Deductive Verification for Rust Programs

Xavier Denis
ETH Zürich

4 Juin 2024

Why Verify?

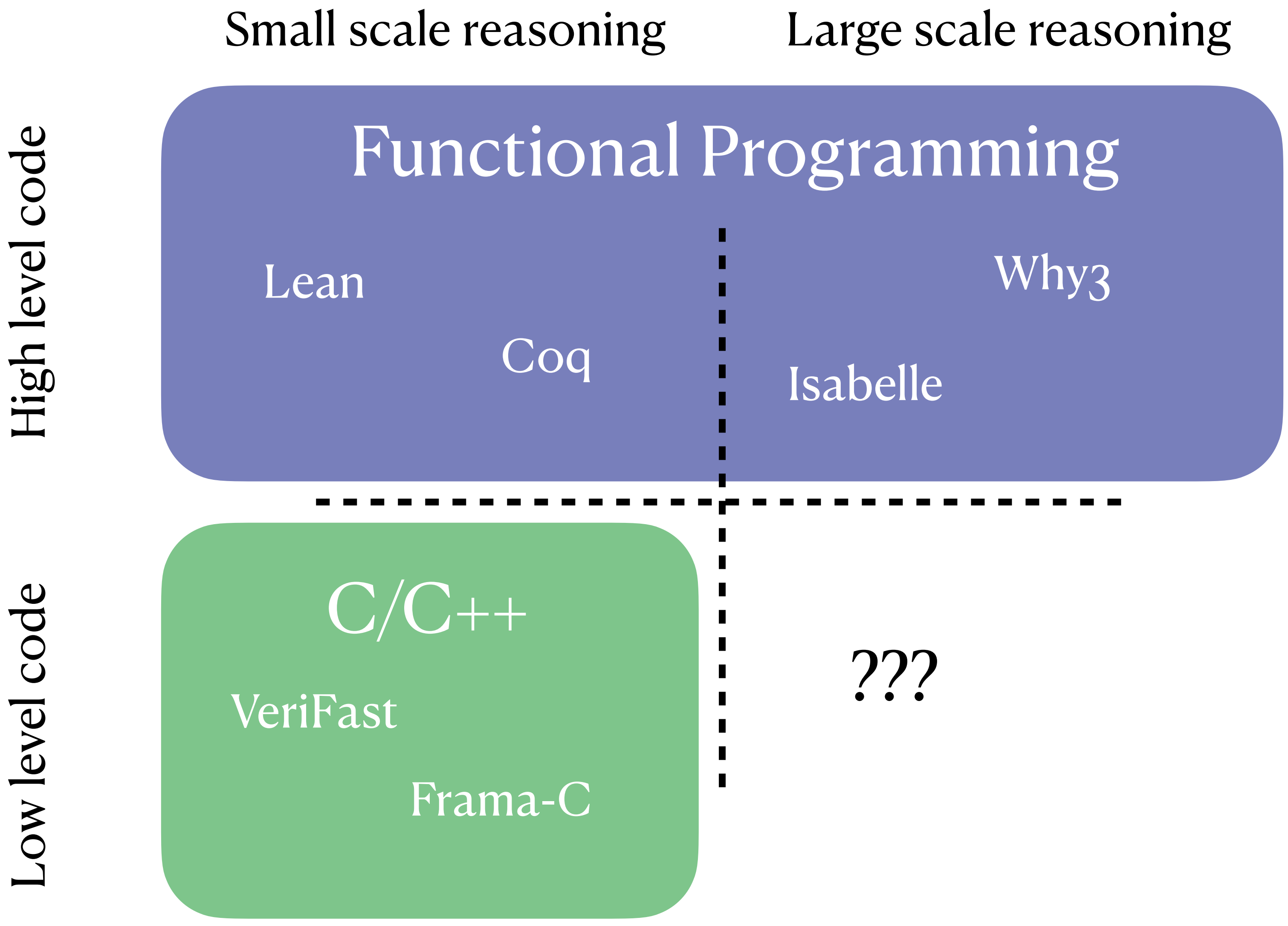
Formal verification is traditionally applied to critical systems, where computers make life-or-death decisions

Aeronautics Rail transport Automotive

Today, many other candidates for formal verification exist...

Industrial Control Cloud Platforms

What is the **problem** with verification?



Small scale reasoning

Large scale reasoning

High level code

Low level code

Functional Programming

Lean

Coq

Why3

Isabelle

C/C++

VeriFast

Frama-C

???

What is the problem with verification?

```
void memcpy(char* tgt, char* src, size_t size) {
    for (int k = 0; k < size; k++) {
        tgt[k] = src[k];
    }
}
```

We want to prove that **`size`** bytes of **`src`** are copied to **`tgt`**

What is the problem with verification?

Pointers could overlap

```
void memcpy(char* tgt, char* src, size_t size) {
    for (int k = 0; k < size; k++) {
        tgt[k] = src[k];
    }
}
```

We want to prove that **`size`** bytes of **`src`** are copied to **`tgt`**

What is the problem with verification?

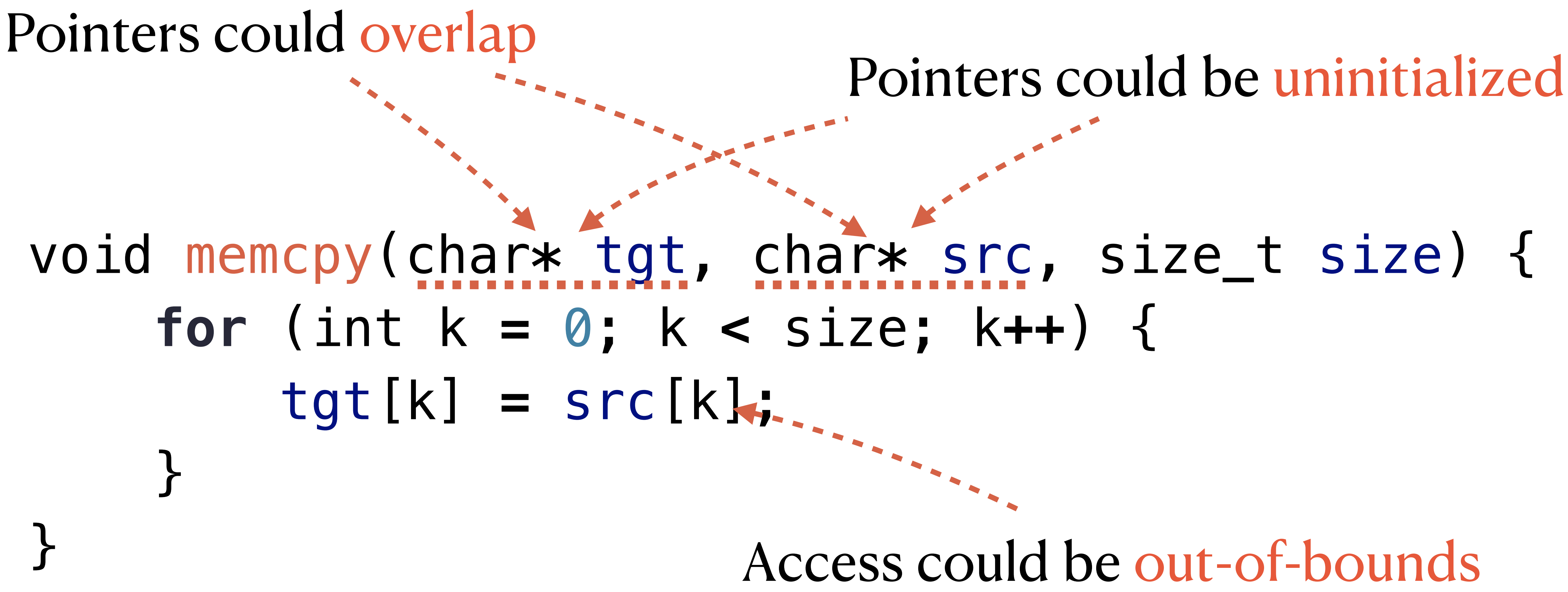
Pointers could overlap

Pointers could be uninitialized

```
void memcpy(char* tgt, char* src, size_t size) {
    for (int k = 0; k < size; k++) {
        tgt[k] = src[k];
    }
}
```

We want to prove that **`size`** bytes of **`src`** are copied to **`tgt`**

What is the problem with verification?



Pointers could overlap

Pointers could be uninitialized

```
void memcpy(char* tgt, char* src, size_t size) {
    for (int k = 0; k < size; k++) {
        tgt[k] = src[k];
    }
}
```

Access could be out-of-bounds

We want to prove that **`size`** bytes of **`src`** are copied to **`tgt`**

What is the problem with verification?

Verified using VeriFast

```
void memcpy(char* src, char* tgt, size_t size)
//@ requires chars(tgt, size, _) &*& chars(src, size, ?s);
//@ ensures chars(tgt, size, s) &*& chars(src, size, s);
{
  for(int k= 0; k < size; k++)
  /*@
     invariant 0 <= k &*& k <= size &*& chars(tgt + k, size - k, _)
       &*& chars(tgt, k, take(k, s)) &*& chars(src, k, take(k, s))
       &*& chars(src + k, size - k, drop(k, s));
  @*/
 {
    //@ open chars(tgt + k, _, _);
    //@ open chars(src + k, _, _);
    tgt[k] = src[k];
    //@ drop_n_plus_one(k, str);
    //@ assert character(tgt + k, ?c0) &*& nth(k, s) == c0;
    //@ append_take_take(s, k, 1);
    //@ close chars(tgt + k, 1, _);
    //@ close chars(src + k, 1, _);
  }
}
```

What is the problem with verification?

Verified using VeriFast

```
void memcpy(char* src, char* tgt, size_t size)
//@ requires chars(tgt, size, _) &*& chars(src, size, ?s);
//@ ensures chars(tgt, size, s) &*& chars(src, size, s);
{
  for(int k= 0; k < size; k++)
  /*@
     invariant 0 <= k &*& k <= size &*& chars(tgt + k, size - k, _)
       &*& chars(tgt, k, take(k, s)) &*& chars(src, k, take(k, s))
       &*& chars(src + k, size - k, drop(k, s));
  @*/
 {
    //@ open chars(tgt + k, _, _);
    //@ open chars(src + k, _, _);
    tgt[k] = src[k];
    //@ drop_n_plus_one(k, str);
    //@ assert character(tgt + k, ?c0) &*& nth(k, s) == c0;
    //@ append_take_take(s, k, 1);
    //@ close chars(tgt + k, 1, _);
    //@ close chars(src + k, 1, _);
  }
}
```

What is the problem with verification?

Formal verification must consider all possible behaviors.

In C-like languages this means…

- Mutable aliasing, pointer arithmetic
- Pervasive undefined behavior
- Weak abstractions in types

Tools like VeriFast do the best given these constraints, improving requires a better language…

What is Rust?

Introduced in 2015, Rust is designed to solve some problems of C

Features a novel ownership type-system

- Forbids mutable aliasing, accessing uninitialized memory

Includes useful high-level features: sum types, closures, traits

From C to Rust

Pointers are disjoint

```
fn memcpy(tgt: &mut [u8], src: & [u8]) {
    let mut i = 0;
    while i < src.len() {
        tgt[i] = src[i];
        i += 1;
    }
}
```

Out-of-bounds access is detected at runtime

Verifying Rust programs?

Rust has grown popular among developers of systems software

- Interest from automotive, cloud, and other critical systems

The Rust type system eliminates many common bugs, but not all.

- Can we leverage it to simplify verification?

Contributions of my thesis

This thesis presents the Creusot verifier for Rust.

- Design and implementation of the actual tool
- Metatheory for a core model of Creusot

It also considers various applications of Creusot:

- Verified Iterators in Rust
- Sprout, a formally verified SMT solver

In this presentation

In the rest of this talk I present…

- Creusot's approach to verification
- Its usage for verifying Iterators
- The metatheory and soundness of Creusot
- An overview of software verified using Creusot

Creusot's approach to verification

The Creusot approach to verification

Creusot hails from a lineage of work starting with RustHorn

- *Matsushita, Y., Tsukada, T. and Kobayashi, N. ESOP 2020*
- Introduces a technique of prophecies to reason about pointers

Creusot uses this to translate Rust programs to functional ones

- Resulting functional programs can be verified with Why3

The big secret: **Rust is a functional* language**

*some squinting required

Encoding Rust in ML

Local variables

```
fn incr(mut x: u64, mut y: u64)
-> u64 {
    x += y;
    x
}
```

```
let incr x y =
  let x = x + y in
  x
```

Locally mut variables can be modeled as shadowing

Encoding Rust in ML

Box

```
fn incr(x: Box<u64>, y: Box<u64>)
-> Box<u64> {
    *x += *y;
    x
}
```

?

Encoding Rust in ML

Box

```
fn incr(x: Box<u64>, y: Box<u64>)
-> Box<u64> {
    *x += *y;
    x
}
```

```
let incr x y =
  let x = x + y in
  x
```

Boxes are erased!

Consequence of uniqueness

Encoding Rust in ML

Immutable References

```
fn incr_immut(x: &u64, y: &u64)
-> u64 {
     *x + *y
}
```

?

Encoding Rust in ML

Immutable References

```
fn incr_immut(x: &u64, y: &u64)
-> u64 {
    *x + *y
}
```

```
let incr_immut x y =
  x + y
```

Also erased!

No mutation = No problems

Encoding Rust in ML

Mutable References

```
fn incr_mut(x: &mut u64, y: u64)
{
    *x += y
}


fn main() {
    let mut x = 0;
    incr_mut(&mut x, 10);
    assert!(x == 10);
}
```

?

Encoding Rust in ML

Mutable References

```rust
fn incr_mut(x: &mut u64, y: u64)
{
    *x += y
}
```

```ocaml
let incr_mut x y = ???
```

```rust
fn main() {
    let mut x = 0;
    incr_mut(&mut x, 10);
    assert!(x == 10);
}
```

```ocaml
let main () =
    let x = 0 in
    incr_mut x 10;
    assert { x == 10 }
```

Mutable borrows can't be erased.

They require a special encoding

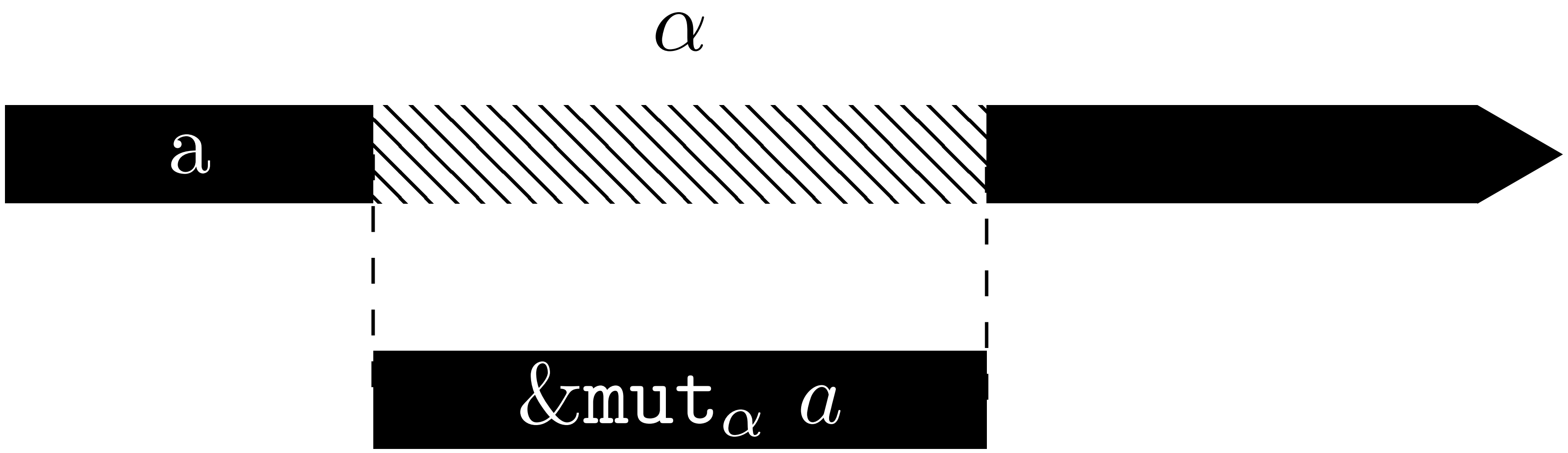
Prophecies

Synchronizing lender and borrower

Models mutable borrows as pair of **current** and **final** values

We prophetize the final value, which the lender recovers.

Depends on **uniqueness** and **lifetimes** of mutable borrows



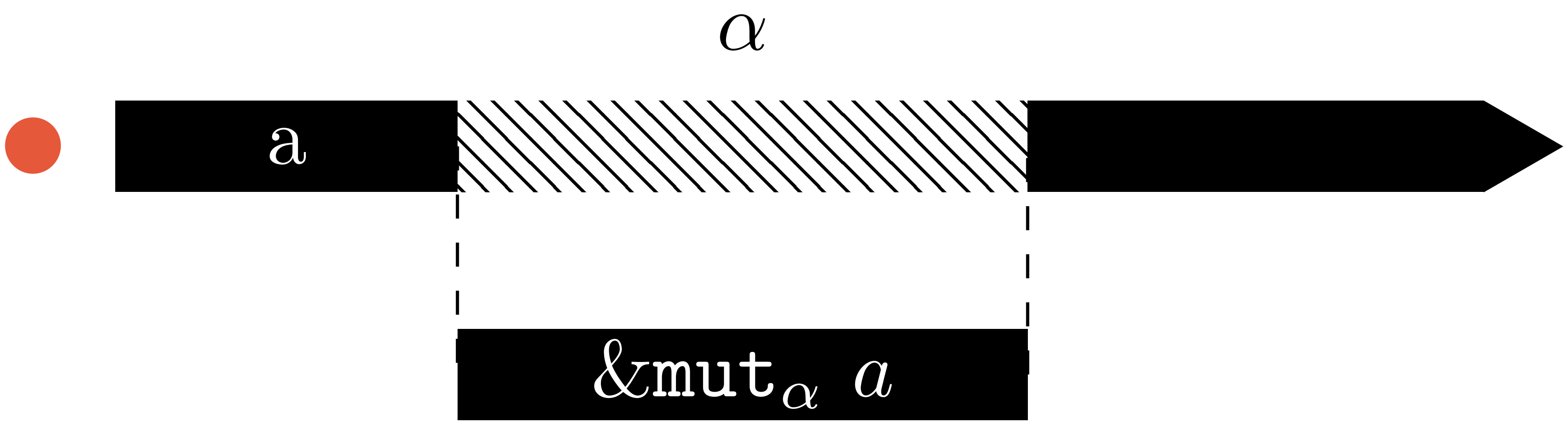
Prophecies

Synchronizing lender and borrower

Models mutable borrows as pair of **current** and **final** values

We prophetize the final value, which the lender recovers.

Depends on **uniqueness** and **lifetimes** of mutable borrows



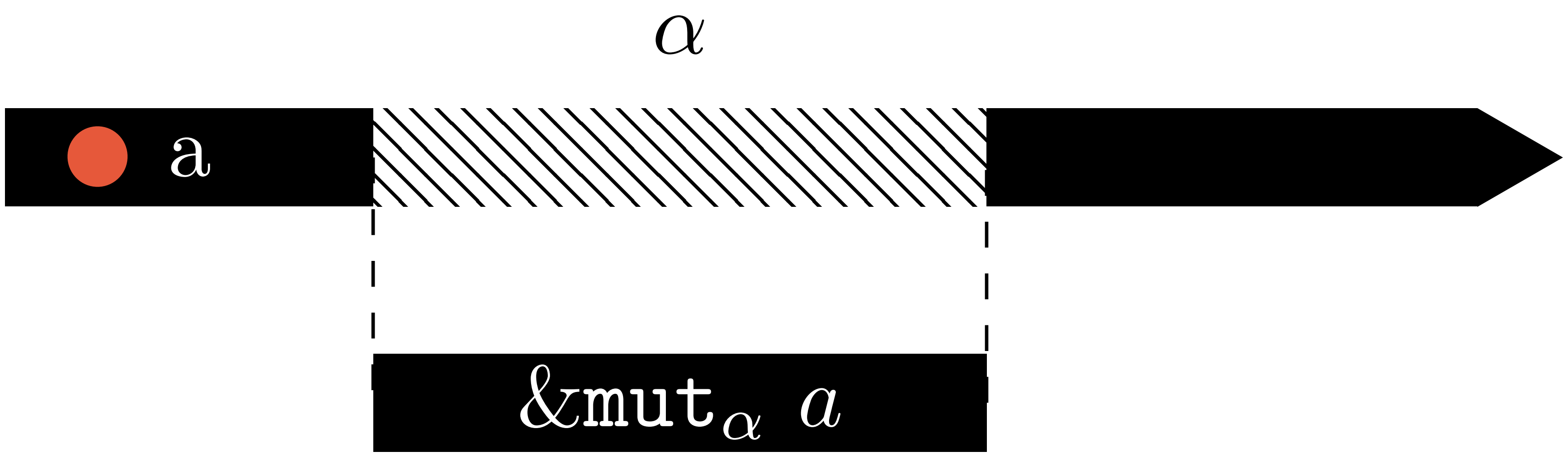
Prophecies

Synchronizing lender and borrower

Models mutable borrows as pair of **current** and **final** values

We prophetize the final value, which the lender recovers.

Depends on **uniqueness** and **lifetimes** of mutable borrows



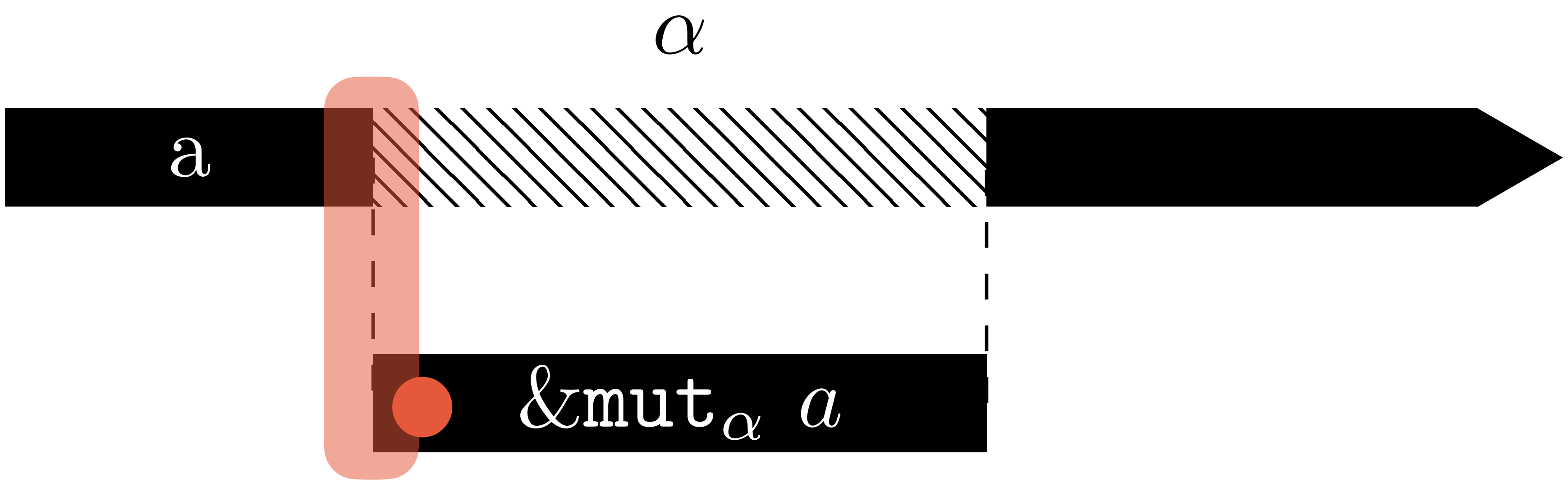
Prophecies

Synchronizing lender and borrower

Models mutable borrows as pair of **current** and **final** values

We prophetize the final value, which the lender recovers.

Depends on **uniqueness** and **lifetimes** of mutable borrows



a is inaccessible for the duration of α

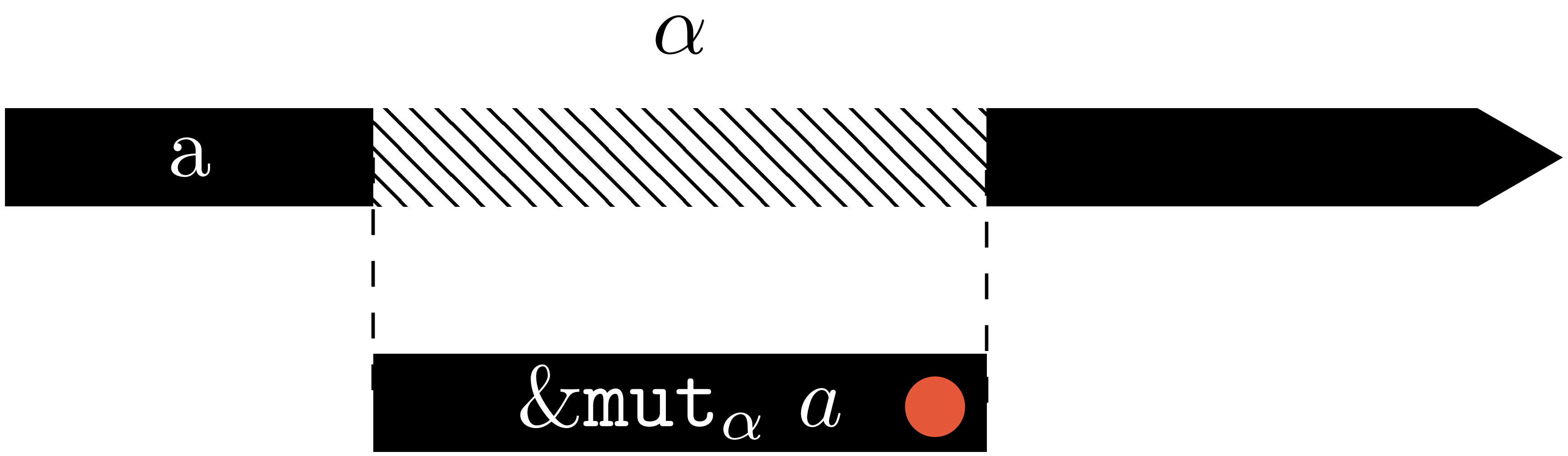
Prophecies

Synchronizing lender and borrower

Models mutable borrows as pair of **current** and **final** values

We prophetize the final value, which the lender recovers.

Depends on **uniqueness** and **lifetimes** of mutable borrows



a is inaccessible for the duration of α

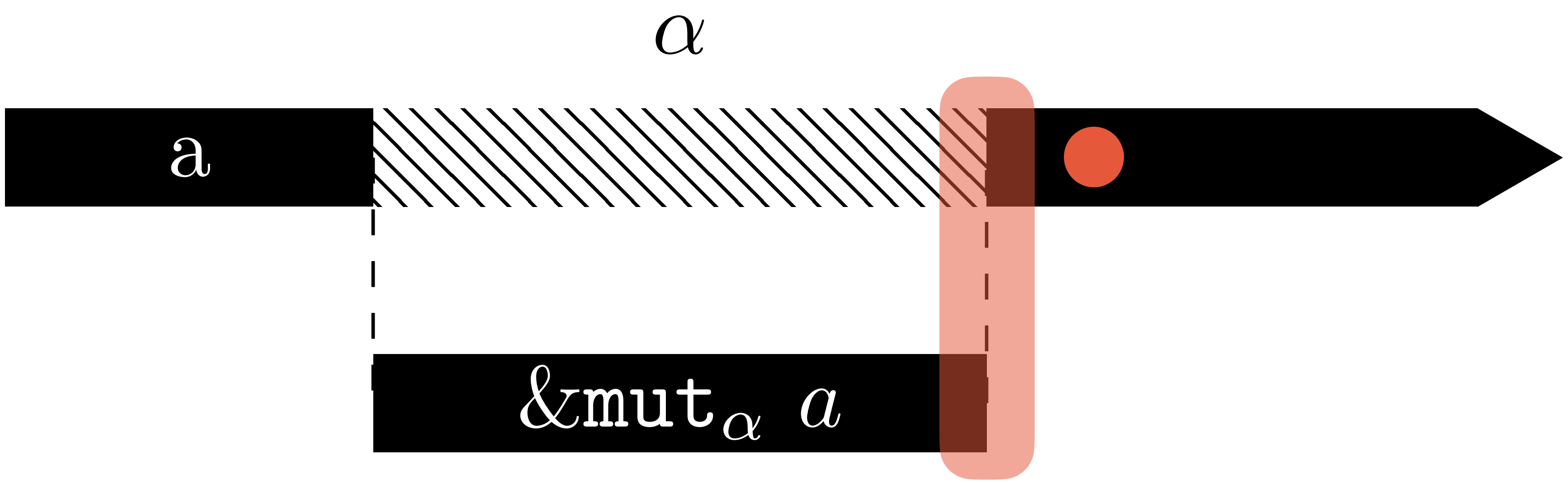
Prophecies

Synchronizing lender and borrower

Models mutable borrows as pair of **current** and **final** values

We prophetize the final value, which the lender recovers.

Depends on **uniqueness** and **lifetimes** of mutable borrows



Can't model the second point; instead *prophetize it*

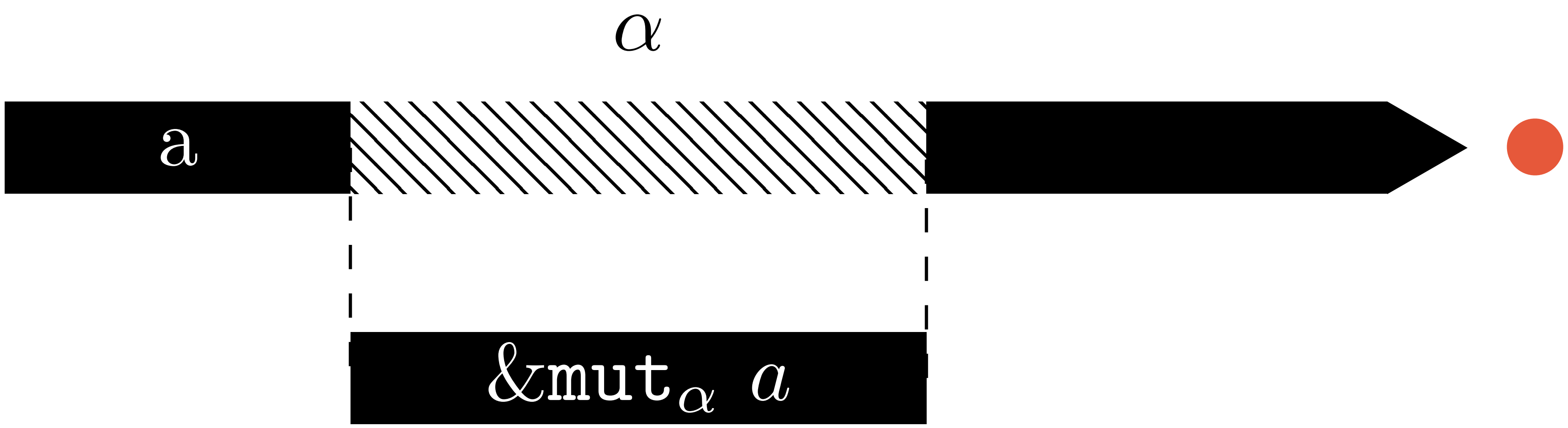
Prophecies

Synchronizing lender and borrower

Models mutable borrows as pair of **current** and **final** values

We prophetize the final value, which the lender recovers.

Depends on **uniqueness** and **lifetimes** of mutable borrows



Can’t model the second point; instead *prophetize it*

Prophecies

Synchronizing lender and borrower

We encode this using *any/assume non-determinism.*

- **`any`** will non-deterministically create a value
- **`assume`** places constraints on *past* choices

Creation

```
let borwr = { cur = lendr; fin = any } in
let lendr = borwr.fin in
```

Resolution

```
assume { borwr.cur = borwr.fin }
```

Encoding Rust in ML

Mutable References

```rust
fn main() {
  let mut a = 0;
  let x = &mut a;
  let y = 10;
  *x += y;
  drop(x);
  assert_eq!(a, 10);
}
```

```
let main () =
  let a = 0 in
  let x = { cur = a ; fin = any } in
  let a = x.fin in
  let y = 10 in
  let x = { x with cur += y } in
  assume { x.fin = x.cur };
  assert { a = 10 }
```

Encoding Rust in ML

Mutable References?

```
fn main() {
  let mut a = 0;
  let x = &mut a;
  let y = 10;
  *x += y;
  drop(x);
  assert_eq!(a, 10);
}
```

```
let main () =
  let a = 0 in
  let x = { cur = a ; fin = any } in
  let a = x.fin in
  let y = 10 in
  let x = { x with cur += y } in
  assume { x.fin = x.cur };
  assert { a = 10 }
```

Encoding Rust in ML

Mutable References?

```
fn main() {
  let mut a = 0;
  let x = &mut a;
  let y = 10;
  *x += y;
  drop(x);
  assert_eq!(a, 10);
}
```

```
let main () =
  let a = 0 in
  let x = { cur = a ; fin = any } in
  let a = x.fin in
  let y = 10 in
  let x = { x with cur += y } in
  assume { x.fin = x.cur };
  assert { a = 10 }
```

Encoding Rust in ML

Mutable References?

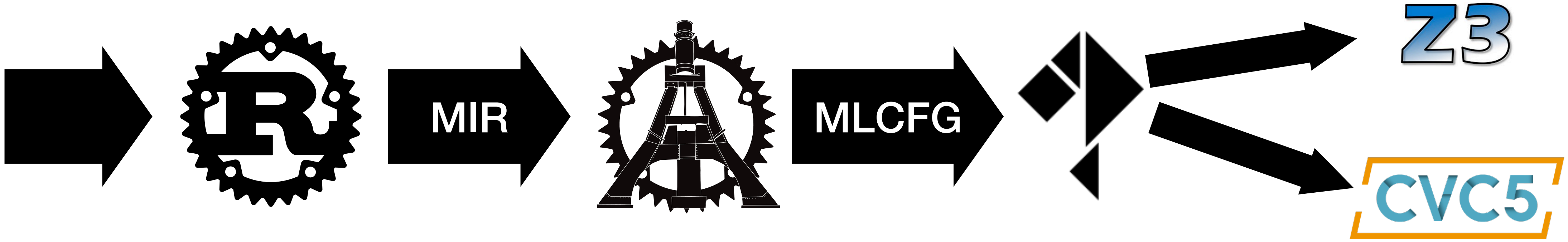
```rust
fn main() {
  let mut a = 0;
  let x = &mut a;
  let y = 10;
  *x += y;
  drop(x);
  assert_eq!(a, 10);
}
```

```ocaml
let main () =
  let a = 0 in
  let x = { cur = a ; fin = any } in
  let a = x.fin in
  let y = 10 in
  let x = { x with cur += y } in
  assume { x.fin = x.cur };
  assert { a = 10 }
```

Creusot today

Creusot verifies programs by translation to a functional language

Resulting proof obligations are discharged by Why3



Includes an expressive specification language to write contracts

Also, logical functions, ghost code, and ghost fields

Beyond the status quo

Verifying with Creusot

```
fn memcpy(tgt: &mut [u8], src: & [u8]) {
    let mut i = 0;


    while i < src.len() {
        tgt[i] = src[i];
    }
}
```

Beyond the status quo

Verifying with Creusot

```rust
#[requires(tgt.len() == src.len())]

fn memcpy(tgt: &mut [u8], src: & [u8]) {
    let mut i = 0;


    while i < src.len() {
        tgt[i] = src[i];
    }
}
```

Beyond the status quo

Verifying with Creusot

```
#[requires(tgt.len() == src.len())]
#[ensures(^tgt == *src)]
fn memcpy(tgt: &mut [u8], src: & [u8]) {
    let mut i = 0;


    while i < src.len() {
        tgt[i] = src[i];
    }
}
```

Beyond the status quo

Verifying with Creusot

```
#[requires(tgt.len() == src.len())]
#[ensures(^tgt == *src)]
fn memcpy(tgt: &mut [u8], src: & [u8]) {
    let mut i = 0;


    while i < sr  .len() {
        tgt[i] =  rc[i];
    }
}
```

The ^ operator accesses the *final* value of a borrow

Beyond the status quo

Verifying with Creusot

```
#[requires(tgt.len() == src.len())]
#[ensures(^tgt == *src)]
fn memcpy(tgt: &mut [u8], src: & [u8]) {
    let mut i = 0;
    #[invariant(i <= src.len())]
    #[invariant(forall<j> j < i ==> tgt[j] == src[i])]
    while i < src.len() {
        tgt[i] = src[i];
    }
}
```

Soundness of Prophecies

Soundness of Prophecies

Prophetic translation is subtle and hard to justify

- *Does a value always exist for a prophecy?*

Syntactic approaches to soundness don’t work well for Rust

- *Unsafe code* allows ‘extending’ Rust with new ‘primitive’ types
- RustBelt solves this using a *semantic* model of Rust type system

RustBelt

What is it?

A formal model of the core Rust type system.

Uses a language called λ_{Rust} approximating **MIR**

Typing judgments are *theorems* in Iris, resulting system is *open*

New rules can be proved *post-hoc* without affecting soundness

Theorem. *(Adequacy). For any λ_{Rust} function f such that*
$\emptyset \mid \emptyset \vdash f \dashv x.x : \mathsf{fn}(\emptyset) \to ()$ *holds,*
no execution of f ends in a stuck state.

RustHornBelt

Type-Spec Judgements

Extends the type judgements of **RustBelt** with specifications

- Specifications are provided as *predicate transformers*

Requires a novel prophecy resource algebra for Iris

- Uses a ‘many-worlds’ interpretation of prophecies

Reuses much of the proof architecture; a natural extension

RustHornBelt

Type-Spec Judgements

$$\mathrm{E}; \mathrm{L} \mid \mathrm{T} \vdash I \dashv \mathrm{T}' \rightsquigarrow \Phi$$

RustHornBelt

Type-Spec Judgements

Lifetime Contexts

Type Context

$$\underbrace{E; L \mid T}_{\text{Input Contexts}} \vdash I \dashv T' \rightsquigarrow \Phi$$

RustHornBelt

Type-Spec Judgements

$$\mathrm{E}; \mathrm{L} \mid \mathrm{T} \vdash \underset{\text{Instruction}}{\underset{\cdot\cdot\cdot\cdot}{I}} \dashv \mathrm{T}' \rightsquigarrow \Phi$$

RustHornBelt

Type-Spec Judgements

$$\mathrm{E}; \mathrm{L} \mid \mathrm{T} \vdash I \dashv \underset{\cdots\cdot}{\mathrm{T}'} \rightsquigarrow \Phi$$

Output Contexts

RustHornBelt

Type-Spec Judgements

$$\mathrm{E}; \mathrm{L} \mid \mathrm{T} \vdash I \dashv \mathrm{T}' \rightsquigarrow \underset{\text{Specification}}{\underset{\cdots\cdot}{\Phi}}$$

RustHornBelt

Interpretation of judgments

$$
\begin{aligned}
&\llbracket \mathbf{L} \mid \mathbf{T} \vdash I \dashv a.\ \mathbf{T}' \rightsquigarrow \Phi \rrbracket \triangleq \\
&\quad \forall\, \hat{\Psi}.\ \Big\{ \exists\, \overline{\hat{a}}.\ \llbracket \mathbf{L} \rrbracket * \llbracket \mathbf{T} \rrbracket (\overline{\hat{a}}) \\
&\qquad\quad * \big\langle \lambda\pi.\Phi\, (\hat{\Psi}\, \pi)\, (\overline{\hat{a}\, \pi}) \big\rangle \Big\} \\
&\qquad\quad I \Big\{ r.\ \exists\, \overline{\hat{b}}.\ \llbracket \mathbf{L} \rrbracket * \llbracket \mathbf{T}' \rrbracket (\overline{\hat{b}}) \\
&\qquad\qquad\quad * \big\langle \lambda\pi.(\hat{\Psi}\, \pi)\, (\overline{\hat{b}\, \pi}) \big\rangle \Big\}
\end{aligned}
$$

RustHornBelt

Interpretation of judgments

$$
\begin{aligned}
&\llbracket \mathbf{L} \mid \mathbf{T} \vdash I \dashv a.\ \ \mathbf{T}' \rightsquigarrow \Phi \rrbracket \triangleq \\
&\qquad \forall\, \hat{\Psi}.\ \Big\{ \exists\, \overline{\hat{a}}.\ \llbracket \mathbf{L} \rrbracket * \llbracket \mathbf{T} \rrbracket (\overline{\hat{a}}) \\
&\qquad\qquad * \left\langle \lambda\pi. \Phi\, (\hat{\Psi}\, \pi)\, (\overline{\hat{a}\, \pi}) \right\rangle \Big\} \\
&\qquad\qquad I \Big\{\, r.\ \exists\, \overline{\hat{b}}.\ \llbracket \mathbf{L} \rrbracket * \llbracket \mathbf{T}' \rrbracket (\overline{\hat{b}}) \\
&\qquad\qquad\qquad * \left\langle \lambda\pi. (\hat{\Psi}\, \pi)\, (\overline{\hat{b}\, \pi}) \right\rangle \Big\}
\end{aligned}
$$

An instruction is well-typed if:

RustHornBelt

Interpretation of judgments

$$\begin{aligned}
&[\![\mathbf{L} \mid \mathbf{T} \vdash I \dashv a.\ \mathbf{T}' \rightsquigarrow \Phi]\!] \triangleq \\
&\quad \forall\, \hat{\Psi}.\ \{\exists\, \overline{\hat{a}}.\ [\![\mathbf{L}]\!] * [\![\mathbf{T}]\!](\overline{\hat{a}}) \\
&\qquad\quad * \langle \lambda\pi.\Phi\, (\hat{\Psi}\,\pi)\, (\overline{\hat{a}\,\pi}) \rangle \} \\
&\qquad\quad I \{ r.\ \exists\, \overline{\hat{b}}.\ [\![\mathbf{L}]\!] * [\![\mathbf{T}']\!](\overline{\hat{b}}) \\
&\qquad\qquad\quad * \langle \lambda\pi.(\hat{\Psi}\,\pi)\, (\overline{\hat{b}\,\pi}) \}
\end{aligned}$$

An instruction is well-typed if:

Given resources for **L** and **T**...

RustHornBelt

Interpretation of judgments

$$
\begin{aligned}
&[\![\mathbf{L} \mid \mathbf{T} \vdash I \dashv a.\ \mathbf{T}' \rightsquigarrow \Phi]\!] \triangleq \\
&\quad \forall\, \hat{\Psi}.\ \{\exists\, \overline{\hat{a}}.\ [\![\mathbf{L}]\!] * [\![\mathbf{T}]\!](\overline{\hat{a}}) \\
&\qquad\qquad * \langle \lambda\pi. \Phi\,(\hat{\Psi}\,\pi)\,(\overline{\hat{a}\,\pi})\rangle\} \\
&\qquad I\{\,r.\ \exists\, \overline{\hat{b}}.\ [\![\mathbf{L}]\!] * [\![\mathbf{T}']\!](\overline{\hat{b}}) \\
&\qquad\qquad\qquad * \langle \lambda\pi.(\hat{\Psi}\,\pi)\,(\overline{\hat{b}\,\pi})\rangle\}
\end{aligned}
$$

An instruction is well-typed if:

Given resources for **L** and **T**...

...and a precondition from Φ...

RustHornBelt

Interpretation of judgments

$$
\begin{aligned}
&[\![\mathbf{L} \mid \mathbf{T} \vdash I \dashv a.\ \mathbf{T}' \rightsquigarrow \Phi]\!] \triangleq \\
&\quad \forall\, \hat{\Psi}.\ \{\exists\, \overline{\hat{a}}.\ [\![\mathbf{L}]\!] * [\![\mathbf{T}]\!](\overline{\hat{a}}) \\
&\qquad * \langle \lambda\pi.\Phi\, (\hat{\Psi}\, \pi)\, (\overline{\hat{a}\, \pi}) \rangle \} \\
&\qquad I \{\, r.\ \exists\, \overline{\hat{b}}.\ [\![\mathbf{L}]\!] * [\![\mathbf{T}']\!](\overline{\hat{b}}) \\
&\qquad\qquad * \langle \lambda\pi.(\hat{\Psi}\, \pi)\, (\overline{\hat{b}\, \pi}) \rangle \}
\end{aligned}
$$

An instruction is well-typed if:

Given resources for **L** and **T**...

...and a precondition from Φ...

...executing *I* gives back **L** and **T**′...

RustHornBelt

Interpretation of judgments

$$
\begin{aligned}
&[\![\mathbf{L} \mid \mathbf{T} \vdash I \dashv a.\ \mathbf{T}' \rightsquigarrow \Phi]\!] \triangleq \\
&\quad \forall\, \hat{\Psi}.\ \Big\{ \exists\, \overline{\hat{a}}.\ [\![\mathbf{L}]\!] * [\![\mathbf{T}]\!](\overline{\hat{a}}) \\
&\qquad\qquad * \langle \lambda\pi.\Phi\, (\hat{\Psi}\, \pi)\, (\overline{\hat{a}\, \pi}) \rangle \Big\} \\
&\qquad\quad I \Big\{ r.\ \exists\, \overline{\hat{b}}.\ [\![\mathbf{L}]\!] * [\![\mathbf{T}']\!](\overline{\hat{b}}) \\
&\qquad\qquad\qquad * \langle \lambda\pi.(\hat{\Psi}\, \pi)\, (\overline{\hat{b}\, \pi}) \rangle \Big\}
\end{aligned}
$$

An instruction is well-typed if:

Given resources for **L** and **T**...

...and a precondition from Φ...

...executing *I* gives back **L** and **T**′...

...and postcondition Ψ

RustHornBelt

Example Judgements

MUTBOR-BOR

$$\alpha \mid \mathtt{a} : \mathbf{\mathtt{own}}\ \mathtt{T} \vdash \&\mathbf{\mathtt{mut}}\ \ast\ \mathtt{a} \dashv \mathtt{b}.\ \mathtt{a} :^{\dagger\alpha} \mathbf{\mathtt{own}}\ \mathtt{T}, \mathtt{b} : \&_{\alpha}\mathbf{\mathtt{mut}}\ \mathtt{T}$$

$$\rightsquigarrow \lambda\Psi, [a].\forall a'.\Psi[a', (a, a')]$$

MUTBOR-WRITE

$$\alpha \mid \mathtt{b} : \&_{\alpha}\mathbf{\mathtt{mut}}\ \mathtt{T}, \mathtt{c} : \mathtt{T} \vdash \ast\mathtt{b}\ =\ \mathtt{c} \dashv \mathtt{b}.\ \mathtt{b} : \&_{\alpha}\mathbf{\mathtt{mut}}\ \mathtt{T}$$

$$\rightsquigarrow \lambda\Psi, [b, c].\Psi[(c, b.2)]$$

MUTBOR-BYE

$$\alpha \mid \mathtt{b} : \&_{\alpha}\mathbf{\mathtt{mut}}\ \mathtt{T} \vdash\ \dashv$$

$$\rightsquigarrow \lambda\Psi, [b].\ b.2 = b.1 \longrightarrow \Psi\ []$$

Adequacy Revisited

Theorem. *(Adequacy). For any* λ_{Rust} *function* f *such that* $\varnothing \mid \varnothing \vdash f \dashv x.x : \mathtt{fn}(\varnothing) \to () \rightsquigarrow \lambda\Psi, [].\ \Psi\ [\lambda\Psi', [].\ \Psi'\ ()]$ *holds, no execution of* f *(with the trivial continuation) ends in a stuck state.*

Our λ_{Rust} contains assertions, theorem implies their validity

Also implies core safety properties like inbounds array accesses

RustHornBelt

Verifying Unsafe Code

Developed **RustHornBelt** with Y. Matsushita and D. Dreyer

PLDI'22 (Distinguished Paper)

Extends **RustBelt** to reason about *functional correctness*

Final Coq proof totals ~19kloc

Can prove safety of unsafe code including `Vec`, `Mutex`, and `Cell`

Applications

CreuSAT: A *performant* formally verified CDCL Sat solver

- Developed and proven by Sarek Skotåm
- ~3kloc specifications, ~1kloc executable
- ~3 mins to check proofs

Sprout: A simple SMT solver, but a good benchmark for Creusot

- Collaboration with M. Bonacina and S. Graham-Lengrand at SRI
- ~2kloc specification, ~1.5kloc executable
- ~1 min to check proofs

Conclusion

Creusot is publicly available today

- Used in collaboration with laboratories all over the world

Key Publications:

- “Creusot: A Foundry for the Deductive Verification of Rust Programs”, ICFEM’22
- “RustHornBelt: A Semantic Foundation for Functional Verification of Rust Programs with Unsafe Code”, PLDI’22
  - Distinguished Paper
- “Specifying and Verifying Higher-Order Rust Iterators”, TACAS’23

Verifying Iterators

An alternative memcpy

```
fn memcpy(tgt: &mut [u8], src: & [u8]) {
    for (t, s) in tgt.iter_mut().zip(src) {
        *t = *s;
    }
}
```

Uses Iterators

An alternative memcpy

```
fn memcpy(tgt: &mut [u8], src: & [u8]) {
    for (t, s) in tgt.iter_mut().zip(src) {
        *t = *s;
    }
}
```

Uses Iterators

An alternative memcpy

```
fn memcpy(tgt: &mut [u8], src: & [u8]) {
    let mut it = tgt.iter_mut().zip(src);
    loop {
        match it.next() {
            Some((t,s)) => { *t = *s }
            None => break
        }
    }
}
```

What are Iterators?

```
trait Iterator {
  type Item;

  fn next(&mut self) -> Option<Self::Item>;
}
```

Rust uses *external iterators* through the `Iterator` trait.

Iterators can be composed and abstracted over

Need generic reasoning principles

Modeling Iterators

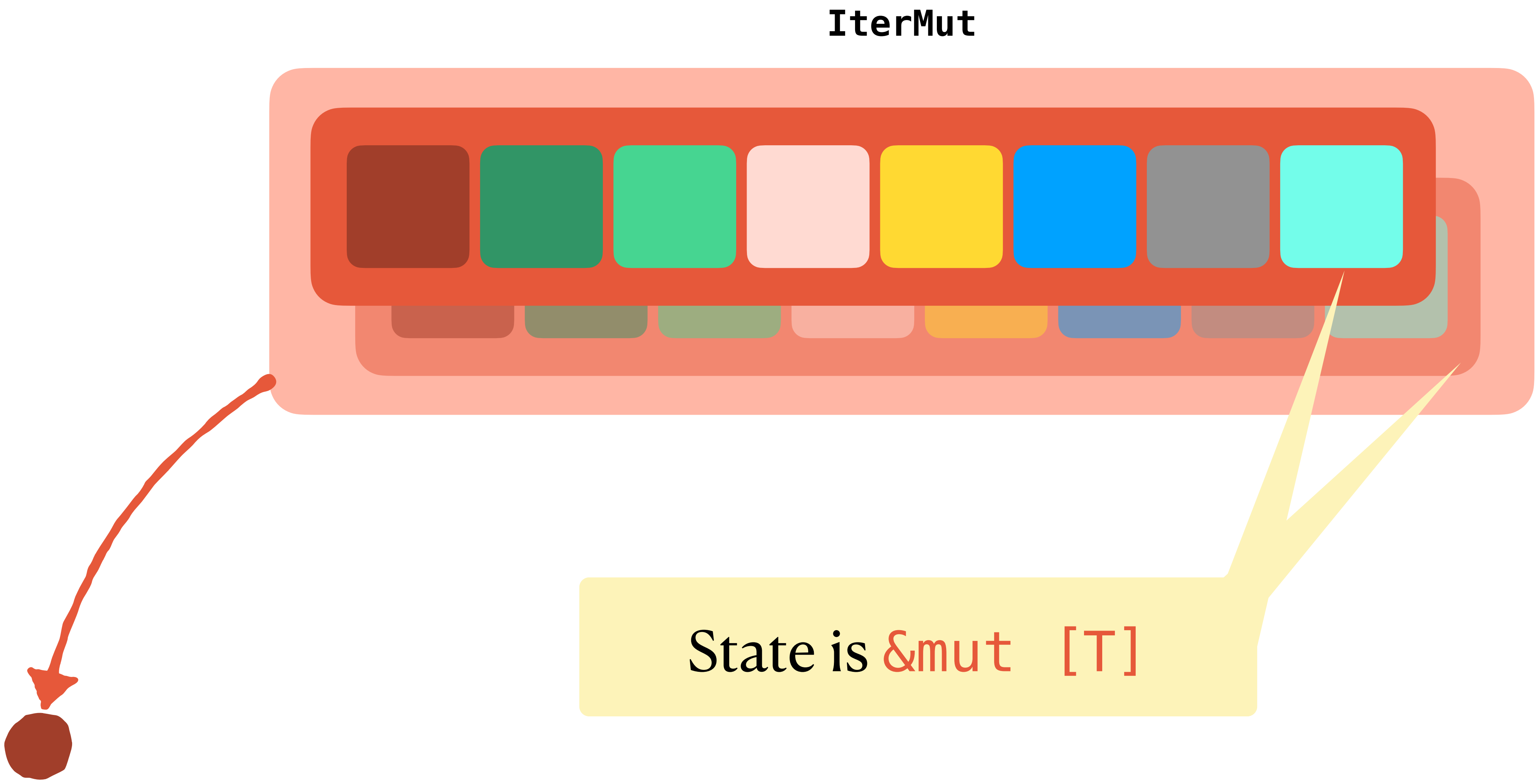
An iterator is a 4-uple $(S, I, \cdot \rightsquigarrow \cdot, C)$:

- A set of states: S
- A set of items: I
- A production relation: $\rightsquigarrow \subseteq S \times I^\star \times S$
- Transitive: $a \overset{s}{\rightsquigarrow} b \land b \overset{t}{\rightsquigarrow} c \to a \overset{s \cdot t}{\rightsquigarrow} c$, reflexive: $a \overset{\epsilon}{\rightsquigarrow} a$
- A set of accepting states: $C \subseteq S$

The `IterMut` Iterator

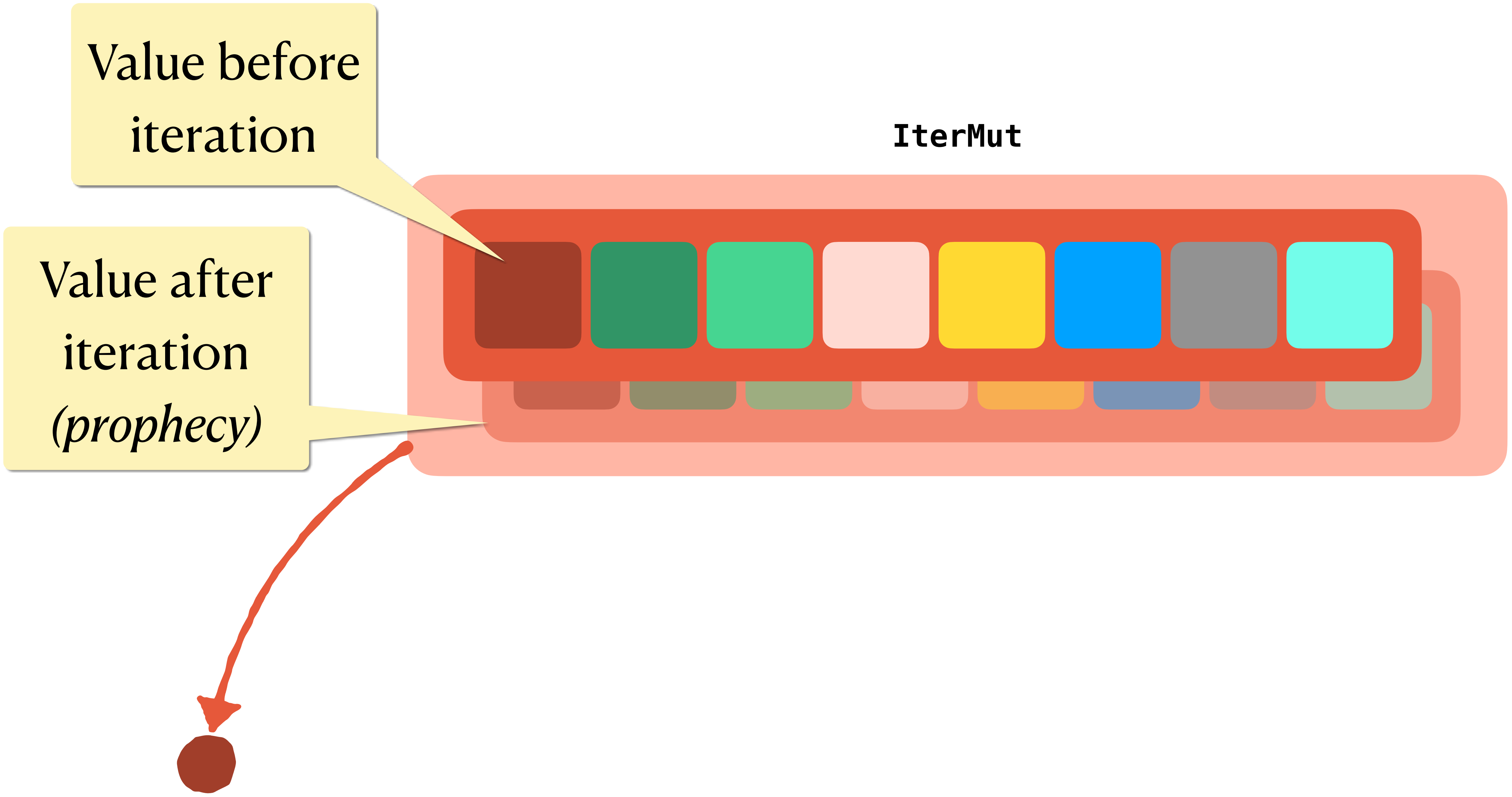
```
struct IterMut<'a, T> { elems: &'a mut [T] }

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<Self::Item> { .. }
}
```

The `IterMut` Iterator

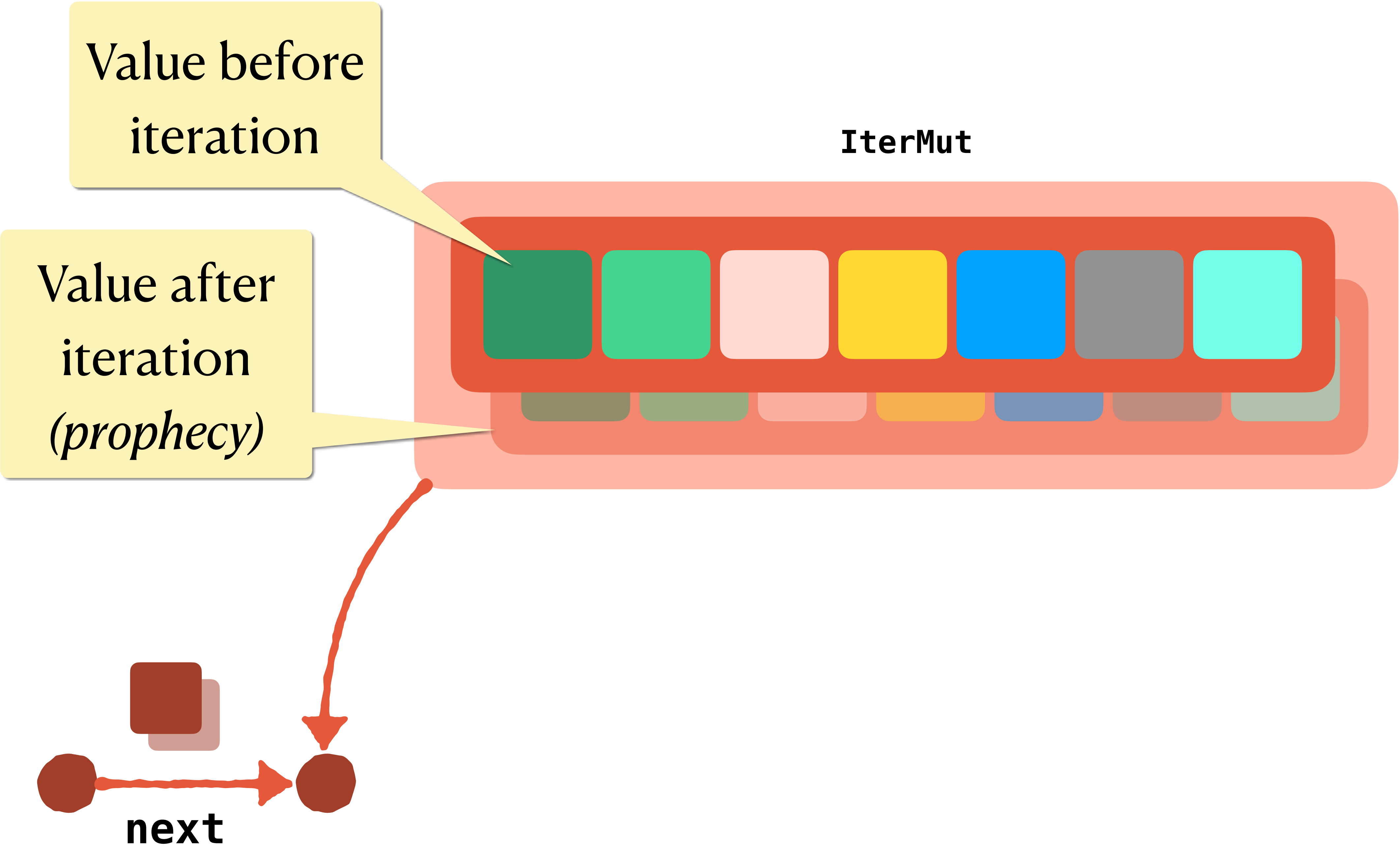


State is `&mut [T]`

The `IterMut` Iterator



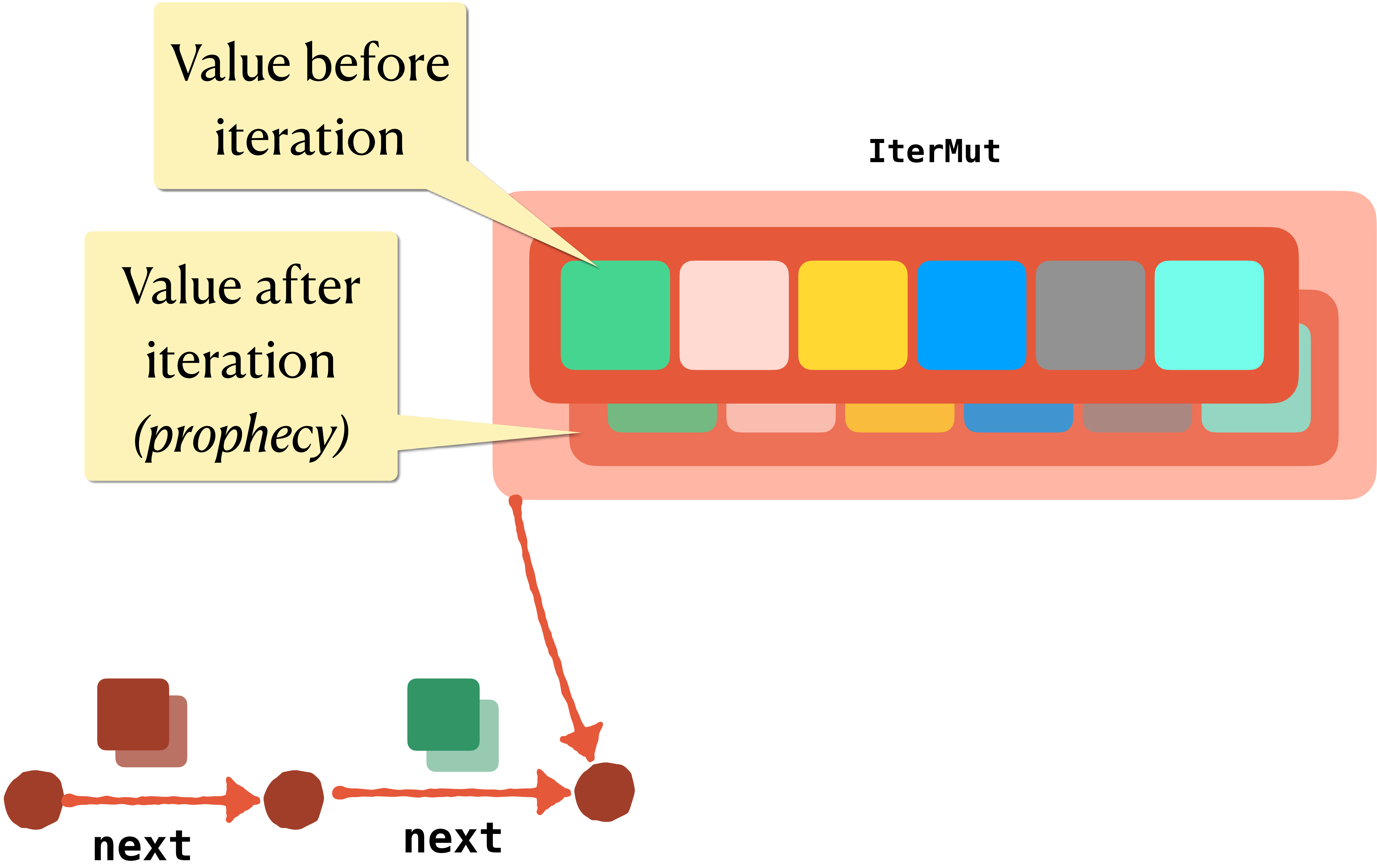
The `IterMut` Iterator



Value before iteration

Value after iteration *(prophecy)*

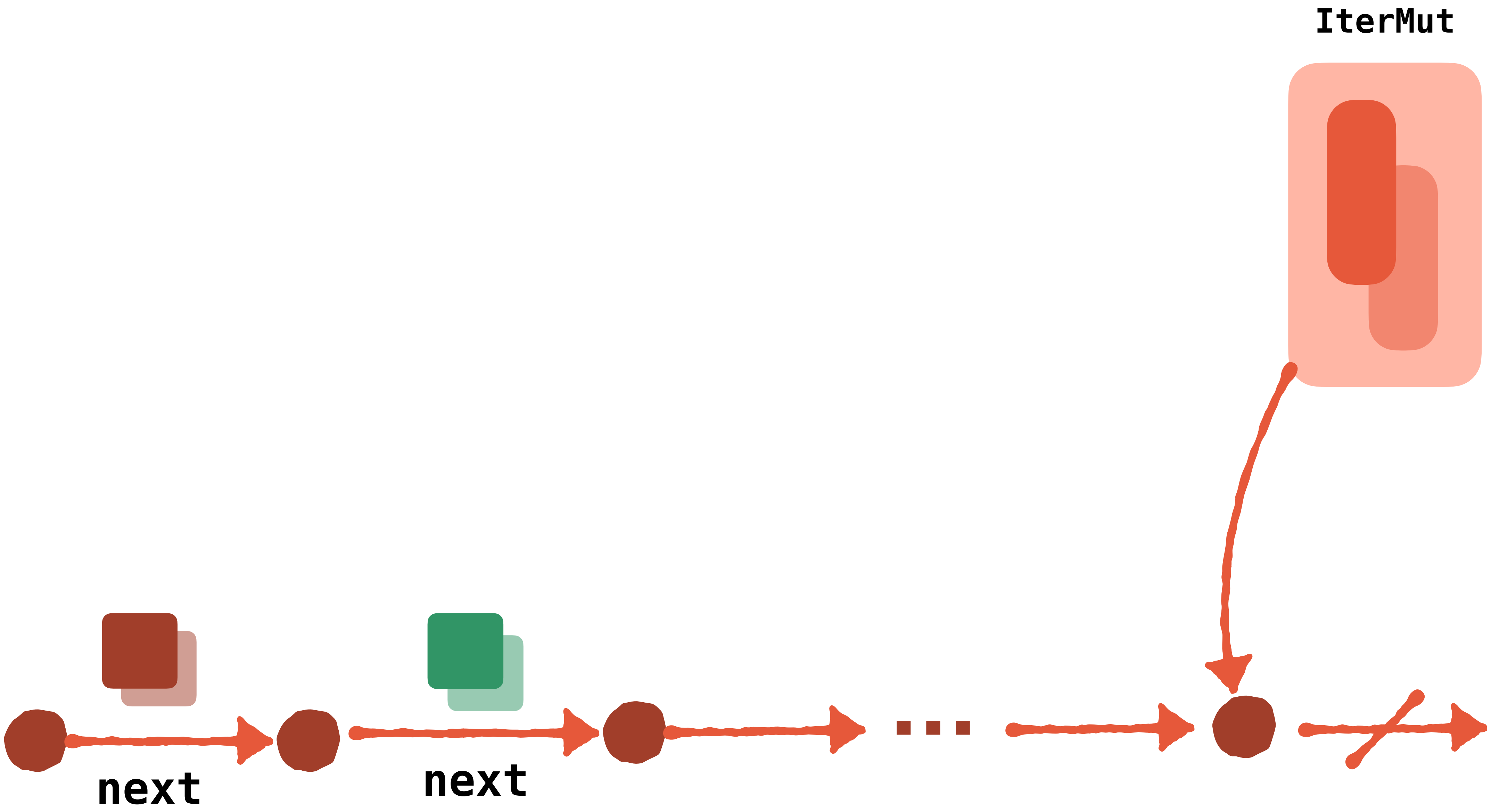
The `IterMut` Iterator



Value before iteration

Value after iteration *(prophecy)*

The `IterMut` Iterator



The `IterMut` Iterator

The transition relation

$$\mathtt{it} \overset{v}{\rightsquigarrow} \mathtt{it}' \triangleq tr(it) = v \cdot tr(it')$$

where

$$tr(s) \triangleq [\texttt{\&mut}\ s[0], \ldots, \texttt{\&mut}\ s[|s| - 1]]$$

The `IterMut` Iterator

Accepting States

$$\text{completed}(s) \triangleq |s| = 0$$

An alternative `memcpy`

With specifications

```
#[requires(tgt.len() == src.len())]
#[ensures(^tgt == *src)]
fn memcpy(tgt: &mut [u8], src: & [u8]) {
  #[invariant(∀ i, 0 ≤ i < produced.len()
     ==> tgt[i] == src[i])]
    for (t, s) in tgt.iter_mut().zip(src) {
       *t = *s;
    }
  }
```

`produced` refers to previous elements of **`for`**-loop

Higher-Order Iteration

Motivating Example

```
fn incr_vec() {
    let v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w : Vec<u32> = v.iter()
        .map(|x| { cnt += 1; *x })
        .collect();

    assert_eq!(w, v);
    assert_eq!(cnt, 4);
}
```

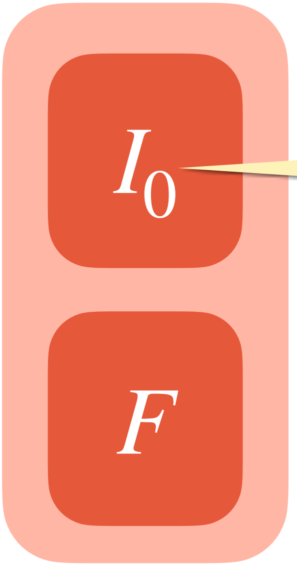
Higher-Order Iteration

Motivating Example

```
fn incr_vec() {
    let v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w : Vec<u32> = v.iter()
        .map(|x| { cnt += 1; *x })
        .collect();

    assert_eq!(w, v);
    assert_eq!(cnt, 4)
}
```

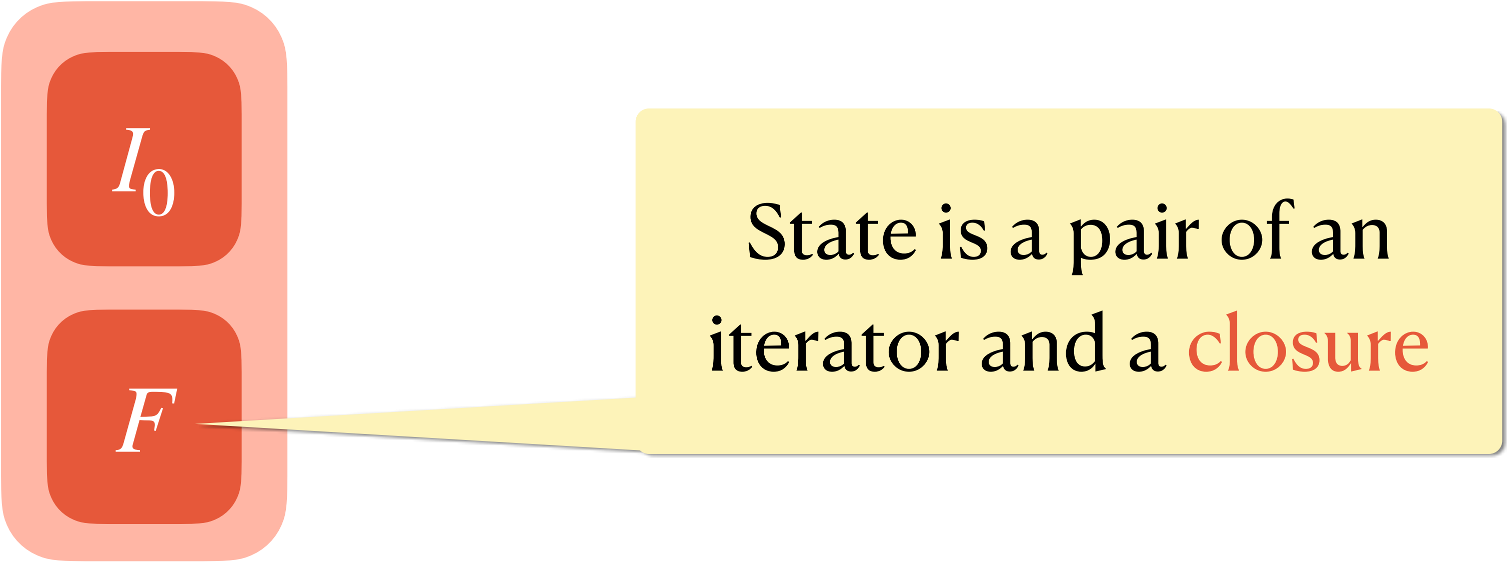
We ignore overflow here

The `Map` Iterator



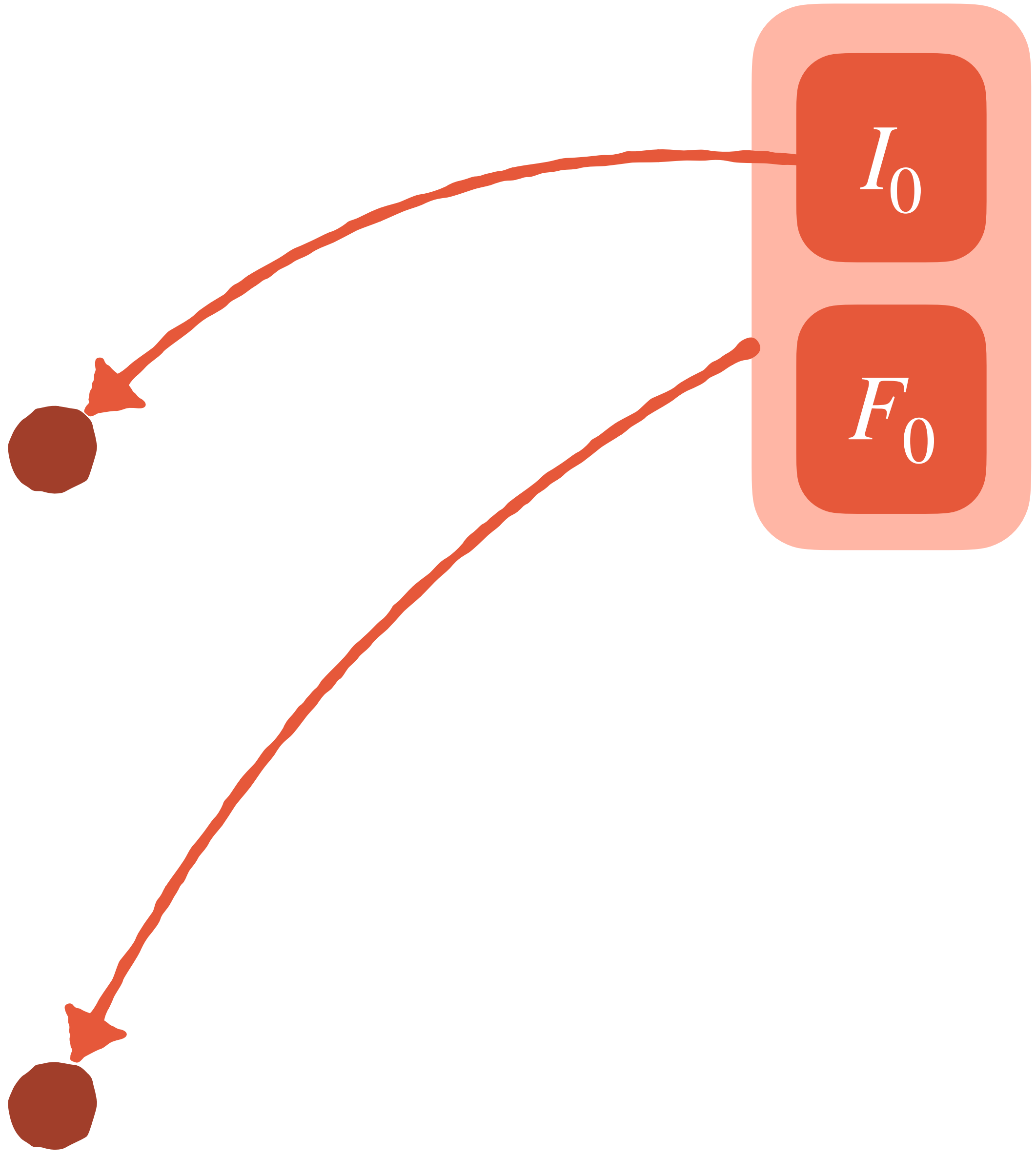
State is a pair of an iterator and a closure

The `Map` Iterator

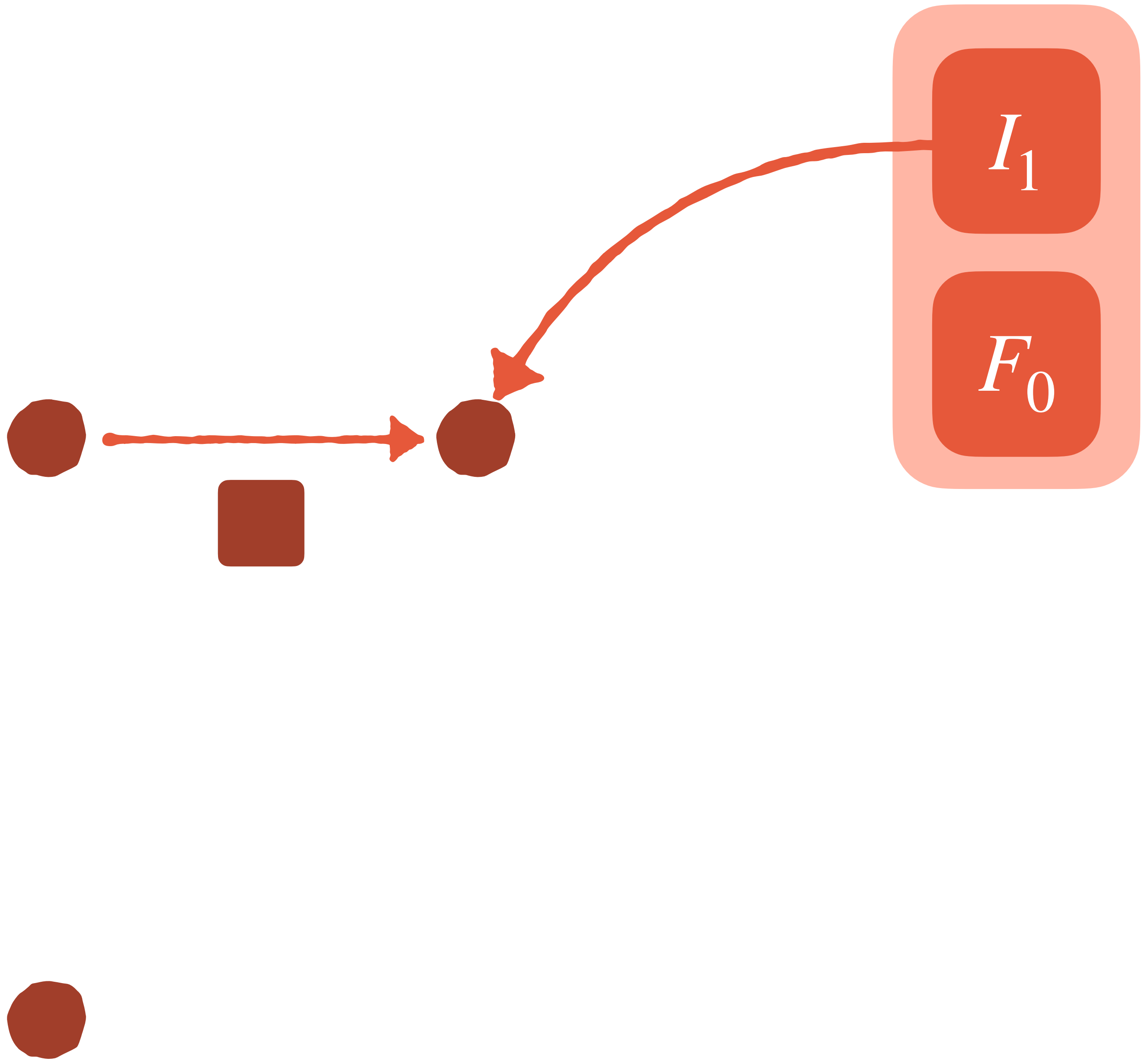


State is a pair of an iterator and a closure

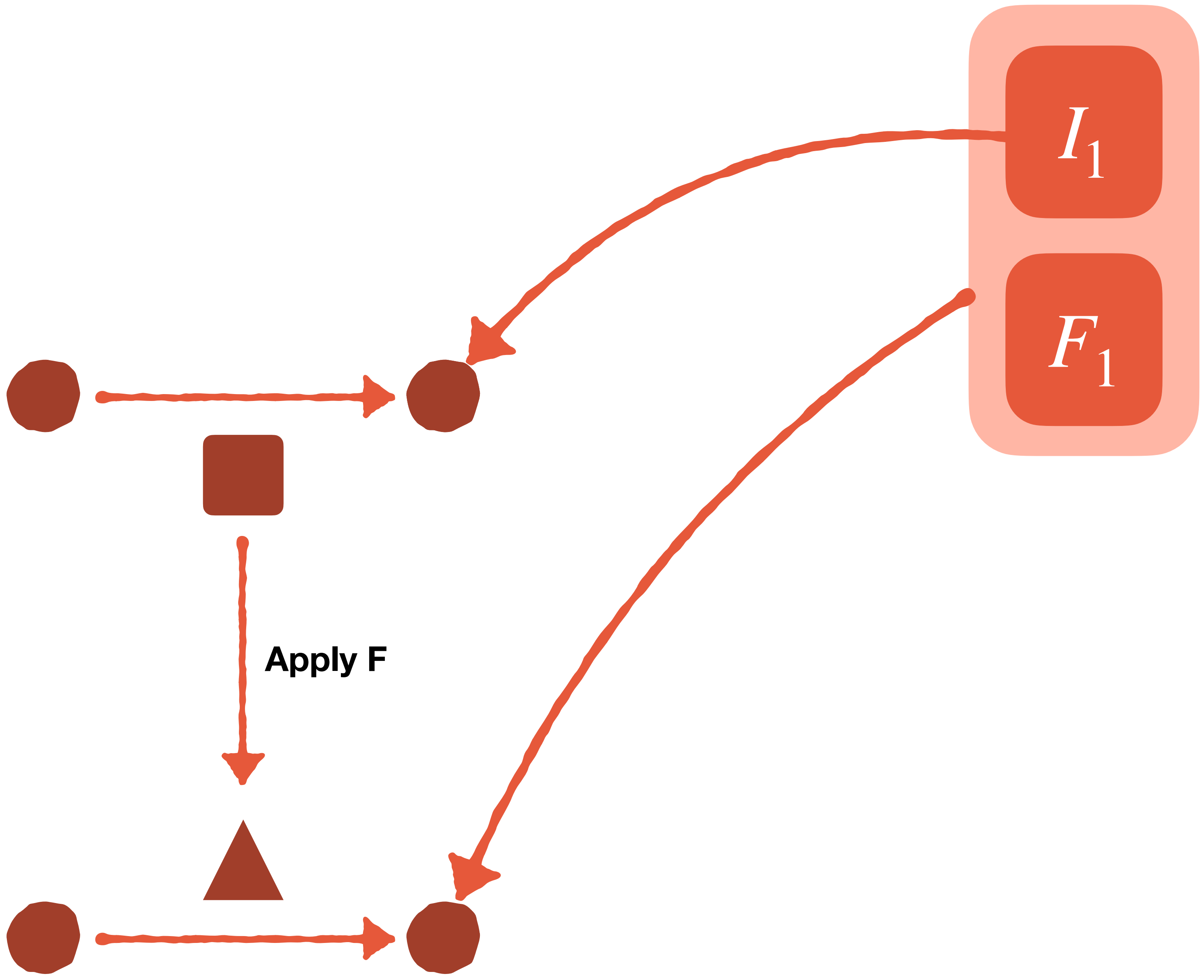
The Map Iterator



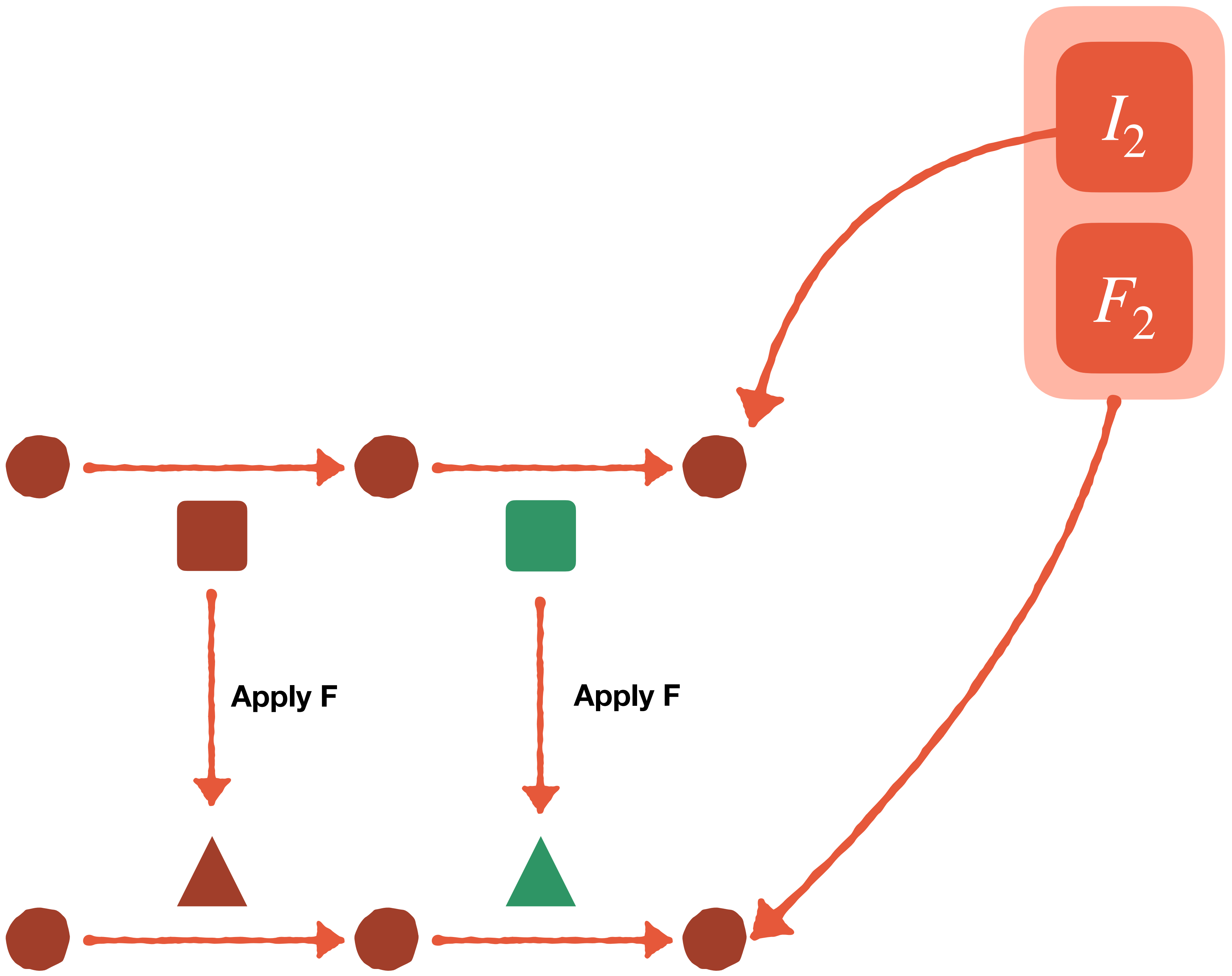
The Map Iterator



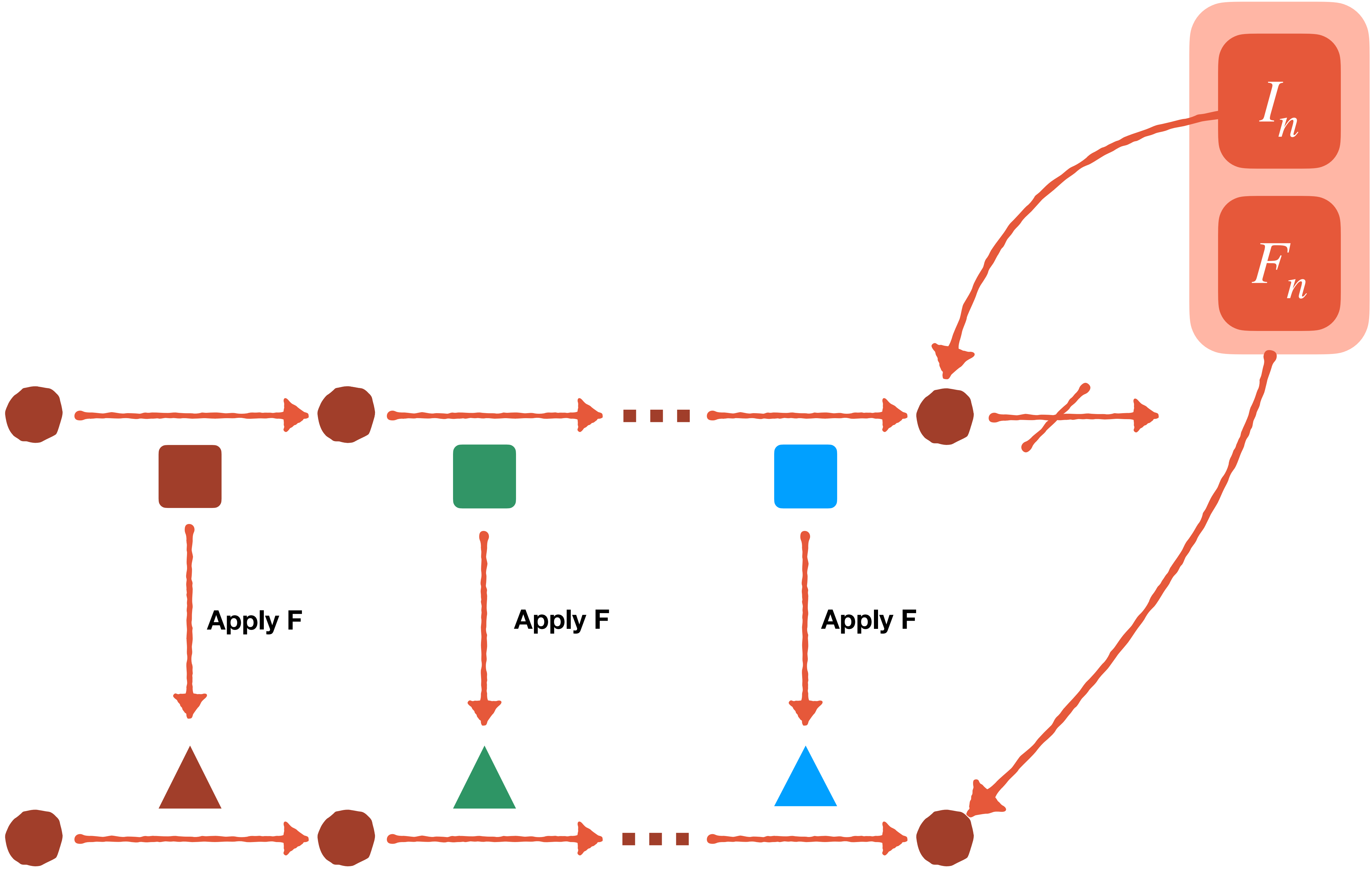
The Map Iterator



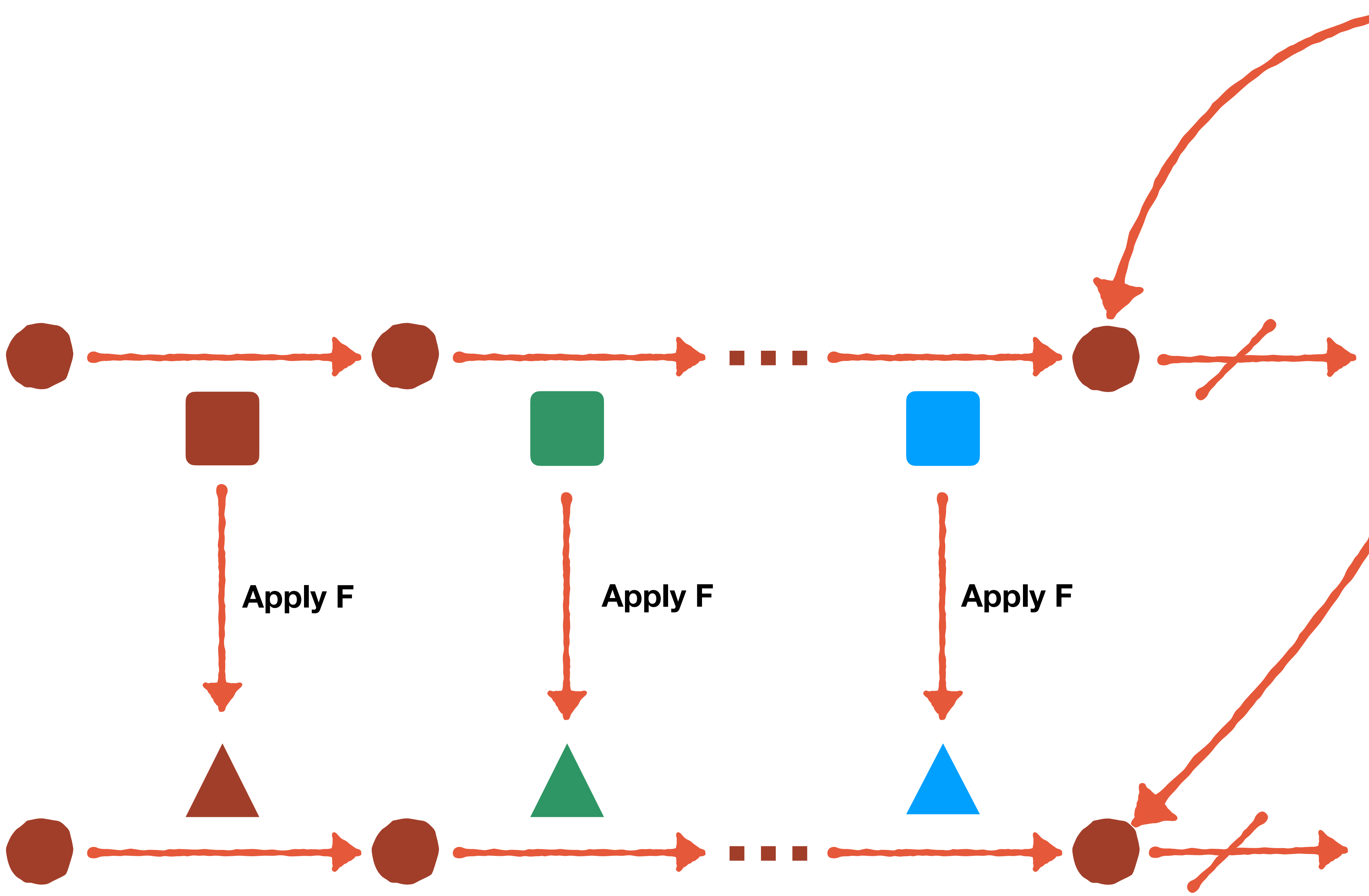
The Map Iterator



The Map Iterator

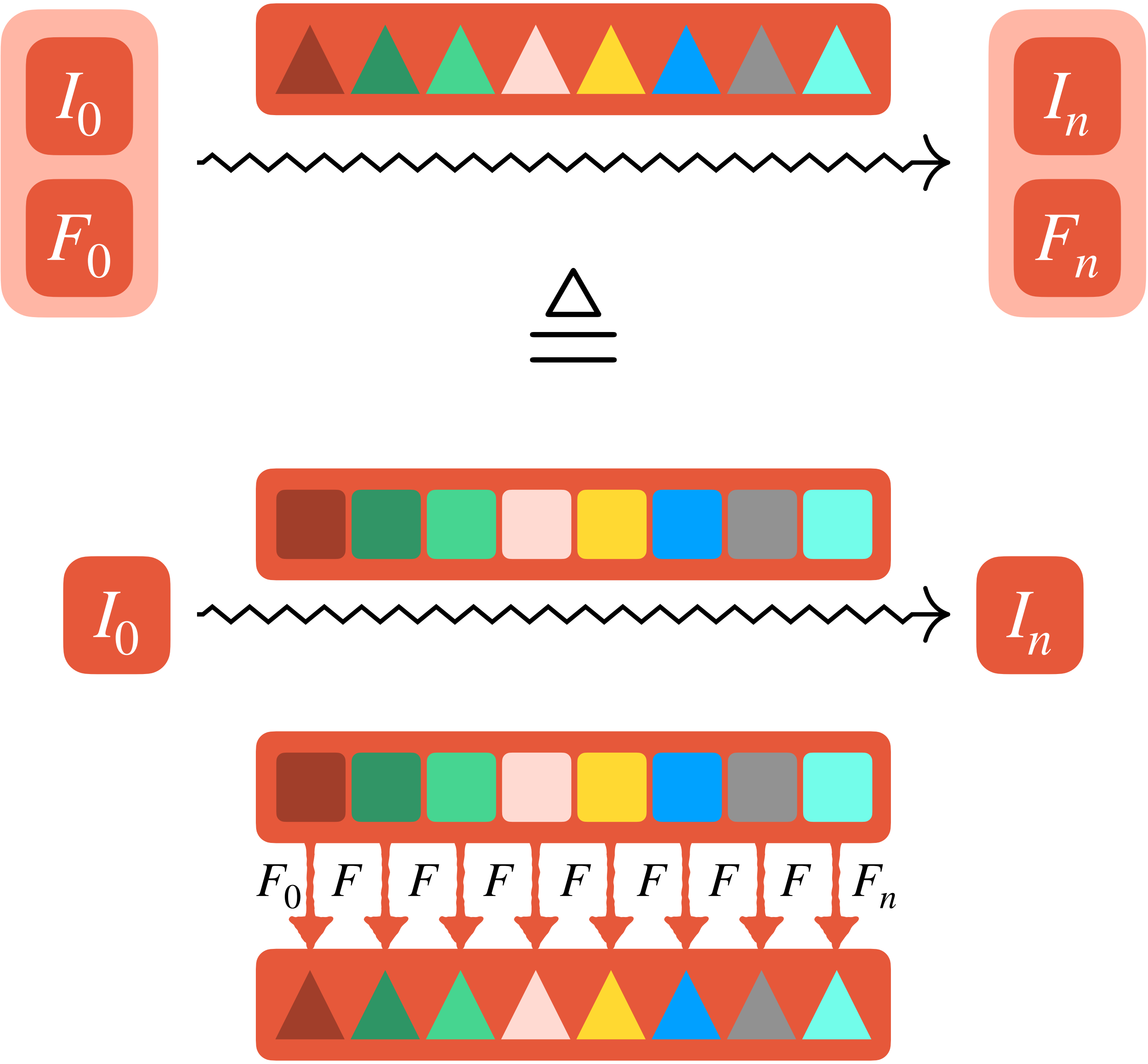


The Map Iterator



The Map Iterator

The produces relation



The Map Iterator

Accepting States

$$\boxed{\begin{matrix}\boxed{I}\\ \boxed{F}\end{matrix}} \in C \Leftrightarrow \boxed{I} \in C$$

The Map Iterator

Side-effects and Preconditions

```
fn incr_vec() {
    let v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w : Vec<u32> = v.iter()
        .map(|x| { cnt += 1; *x })
        .collect();

    assert_eq!(w, v);
    assert_eq!(cnt, 4);
}
```

The Map Iterator

Side-effects and Preconditions

```
fn incr_vec() {
    let v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w : Vec<u32> = v.iter()
        .map(|x| { cnt += 1; *x })
        .collect();

    assert_eq!(w, v);
    assert_eq!(cnt, 4);
}
```

How do we prove this assertion?

The Map Iterator

Side-effects and Preconditions

```
fn incr_vec() {
    let v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w : Vec<u32> = v.iter()
        .map(
            #[ensures(result == *x)]
            |x| { cnt += 1; *x }
        )
        .collect();

    assert_eq!(w, v);
    assert_eq!(cnt, 4);
}
```

Map propagates this through `collect` to `w`

The Map Iterator

Side-effects and Preconditions

```
fn incr_vec() {
    let v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w : Vec<u32> = v.iter()
        .map(
            #[ensures(result == *x)]
            |x| { cnt += 1; *x }
        )
        .collect();

    assert_eq!(w, v);
    assert_eq!(cnt, 4);
}
```

The Map Iterator

Side-effects and Preconditions

```
fn incr_vec() {
    let v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w : Vec<u32> = v.iter()
        .map(
            #[ensures(result == *x)]
            |x| { cnt += 1; *x }
        )
        .collect();

    assert_eq!(w, v);
    assert_eq!(cnt, 4);
}
```

How do we prove *this* assertion?

The Map Iterator

Side-effects and Preconditions

```
fn incr_vec() {
    let v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w : Vec<u32> = v.iter()
        .map(
            #[ensures(result == *x)]
            |x| { cnt += 1; *x }
        )
        .collect();

    assert_eq!(w, v);
    assert_eq!(cnt, 4);
}
```

`cnt` maintains an *invariant* counting the number of iterated elements.

The Map Iterator

Side-effects and Preconditions

```
fn incr_vec() {
    let v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w : Vec<u32> = v.iter()
        .map_hist(
            #[requires(cnt == prod.len())]
            #[ensures(cnt == prod.len() + 1)]
            #[ensures(result == *x)]
            |x, prod| { cnt += 1; *x }
        )
        .collect();
    assert_eq!(w, v);
    assert_eq!(cnt, 4);
}
```

The Map Iterator

Side-effects and Preconditions

```
fn incr_vec() {
    let v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w : Vec<u32> = v.iter()
        .map_hist(
            #[requires(cnt == prod.len())]
            #[ensures(cnt == prod.len() + 1)]
            #[ensures(result == *x)
            |x, prod| { cnt += 1; *x }
        )
        .collect();
    assert_eq!(w, v);
    assert_eq!(cnt, 4);
}
```

Ghost access to past elements

Recap

We showed a specification for iterators which leverages Rust types

- Can handle mutability and side-effects (**`IterMut`**)
- Can handle higher-order with mutable captures in FOL (**`Map`**)

We have proven **15+** different iterators and clients

- `Map`, `IterMut`, `Zip`, `Enumerate`, `collect`, …