Latest Improvements and Applications of Steel, a Concurrent Separation Logic for F*

Aymeric Fromherz
Inria Paris

What is F*

- A **proof-oriented**, functional programming language
- With support for dependent types, user-defined effects, …
- Semi-automated verification by relying on SMT solving
- Also offers a metaprogramming and tactic framework (Meta-F*)

F* Successes

- Vale/HACL*/EverCrypt:
  - A verified, large, industrial-grade cryptographic provider
  - Over 100k lines of verified C and Assembly code

    (~200k-300k lines of manually-written F* code)
  - Deployed in Firefox, Linux, Wireguard, Tezos, …

F* Successes

- Vale/HACL*/EverCrypt:
  - A verified, large, industrial-grade cryptographic provider
  - Over 100k lines of verified C and Assembly code
    (~200k-300k lines of manually-written F* code)
  - Deployed in Firefox, Linux, Wireguard, Tezos, …
- EverParse
  - Verified parsers and serializers for binary formats
  - Deployed in Microsoft Azure
- Also EverQuic, Noise*, Signal*, miTLS, …

F* Successes

- Vale/HACL*/EverCrypt:
  - A verified, large, industrial-grade cryptographic provider
  - Over 100k lines of verified C and Assembly code

    (~200k-300k lines of manually-written F* code)
  - Deployed in Firefox, Linux, Wireguard, Tezos, …
- EverParse
  - Verified parsers and serializers for binary formats
  - Deployed in Microsoft Azure
- Also EverQuic, Noise*, Signal*, miTLS, …
- But no concurrency, and memory reasoning is tedious

A Different Approach: Separation Logic

- Separating memory through the ⋆ operator: $P \star Q$

A Different Approach: Separation Logic

- Separating memory through the ⋆ operator: P ⋆ Q
- Modular heap reasoning through the Frame rule

$$\frac{\{Q\}\ c\ \{R\}}{\{P \star Q\}\ c\ \{P \star R\}}$$

A Different Approach: Separation Logic

- Separating memory through the ⋆ operator: P ⋆ Q
- Modular heap reasoning through the Frame rule

$$\frac{\{Q\}\ c\ \{R\}}{\{P \star Q\}\ c\ \{P \star R\}}$$

- Predicates to reason about memory: r ↦ v

$$\frac{\{r \mapsto v\}\ r\ \coloneqq 0\ \{r \mapsto 0\}}{\{s \mapsto u \star r \mapsto v\}\ r\ \coloneqq 0\ \{s \mapsto u \star r \mapsto 0\}}$$

- Many extensions (Concurrency, Resource usage, …)

Steel: An Overview

- A shallow embedding of Concurrent Separation Logic (CSL) in F*

Steel: An Overview

- A shallow embedding of Concurrent Separation Logic (CSL) in F*
- With an expressive, extensible program logic [ICFP’ 20]

  Partial Commutative Monoids (PCMs), Dynamically-allocated invariants, Monotonicity, Impredicativity, …

Steel: An Overview

- A shallow embedding of Concurrent Separation Logic (CSL) in F*
- With an expressive, extensible program logic [ICFP’ 20]

  Partial Commutative Monoids (PCMs), Dynamically-allocated invariants, Monotonicity, Impredicativity, …
- Automation through a cooperation between SMT solving and custom separation logic decision procedures [ICFP’ 21]

Steel: An Overview

- A shallow embedding of Concurrent Separation Logic (CSL) in F*
- With an expressive, extensible program logic [ICFP’ 20]

  Partial Commutative Monoids (PCMs), Dynamically-allocated invariants, Monotonicity, Impredicativity, …
- Automation through a cooperation between SMT solving and custom separation logic decision procedures [ICFP’ 21]
- Many verified, dependently-typed libraries (AVL trees, concurrent queues, lock-free concurrency, message-passing concurrency, …)

Steel by Example

Steel a p q: a computation that has return type a, under the precondition p, and with the postcondition q

```
let swap (p1 p2:ref int) : Steel unit
   (ptr p1 ★ ptr p2)
   (λ _→ ptr p1 ★ ptr p2)
   (requires λ _→ T)
   (ensures λ s0 _ s1 → s0.[p1] == s1.[p2] /\ s0.[p2] == s1.[p1])
```

Steel by Example

Steel a p q: a computation that has return type a, under the precondition p, and with the postcondition q

```
let swap (p1 p2:ref int) : Steel unit
   (ptr p1 ★ ptr p2)
   (λ _→ ptr p1 ★ ptr p2)
   (requires λ _→ T)
   (ensures λ s0 _ s1 → s0.[p1] == s1.[p2] /\ s0.[p2] == s1.[p1])
```

Expects two valid, disjoint pointers

Returns two valid, disjoint pointers

} Memory Shape

Steel by Example

Steel a p q: a computation that has return type a, under the precondition p, and with the postcondition q

```
let swap (p1 p2:ref int) : Steel unit
  (ptr p1 ★ ptr p2)
  (λ _→ ptr p1 ★ ptr p2)
  (requires λ _→ T)
  (ensures λ s0 _ s1 → s0.[p1] == s1.[p2] /\ s0.[p2] == s1.[p1])
```

Expects two valid, disjoint pointers

Returns two valid, disjoint pointers

} Memory Shape

} Functional Correctness

Steel by Example

Steel a p q: a computation that has return type a, under the precondition p, and with the postcondition q

```
let swap (p1 p2:ref int) : Steel unit
   (ptr p1 ★ ptr p2)
   (λ _→ ptr p1 ★ ptr p2)
   (requires λ _→ T)
   (ensures λ s0 _ s1 → s0.[p1] == s1.[p2] /\ s0.[p2] == s1.[p1])
= let v1 = read p1 in
   let v2 = read p2 in
   write p1 v2;
   write p2 v1
```

← Expects two valid, disjoint pointers

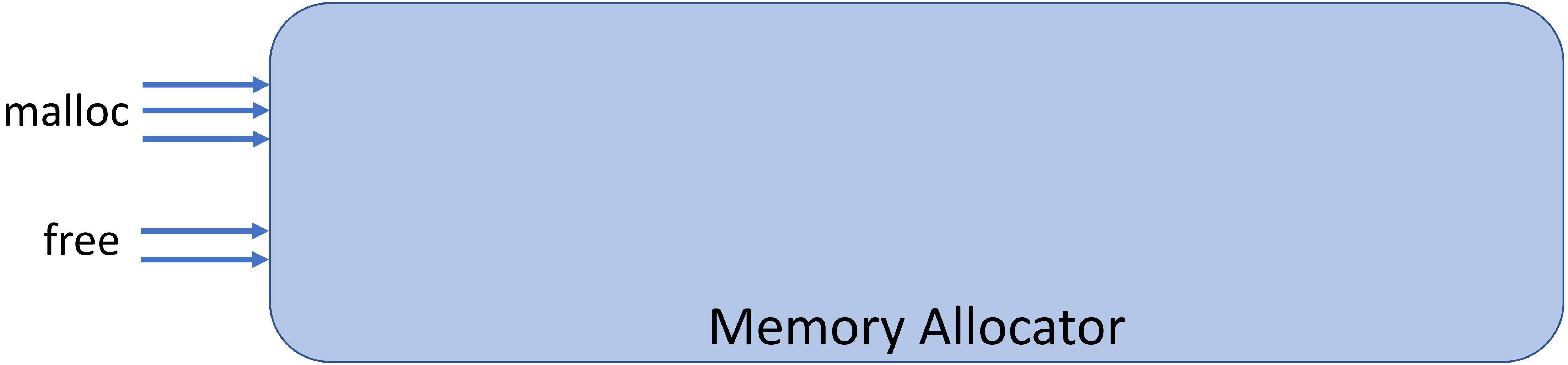
← Returns two valid, disjoint pointers

} Memory Shape

} Functional Correctness

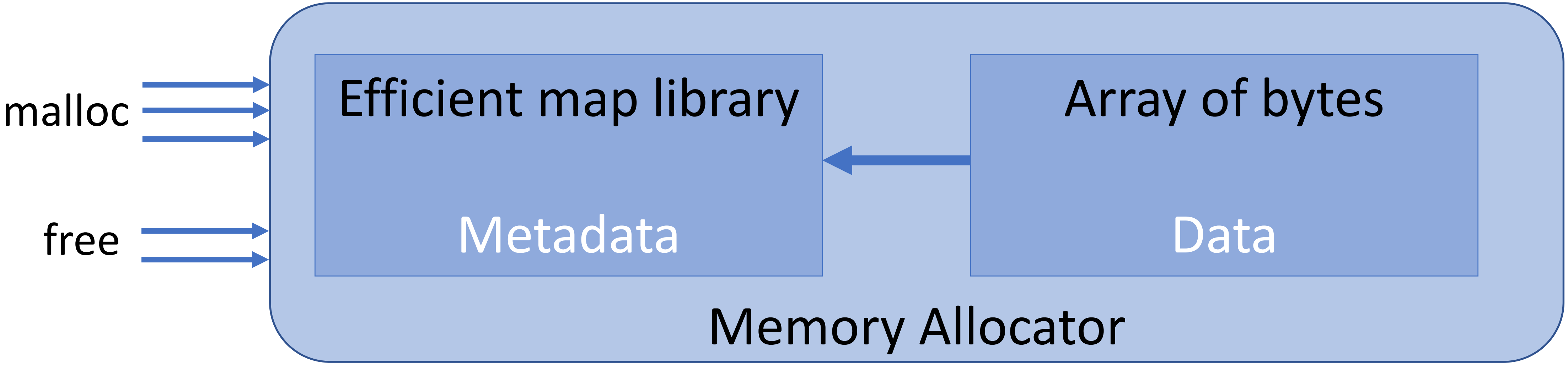
Ongoing Project: Verifying a Memory Allocator

- **<u>Goal:</u>** Develop a verified, performant, concurrent **memory allocator** with modern security defenses in Steel



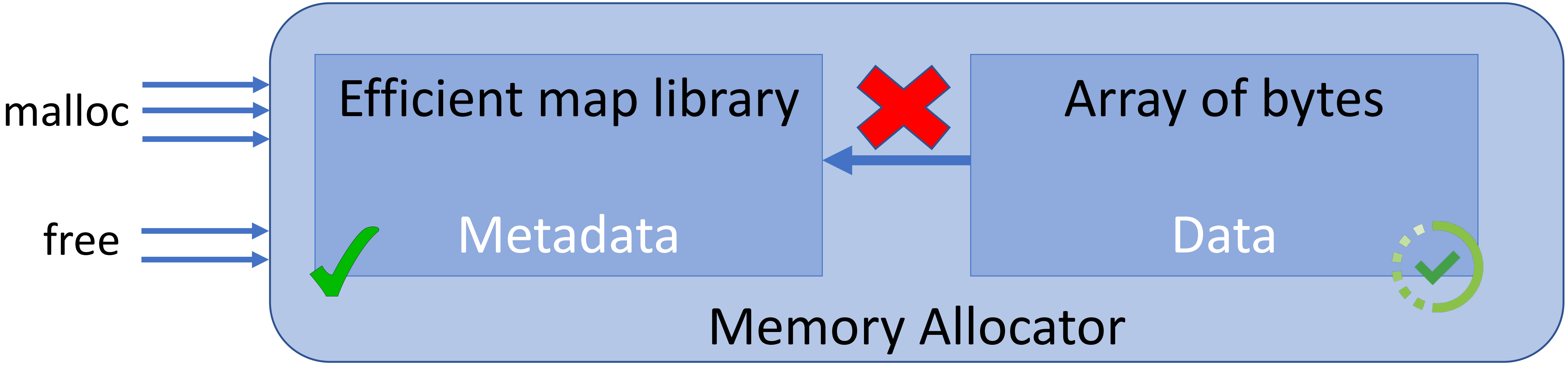
Ongoing Project: Verifying a Memory Allocator

- **<u>Goal:</u>** Develop a verified, performant, concurrent **memory allocator** with modern security defenses in Steel



Ongoing Project: Verifying a Memory Allocator

- **<u>Goal:</u>** Develop a verified, performant, concurrent **memory allocator** with modern security defenses in Steel



- **<u>Current status:</u>**
  - Partially verified C implementation
  - Working with the Zathura PDF viewer

Ongoing Project: Improving Automation

- **Problem:** How to reduce SMT response time to make verification more developer-friendly?
- **Idea**: Leverage specification logic specifications to reason on a *functionalized version* of the program

Ongoing Project: Improving Automation

- **Problem:** How to reduce SMT response time to make verification more developer-friendly?
- **Idea**: Leverage specification logic specifications to reason on a *functionalized version* of the program

```
let swap (p1 p2:ref int) : Steel unit
  (ptr p1 ★ ptr p2) (ptr p1 ★ ptr p2)


= let v1 = read p1 in
  let v2 = read p2 in
  write p1 v2; write p2 v1
```

Ongoing Project: Improving Automation

- **Problem:** How to reduce SMT response time to make verification more developer-friendly?
- **Idea**: Leverage specification logic specifications to reason on a *functionalized version* of the program

```
let swap (p1 p2:ref int) : Steel unit
  (ptr p1 ★ ptr p2) (ptr p1 ★ ptr p2)

= let v1 = read p1 in
  let v2 = read p2 in
  write p1 v2; write p2 v1
```

→

```
let swap_func (p1 p2:int)
  : Pure (int * int)

= let v1 = p1 in
  let v2 = p2 in
  return (v2, v1)
```

Ongoing Project: Improving Automation

- **<u>Problem:</u>** How to reduce SMT response time to make verification more developer-friendly?
- **<u>Idea</u>**: Leverage specification logic specifications to reason on a *functionalized version* of the program

```
let swap (p1 p2:ref int) : Steel unit
  (ptr p1 ★ ptr p2) (ptr p1 ★ ptr p2)
  (ensures λ s0 _ s1 → s0.[p1] == s1.[p2])
= let v1 = read p1 in
  let v2 = read p2 in
  write p1 v2; write p2 v1
```

→

```
let swap_func (p1 p2:int)
  : Pure (int * int)
 (ensures λ (p1′, p2′) →  p1 == p2’)
= let v1 = p1 in
  let v2 = p2 in
  return (v2, v1)
```

- Translation is entirely done using tactics, and is hence **provably sound**

A Vision for Steel

- **Steel**: A foundation for high-assurance systems programming
  - Extraction to verified C
  - Support for lock-free concurrency
  - High level of automation through a mixture of tactics and SMT

A Vision for Steel

- **Steel**: A foundation for high-assurance systems programming
  - Extraction to verified C
  - Support for lock-free concurrency
  - High level of automation through a mixture of tactics and SMT

- Ongoing and Future Directions:
  - Verification of a secure memory allocator
  - Improve the programmability, usability and tooling
  - End-to-end verification of secure communication protocols
  - Drop-in replacements for high-assurance Rust libraries

aymeric.fromherz@inria.fr