rightarrow \\
\text{RST unit} \\
\quad (\text{expects (} \text{ptr } p)) \\
\quad (\text{provides (} \lambda _\_ \rightarrow \text{ptr } p)) \\
\quad (\text{ensures (} \lambda \text{ old_new } \rightarrow 
\]
Resources: view specification

Separation-logic style specification:

\[
\text{val inc : (p: pointer int) \to (v: ghost int) \to RST unit}
\]
\[
\quad (\text{expects (ptr p \to v)})
\]
\[
\quad (\text{provides (\lambda \_ \to (ptr p \to v + 1)))}
\]

More stateful specification:

\[
\text{val inc : (p: pointer int) \to RST unit}
\]
\[
\quad (\text{expects (ptr p)})
\]
\[
\quad (\text{provides (\lambda \_ \to ptr p)})
\]
\[
\quad (\text{ensures (\lambda old\_new \to new (ptr p) = old (ptr p) + 1)}
\]
A frame rule for Steel
Resources: view specification

Separation-logic style specification:

\[
\text{val inc} : (p: \text{pointer int}) \rightarrow (v: \text{ghost int}) \rightarrow \text{RST \ unit}
\]
\[
\quad \text{(expects (ptr p \rightarrow v))}
\]
\[
\quad \text{(provides (λ_ \rightarrow (ptr p \rightarrow v + 1)))}
\]

More stateful specification:

\[
\text{val inc} : (p: \text{pointer int}) \rightarrow \text{RST \ unit}
\]
\[
\quad \text{(expects (ptr p))}
\]
\[
\quad \text{(provides (λ_ \rightarrow ptr p))}
\]
\[
\quad \text{(ensures (λ old_new \rightarrow new (ptr p) = old (ptr p) + 1))}
\]
Resources: view specification

Separation-logic style specification:

```haskell
val inc : (p: pointer int) → (v: ghost int) →
  RST unit
  (expects (ptr p ↦ v))
  (provides (λ → (ptr p ↦ v + 1)))
```

More stateful specification:

```haskell
val inc : (p: pointer int) →
  RST unit
  (expects (ptr p))
  (provides (λ → ptr p))
  (ensures (λ old_new →
             new (ptr p) = old (ptr p) + 1))
```
A frame rule for Steel
A frame rule for Steel

Classic frame rule:

\[
\{Q\} f \{R\} \\
\{P \star Q\} f \{P \star R\}
\]
A frame rule for Steel

Classic frame rule:  
\[
\frac{\{Q\} \ f \ \{R\}}{\{P \ast Q\} \ f \ \{P \ast R\}}
\]

In Steel:
A frame rule for Steel

**Classic frame rule:**

\[
\frac{\{Q\} \ f \ \{R\}}{\{P \star Q\} \ f \ \{P \star R\}}
\]

**In Steel:**

val frame :

(P \star Q: resource) \to
A frame rule for Steel

Classic frame rule:

\[
\frac{\{Q\} f \{R\}}{\{P \star Q\} f \{P \star R\}}
\]

In Steel:

```java
val frame :
     (P \star Q: resource) →
     (P \star R: resource) →
```
A frame rule for Steel

Classic frame rule:

\[
\begin{align*}
\{Q\} & f \{R\} \\
\{P \ast Q\} & f \{P \ast R\}
\end{align*}
\]

In Steel:

```scala
val frame : 
(P \ast Q: resource) → 
(P \ast R: resource) → 
(f: unit → RST α (expects Q) (provides R)) →
```
A frame rule for Steel

Classic frame rule:

\[
\frac{\{Q\} \ f \ \{R\}}{\{P \star Q\} \ f \ \{P \star R\}}
\]

In Steel:

```plaintext
val frame :
  (P \star Q: resource) →
  (P \star R: resource) →
  (f: unit → RST α (expects Q) (provides R)) →
  RST α
  (expects (P \star Q))
  (provides (P \star R))
```
Challenge: efficient $F^*$ embedding
Challenge: efficient $F^*$ embedding

Associative/Commutative rewriting:
Challenge: efficient F* embedding

Associative/Commutative rewriting:

\[ P \ast Q \ast R \]

\[
\begin{align*}
\text{val } f &: \text{ unit } \rightarrow \text{ RST } \text{ unit} \\
& \quad (\text{expects } Q) \\
& \quad (\text{provides } Q')
\end{align*}
\]
Challenge: efficient F* embedding

Associative/Commutative rewriting:

\[ P \ast Q \ast R \]

AC rewriting

\[ Q \ast (P \ast R) \]

val \( f : \) unit \( \rightarrow \) RST unit
(expects Q)
(provides Q')
Challenge: efficient F* embedding

Associative/Commutative rewriting:

\[
\begin{align*}
P \ast Q \ast R \\
Q \ast (P \ast R) \\
Q \quad P \ast R
\end{align*}
\]

AC rewriting

\[
\text{val } f : \text{unit } \rightarrow \text{RST unit (expects Q) (provides Q')}
\]
Challenge: efficient F* embedding

Associative/Commutative rewriting:

\[ P \ast Q \ast R \]

AC rewriting

\[ Q \ast (P \ast R) \]

framing

\[ Q \quad P \ast R \]

\[ f \]

\[ Q' \quad P \ast R \]

\[ \text{val } f : \text{unit } \rightarrow \text{RST unit} \]

(expects Q)

(provides Q')
Challenge: efficient F* embedding

Associative/Commutative rewriting:

\[
P \star Q \star R
\]

AC rewriting

\[
Q \star (P \star R)
\]

framing

\[
Q
P \star R
\]

\[
f
Q'
P \star R
\]

val f : unit → RST unit
(expects Q)
(provides Q')

\[
Q' \star (P \star R)
\]

framing
Challenge: efficient F* embedding

Associative/Commutative rewriting:

\[
P \ast Q \ast R
\]

AC rewriting

\[
Q \ast (P \ast R)
\]

framing

\[
Q
\]

\[
P \ast R
\]

\[
Q'
\]

\[
P \ast R
\]

\[
Q' \ast (P \ast R)
\]

\[
P \ast Q' \ast R
\]

val f : unit → RST unit
(expects Q)
(provides Q')

framing

AC rewriting
Challenge: efficient F* embedding

Associative/Commutative rewriting:

\[ P \ast Q \ast R \]

\[ Q \ast (P \ast R) \]

AC rewriting

\[ Q \]

\[ P \ast R \]

\[ f \]

\[ Q' \]

\[ P \ast R \]

framing

Goong higher order:

\[ \text{val } f : \text{unit } \rightarrow \text{RST unit} \]

(expects \( Q \))

(provides \( Q' \))

\[ Q' \ast (P \ast R) \]

\[ P \ast Q' \ast R \]

framing

AC rewriting
Challenge: efficient $F^*$ embedding

**Associative/Commutative rewriting:**

\[ P \ast Q \ast R \]

\[ Q \ast (P \ast R) \]

\[ Q \]

\[ P \ast R \]

\[ f \]

\[ Q' \]

\[ P \ast R \]

\[ Q' \ast (P \ast R) \]

\[ P \ast Q' \ast R \]

**Going higher order:**

- Current heap model: only first-order logic

\[ \text{val } f : \text{unit } \rightarrow \text{RST unit} \]

(expects $Q$)

(provides $Q'$)
Challenge: efficient F* embedding

Associative/Commutative rewriting:

\[ P \ast Q \ast R \]

Going higher order:

- Current heap model: only first-order logic
- resource \( \approx \) heap \( \rightarrow \) prop
Challenge: efficient F* embedding

Associative/Commutative rewriting:

\[ P \times Q \times R \]

AC rewriting

\[ Q \times (P \times R) \]

framing

\[ Q \]

\[ P \times R \]

\[ f \]

\[ Q' \]

\[ P \times R \]

\[ f \]

val \( f : \text{unit} \rightarrow \text{RST unit} \)

(expects \( Q \))

(provides \( Q' \))

\[ Q' \times (P \times R) \]

framing

AC rewriting

\[ P \times Q' \times R \]

Going higher order:

- Current heap model: only first-order logic
- resource \( \approx \) heap \( \rightarrow \) prop
- \( \ast : \text{resource} \rightarrow \text{resource} \rightarrow \text{resource} \) is higher-order
Mixing tactics and SMT
Mixing tactics and SMT

Verification conditions
Mixing tactics and SMT

- Verification conditions
- SMT (Z3)
Mixing tactics and SMT

- Verification conditions
- Meta-F* tactics (since 2017)
- SMT (Z3)
Mixing tactics and SMT

- Verification conditions
  - separation logic
  - non-heap VCs
- Meta-F* tactics (since 2017)
- SMT (Z3)
Main idea 2: Permissions over shared resources
Main idea 2: Permissions over shared resources

Updating requires exclusive ownership:
Main idea 2: Permissions over shared resources

Updating requires exclusive ownership:

\[
\text{val} \ (:=) : (p: \text{pointer } \alpha) \rightarrow (v: \alpha) \rightarrow \text{RST} (\text{pointer } \alpha)
\]

\[
\begin{align*}
\text{(expects} & (\text{ptr } p)) \\
\text{(provides} & (\lambda p \rightarrow (\text{ptr } p \mapsto v)))
\end{align*}
\]
Main idea 2: Permissions over shared resources

Updating requires exclusive ownership:

```
val (::) : (p: pointer α) → (v: α) →
RST (pointer α)
  (expects (ptr p))
  (provides (λ p → (ptr p ⪯ v)))
```

What if we only want to read p?
Main idea 2: Permissions over shared resources

Updating requires exclusive ownership:

\[ \text{val} \ (\text{:=}) : (p : \text{pointer } \alpha) \to (v : \alpha) \rightarrow \]
\[ \text{RST} (\text{pointer } \alpha) \]
\[ (\text{expects } (\text{ptr } p)) \]
\[ (\text{provides } (\lambda p \rightarrow (\text{ptr } p \mapsto v))) \]

What if we only want to read \( p \)?

\[ \text{read\_only } p; \]
\[ ... \]
\[ p := 0; \]
\[ ... \]
Sharing and read-only resources

Read-only resource:
RO \textit{R}

Read-write resource:
RW \textit{R}
Sharing and read-only resources

Read-only resource:

**RO R**

Read-write resource:

**RW R**

**Pointer dereference:**
Sharing and read-only resources

- Read-only resource: \( RO \ R \)
- Read-write resource: \( RW \ R \)

**Pointer dereference:**

\[
\text{val}(!) : (p: \text{pointer } \alpha) \rightarrow \text{RST } \alpha
\]
Sharing and read-only resources

Read-only resource:

RO $R$

Read-write resource:

RW $R$

Pointer dereference:

\[
\text{val} \; (!) \; : \; (p: \text{pointer } \alpha) \rightarrow \\
\text{RST } \alpha \\
(\text{expects} \; (\text{RO} \; (\text{ptr} \; p)))
\]
Sharing and read-only resources

Read-only resource:

\[ \text{RO } R \]

Read-write resource:

\[ \text{RW } R \]

Pointer dereference:

\[ \text{val} (!) : (p: \text{pointer } \alpha) \rightarrow \]
\[ \text{RST } \alpha \]
\[ (\text{expects } (\text{RO } (\text{ptr } p))) \]
\[ (\text{provides } (\lambda v \rightarrow \text{RO } (\text{ptr } p \mapsto v))) \]
Sharing and read-only resources

Read-only resource:

RO $R$

Read-write resource:

RW $R$

Pointer dereference:

\[
\text{val} (!) : (p: \text{pointer } \alpha) \to \\
\text{RST } \alpha \\
\quad (\text{expects } (\text{RO } (\text{ptr } p))) \\
\quad (\text{provides } (\lambda v \to \text{RO } (\text{ptr } p \leftrightarrow v)))
\]

Immutable sharing:
Sharing and read-only resources

Read-only resource:
RO R

Read-write resource:
RW R

Pointer dereference:

\[
\text{val} \ (\_\_\_) \ : \ (p: \text{pointer } \alpha) \rightarrow \text{RST } \alpha \\
\text{ (expects } (\text{RO } (\text{ptr } p))) \\
\text{ (provides } (\lambda \nu \rightarrow \text{RO } (\text{ptr } p \mapsto \nu)))
\]

Immutable sharing:

\[
\text{val} \ \text{share} \ : \ (p: \text{pointer } \alpha) \rightarrow \text{RST } (\text{pointer } \alpha)
\]
Sharing and read-only resources

Read-only resource:

RO \( R \)

Read-write resource:

RW \( R \)

Pointer dereference:

\[
\text{val} \; (\downarrow) : (p: \text{pointer } \alpha) \to \\
\text{RST} \; \alpha \\
\quad (\text{expects } (\text{RO } (\text{ptr } p))) \\
\quad (\text{provides } (\lambda v \to \text{RO } (\text{ptr } p \mapsto v)))
\]

Immutable sharing:

\[
\text{val} \; \text{share} : (p: \text{pointer } \alpha) \to \\
\text{RST} \; (\text{pointer } \alpha) \\
\quad (\text{expects } (\text{RW } (\text{ptr } p)))
\]
Sharing and read-only resources

Read-only resource:
RO $R$

Read-write resource:
RW $R$

Pointer dereference:

```haskell
val (!) : (p: pointer $\alpha$) →
RST $\alpha$
(expects (RO (ptr p)))
(provides (\lambda v → RO (ptr p ↦ v)))
```

Immutable sharing:

```haskell
val share : (p: pointer $\alpha$) →
RST (pointer $\alpha$)
(expects (RW (ptr p)))
(provides (\lambda p' → RO (ptr p) * RO (ptr p')))
```
Managing fractional permissions
Managing fractional permissions

Permissions are fractions:

\[ RO \ R = R_{Perm=f}, \quad 0 < f < 1 \]
\[ RW \ R = R_{Perm=1} \]
Managing fractional permissions

Permissions are fractions:

\[ \text{RO } R = R_{\text{Perm}=f}, \quad 0 < f < 1 \]

\[ \text{RW } R = R_{\text{Perm}=1} \]

Gathering back permissions:
Managing fractional permissions

Permissions are fractions:

\[ RO\, R = R_{\text{Perm}=f}, \; 0 < f < 1 \]
\[ RW\, R = R_{\text{Perm}=1} \]

Gathering back permissions:

\[ \text{val gather} : (p: \text{pointer } \alpha) \rightarrow (p': \text{pointer } \alpha) \rightarrow \text{RST unit} \]
Managing fractional permissions

Permissions are fractions:

\[
\begin{align*}
\text{RO } R &= R_{\text{Perm}=f}, \ 0 < f < 1 \\
\text{RW } R &= R_{\text{Perm}=1}
\end{align*}
\]

Gathering back permissions:

\[
\text{val gather : (p: pointer } \alpha \text{) } \rightarrow (p': \text{ pointer } \alpha) \rightarrow \\
\text{ RST unit} \\
(\text{expects } ((\text{ptr } p)_{\text{Perm}=f} \ast (\text{ptr } p')_{\text{Perm}=f'}))
\]
Managing fractional permissions

Permissions are fractions:

\[
\begin{align*}
RO R &= R_{\text{Perm}=f}, \quad 0 < f < 1 \\
RW R &= R_{\text{Perm}=1}
\end{align*}
\]

Gathering back permissions:

\[
\begin{align*}
\text{val gather} : (p : \text{pointer } \alpha) &\to (p' : \text{pointer } \alpha) \\
&\to \text{RST unit} \\
&(\text{expects } ((\text{ptr } p)_{\text{Perm}=f} \ast (\text{ptr } p')_{\text{Perm}=f'})) \\
&(\text{provides } (\lambda \_ \rightarrow (\text{ptr } p)_{\text{Perm}=f+f'}))
\end{align*}
\]
Managing fractional permissions

Permissions are fractions:

\[
\begin{align*}
RO R &= R_{Perm=f}, \ 0 < f < 1 \\
RW R &= R_{Perm=1}
\end{align*}
\]

Gathering back permissions:

\[
\text{val} \ \text{gather} : (p : \text{pointer } \alpha) \rightarrow (p' : \text{pointer } \alpha) \rightarrow \text{RST} \ \text{unit}
\]

(expects \ ((\text{ptr } p)_{Perm=f} \ast (\text{ptr } p')_{Perm=f'}))
(provides \ (\lambda_\rightarrow (\text{ptr } p)_{Perm=f+f'}))
(requires \ (\text{gatherable } p \ p'))
Managing fractional permissions

Permissions are fractions:

\[ RO R = R_{\text{Perm}=f}, \quad 0 < f < 1 \]

\[ RW R = R_{\text{Perm}=1} \]

Gathering back permissions:

```ocaml
val gather : (p : pointer α) \to (p' : pointer α) \to
RST unit
(expects ((\text{ptr } p)_{\text{Perm}=f} \ast (\text{ptr } p')_{\text{Perm}=f'}))
(provides (λ_ \to (\text{ptr } p)_{\text{Perm}=f+f'}))
(requires (gatherable p p'))
```

- Within TCB and memory model
Managing fractional permissions

Permissions are fractions:

\[
\begin{align*}
\text{RO } R &= R_{\text{Perm}=f}, \ 0 < f < 1 \\
\text{RW } R &= R_{\text{Perm}=1}
\end{align*}
\]

Gathering back permissions:

\[
\text{val gather : } (p: \text{ pointer } \alpha) \to (p': \text{ pointer } \alpha) \to
\begin{align*}
\text{ RST unit} \\
\text{(expects } ((\text{ptr } p)_{\text{Perm}=f} \ast (\text{ptr } p')_{\text{Perm}=f'})) \\
\text{(provides } (\lambda_\_ \to (\text{ptr } p)_{\text{Perm}=f+f'})) \\
\text{(requires } \text{gatherable } p \ p')
\end{align*}
\]

- Within TCB and memory model
- Statically checked with SMT
Managing fractional permissions

Permissions are fractions:

\[ RO \, R = R_{\text{Perm}=f}, \, 0 < f < 1 \]
\[ RW \, R = R_{\text{Perm}=1} \]

Gathering back permissions:

```plaintext
val gather : (p: pointer α) → (p': pointer α) →
RST unit
(expects ((ptr p)_{\text{Perm}=f} * (ptr p')_{\text{Perm}=f'}))
(provides (λ_ → (ptr p)_{\text{Perm}=f+f'}))
(requires (gatherable p p'))
```

- Within TCB and memory model
- Statically checked with SMT
- Users can define scoped sharing, etc.
Main idea 3:
Concurrency in Steel

Currently in scope:
- Data-race freedom
- Sequential consistency
- Scoped fork-join model (par combinator)
- Mutable memory shared through locks
Managing fractional permissions

Permissions are fractions:
\[
\begin{align*}
\text{RO } R &= R_{\text{Perm}=f}, \quad 0 < f < 1 \\
\text{RW } R &= R_{\text{Perm}=1}
\end{align*}
\]

Gathering back permissions:
\[
\text{val gather : (p: pointer } \alpha) \to (p': \text{ pointer } \alpha) \to
\begin{align*}
\text{unit} \\
\text{expects } ((\text{ptr } p)_{\text{Perm}=f} \times (\text{ptr } p')_{\text{Perm}=f'}) \\
\text{provides } (\lambda_\_ \to (\text{ptr } p)_{\text{Perm}=f+f'}) \\
\text{requires } (\text{gatherable } p \ p')
\end{align*}
\]

- Within TCB and memory model
- Statically checked with SMT
- Users can define scoped sharing, etc.
Main idea 3: Concurrency in Steel

Currently in scope:
- Data-race freedom
- Sequential consistency
- Scoped fork-join model (par combinator)
- Mutable memory shared through locks
The “par” combinator
The “par” combinator

Concurrent separation logic:

\[ \{P_1\} f \{Q_1\} \quad \{P_2\} g \{Q_2\} \]
The “par” combinator

Concurrent separation logic:

\[
\frac{\{P_1\} f \{Q_1\} \quad \{P_2\} g \{Q_2\}}{\{P_1 \ast P_2\} f \parallel g \{Q_1 \ast Q_2\}}
\]
The "par" combinator

Concurrent separation logic:

\[
\frac{\{P_1\} f \{Q_1\} \quad \{P_2\} g \{Q_2\}}{\{P_1 \times P_2\} \mathbf{f} \parallel \mathbf{g} \{Q_1 \times Q_2\}}
\]

Steel:

```plaintext
val par:
  (f: unit → RST α P_1 Q_1) →
  (g: unit → RST β P_2 Q_2) →
  RST (α * β)
  (expects P_1 * P_2)
  (provides Q_1 * Q_2)
```
Shared mutable access using locks

Concurrent separation logic:
Shared mutable access using locks

Concurrent separation logic:

\{ P \} \quad \text{new\_lock: lock } P \quad \{ \text{emp} \}
Shared mutable access using locks

Concurrent separation logic:

\[
\begin{align*}
&\{ P \} \quad \text{new\_lock: lock } P \quad \{ \text{emp} \} \\
&\{ \text{emp} \} \quad \text{acquire (} \ell \text{: lock } P \text{)} \quad \{ P \} \\
&\{ P \} \quad \text{release (} \ell \text{: lock } P \text{)} \quad \{ \text{emp} \}
\end{align*}
\]
Shared mutable access using locks

Concurrent separation logic:

Stable invariant

\[
\{P\} \quad \text{new\_lock: lock } P \quad \{\text{emp}\} \\
\{\text{emp}\} \quad \text{acquire } (\ell: \text{lock } P) \quad \{P\} \\
\{P\} \quad \text{release } (\ell: \text{lock } P) \quad \{\text{emp}\}
\]
Shared mutable access using locks

Concurrent separation logic:

Stable invariant

\[
\begin{align*}
\{ P \} & \quad \text{new\_lock: lock \( P \)} & \quad \{ \text{emp} \} \\
\{ \text{emp} \} & \quad \text{acquire (\( \ell \): lock \( P \))} & \quad \{ P \} \\
\{ P \} & \quad \text{release (\( \ell \): lock \( P \))} & \quad \{ \text{emp} \}
\end{align*}
\]
Shared mutable access using locks

Concurrent separation logic:

Stable invariant

\[
\begin{align*}
\{ P \} & \quad \text{new\_lock: lock } P \quad \{ \text{emp} \} \\
\{ \text{emp} \} & \quad \text{acquire } (\ell: \text{lock } P) \quad \{ P \} \\
\{ P \} & \quad \text{release } (\ell: \text{lock } P) \quad \{ \text{emp} \}
\end{align*}
\]

Steel:

\[
\text{val acquire } (\ell:\text{lock } R) \to \\
\text{RST unit} \\
(\text{expects emp}) \\
(\text{provides } R)
\]
Shared mutable access using locks

Concurrent separation logic:

Stable invariant

\[
\begin{align*}
\{P\} & \quad \text{new\_lock: lock } P \quad \{\text{emp}\} \\
\{\text{emp}\} & \quad \text{acquire (} \ell: \text{ lock } P \} \quad \{P\} \\
\{P\} & \quad \text{release (} \ell: \text{ lock } P \} \quad \{\text{emp}\}
\end{align*}
\]

Steel:

\[
\begin{align*}
\text{val acquire (} \ell: \text{lock } R \to \\
\text{RST unit} \\
\text{(expects emp)} \\
\text{(provides } R) \end{align*}
\]

- Lock predicates checked statically
- Lock availability checked at runtime
Soundness of the concurrency model
(Work in progress)
Soundness of the concurrency model (Work in progress)

- Soundness for any **non-blocking** interleaving
- Machine checked in F* !
Soundness of the concurrency model (Work in progress)

- Soundness for any non-blocking interleaving
- Machine checked in F*!
Case study: Linked lists specification
Case study: Linked lists specification

Steel:

```ocaml
let rec slist x (ℓ:ghost (list cell)) : resource =
  match ℓ with
  | []   -> emp
  | hd::tl -> (x ↦ hd) * (slist hd.next tl)
```
Case study: Linked lists specification

Steel:

```ocaml
let rec slist x (ℓ:ghost (list cell)) : resource =
  match ℓ with
  | [] -> emp
  | hd::tl -> (x -> hd) * (slist hd.next tl)

val cons (p: pointer) v x ℓ : RST unit
(expects ((p → v) * (slist x ℓ)))
(provides (slist p (v :: ℓ)))
```
Case study: Linked lists specification

Steel:

```ocaml
let rec slist x (ℓ:ghost (list cell)) : resource =
  match ℓ with
  | [] -> emp
  | hd::tl -> (x ↦ hd) * (slist hd.next tl)

val cons (p: pointer) v x ℓ : RST unit
(expects ((p ↦ v) * (slist x ℓ)))
(provides (slist p (v :: ℓ)))

val map f p ℓ : RST unit
(expects (slist p ℓ))
(provides (slist p (map_cell f ℓ)))
```
Case study: Linked lists specification

Steel:

```ocaml
let rec slist x (ℓ:ghost (list cell)) : resource =
  match ℓ with
  | [] -> emp
  | hd::tl -> (x -> hd) * (slist hd.next tl)

val cons (p: pointer) v x ℓ : RST unit
  (expects ((p -> v) * (slist x ℓ)))
  (provides (slist p (v :: ℓ)))

val map f p ℓ : RST unit
  (expects (slist p ℓ))
  (provides (slist p (map_cell f ℓ)))
```

Low*:

```ocaml
let well_formed x ℓ = ... (10 lines)
let footprint x ℓ = ... (8 lines)

val cons p v x ℓ : ST unit
  (requires
    well_formed x ℓ &&
    disjoint (loc p) (footprint x ℓ) &&
    live p &&
    get p = v)
  (ensures
    well_formed p (v :: ℓ) &&
    modifies (footprint p (v :: ℓ)))
```

cons, head, tail, map specification
• Steel: 30 LOC
• Low*: 100 LOC
Case study: Linked lists implementation

Steel:

let rec map f x ℓ =
...
frame
((x ↦ hd ℓ) * slist (hd ℓ).next (tl ℓ))
(λ _ → (x ↦ f (hd ℓ)) * slist (hd ℓ).next (tl ℓ))
(update_cell f x);
frame
((x ↦ f (hd ℓ)) * slist (hd ℓ).next (tl ℓ))
(λ _ → (x ↦ f (hd ℓ)) * slist (hd ℓ).next (map_cell f (tl ℓ)))
(map f (hd ℓ).next (tl ℓ))
Case study: Linked lists implementation

Steel:

```
let rec map f x ℓ =
    ...
    frame
        ((x ↦ hd ℓ) * slist (hd ℓ).next (tl ℓ))
        (λ _ → (x ↦ f (hd ℓ)) * slist (hd ℓ).next (tl ℓ))
        (update_cell f x);
    frame
        ((x ↦ f (hd ℓ)) * slist (hd ℓ).next (tl ℓ))
        (λ _ → (x ↦ f (hd ℓ)) * slist (hd ℓ).next (map_cell f (tl ℓ)))
        (map f (hd ℓ).next (tl ℓ))
```

Ideally:

```
let rec map f x ℓ =
    ...
    update_cell f x;
    map f (hd ℓ).next (tl ℓ)
```

Work in progress: Better frame inference
Case study: Doubly-linked lists

- Doubly-linked lists in Steel: 400 LoCs
- In Low*: 4000 LoCs!
Case study: Doubly-linked lists

- Doubly-linked lists in Steel: 400 LoCs
- In Low*: 4000 LoCs!

- Doubly-linked lists are not expressible in Rust without unsafe blocks due to aliasing restrictions
- Steel is expressive enough to capture complex aliasing patterns
Future work

- Improve usability of the framework (3 – 6 months)
  - Frontend syntax
  - More fine-tuning of SMT queries
  - Additional libraries
  - Complete interoperation with Low*
Future work

- Improve usability of the framework (3 – 6 months)
  - Frontend syntax
  - More fine-tuning of SMT queries
  - Additional libraries
  - Complete interoperation with Low*

- Concurrency (1 – 2 months)
  - Deadlock prevention
  - Complete proof of soundness
At last, a ★ for F*!
At last, a ★ for F*!

- Separation logic in F*: Why so long?
  - Separation logic with SMT only is impossible
  - Meta-F*: Tactics + SMT make it possible
At last, a ★ for F★!

- Separation logic in F★: Why so long?
  - Separation logic with SMT only is impossible
  - Meta-F★: Tactics + SMT make it possible

- This summer: The right abstractions with resource typing to make it scale
At last, a ★ for F*!

- Separation logic in F*: Why so long?
  - Separation logic with SMT only is impossible
  - Meta-F*: Tactics + SMT make it possible

- **This summer:** The right abstractions with resource typing to make it scale

- Many applications targeted: Beyond crypto verification
  - Concurrent networking protocols, e.g. Quic
  - Critical systems components in Azure: Parts of Hyper-V? Azure CCF?
At last, a ★ for F*!

• Separation logic in F*: Why so long?
  • Separation logic with SMT only is impossible
  • Meta-F*: Tactics + SMT make it possible

• This summer: The right abstractions with resource typing to make it scale

• Many applications targeted: Beyond crypto verification
  • Concurrent networking protocols, e.g. Quic
  • Critical systems components in Azure: Parts of Hyper-V? Azure CCF?

• Many possible synergies:
  • Verifying Rust programs inside of Steel?
  • Verifying Verona components (ownership-based systems language from MSR Cambridge)?
At last, a ★ for F*!

• Separation logic in F*: Why so long?
  • Separation logic with SMT only is impossible
  • Meta-F*: Tactics + SMT make it possible

• This summer: The right abstractions with resource typing to make it scale

• Many applications targeted: Beyond crypto verification
  • Concurrent networking protocols, e.g. Quic
  • Critical systems components in Azure: Parts of Hyper-V? Azure CCF?

• Many possible synergies:
  • Verifying Rust programs inside of Steel?
  • Verifying Verona components (ownership-based systems language from MSR Cambridge)?

fromherz@cmu.edu          denis.merigoux@inria.fr
A frame rule for Steel

Classic frame rule:

\[
\frac{\{Q\} f \{R\}}{\{P \star Q\} f \{P \star R\}}
\]

In Steel:

```plaintext
val frame :
(P \star Q: resource) \rightarrow
(P \star R: resource) \rightarrow
(f: unit \rightarrow RST \alpha (expects Q) (provides R)) \rightarrow
RST \alpha
(expects (P \star Q))
(provides (P \star R))
```