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?

Low level code
- C/C++
- VeriFast
- Frama-C

High level code
- Lean
- Coq
- Why3
- Isabelle

Small scale reasoning
- Functional Programming

Large scale reasoning

???
What is the problem with verification?

```c
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

```c
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**

```c
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**

```c
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**

Pointers could be **uninitialized**

We want to prove that **size** bytes of **src** are copied to **tgt**
What is the **problem** with verification?

**Verified using VeriFast**

```c
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

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

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

Box

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

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

Boxes are erased!

Consequence of uniqueness
fn incr_immut(x: &u64, y: &u64)
    -> u64 {
    *x + *y
}
Encoding Rust in ML

Immutable References

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

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

Also erased!

No mutation = No problems
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
def incr_mut(x: &mut u64, y: u64) {
    \*x += y
}
```

```rust
fn main() {
    let mut x = 0;
    incr_mut(&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 $\alpha$
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 $\alpha$. 

Prophecies

Synchronizing lender and borrower

\[ &\text{mut}_\alpha \ a \]
Prophecies
Synchronizing lender and borrower

Models mutable borrows as pair of current and final values.

We prophesize the final value, which the lender recovers.

Depends on uniqueness and lifetimes of mutable borrows $\alpha$.
Prophecies
Synchronizing lender and borrower

Models mutable borrows as pair of \textit{current} and \textit{final} values

We prophetize the final value, which the lender recovers.

Depends on \textit{uniqueness} and \textit{lifetimes} of mutable borrows

\[ \alpha \]

\(a\) is inaccessible for the duration of \(\alpha\)
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.

\[ \alpha \]

\[ \&\text{mut}_\alpha a \]

\( a \) is inaccessible for the duration of \( \alpha \)
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 $\alpha$.

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 $\alpha$.

Can’t model the second point; instead propheticize 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**

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

**Resolution**

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

```rust
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())]

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())]
#[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

```rust
#[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];
        i += 1;
    }
}
```

The `^` operator accesses the final value of a borrow.
Beyond the status quo

Verifying with Creusot

```rust
#[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
A formal model of the core Rust type system.

Uses a language called $\lambda_{Rust}$ 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 $\lambda_{Rust}$ function $f$ such that

$\emptyset \mid \emptyset \vdash f \rightarrow x.x : fn(\emptyset) \rightarrow ()$ 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

\[ E; L \mid T \vdash I \vdash T' \leadsto \Phi \]
RustHornBelt
Type-Spec Judgements
RustHornBelt
Type-Spec Judgements

E; L | T ⊢ I ⊸ T’ ⟷ Φ

Instruction
RustHornBelt
Type-Spec Judgements

\[ E; L \vdash T \vdash I \vdash T' \sim \Phi \]

Output Contexts
E; L | T ⊢ I ⊢ T' ~/Φ

Specification
RustHornBelt

Interpretation of judgments

\[
\begin{align*}
L | T \vdash I \vdash a. & \quad T' \rightsquigarrow \Phi \downarrow \\
\forall \hat{\Psi}. & \{ \exists \hat{a}. [L] \ast [T](\hat{a}) \\
& \ast \langle \lambda \pi. \Phi (\hat{\Psi} \pi) (\hat{a} \pi) \rangle \} \\
I \{ & \quad r. \exists \hat{b}. [L] \ast [T'](\hat{b}) \\
& \ast \langle \lambda \pi. (\hat{\Psi} \pi) (\hat{b} \pi) \rangle \}
\end{align*}
\]
RustHornBelt

Interpretation of judgments

\[
\begin{align*}
&\left[ L \mid T \vdash I \vdash a. \ T' \rightsquigarrow \Phi \right] \triangleq \\
&\forall \hat{\Psi}. \left\{ \exists \hat{a}. \left[ L \right] \ast \left[ T \right](\hat{a}) \right. \\
&\quad \ast \left\{ \lambda \pi. \Phi (\hat{\Psi} \pi)(\hat{a} \pi) \right\} \\
&I\left\{ r. \exists \hat{b}. \left[ L \right] \ast \left[ T' \right](\hat{b}) \\
&\quad \ast \left\{ \lambda \pi. (\hat{\Psi} \pi)(\hat{b} \pi) \right\}
\end{align*}
\]

An instruction is well-typed if:
An instruction is well-typed if:

\[
\forall \hat{\Psi}. \left\{ \exists \hat{a}. \left[ L \right] * \left[ T \right](\hat{a}) \right. \\
\left. * \left\langle \lambda \pi. \Phi \left( \hat{\Psi} \pi \right) (\hat{a} \pi) \right\rangle \right\}
\]

\[
I \left\{ \left. r. \exists \hat{b}. \left[ L \right] * \left[ T' \right](\hat{b}) \right. \\
\left. * \left\langle \lambda \pi. \left( \hat{\Psi} \pi \right) (\hat{b} \pi) \right\rangle \right\}
\]
RustHornBelt

Interpretation of judgments

\[
\begin{align*}
\llbracket L \mid T \vdash I \downarrow a. \quad T' \rightsquigarrow \Phi \rrbracket & \triangleq \\
\forall \hat{\Psi}. \{ \exists \hat{a}. [L] \ast [T](\hat{a}) & \ast \langle \lambda \pi. \Phi (\hat{\Psi} \pi) (\hat{a} \pi) \rangle \} \\
I \{ \ r. \ \exists \hat{b}. [L] \ast [T'](\hat{b}) & \ast \langle \lambda \pi. (\hat{\Psi} \pi) (\hat{b} \pi) \rangle \}
\end{align*}
\]

An instruction is well-typed if:

Given resources for \( L \) and \( T \)...

...and a precondition from \( \Phi \)...

37
**RustHornBelt**

Interpretation of judgments

\[
\begin{align*}
\llbracket L \mid T \vdash I \vdash a. \ T' \rightsquigarrow \Phi \rrbracket & \triangleq \\
\forall \hat{\Psi}. \ \{ \exists \hat{a}. \ [L] \ast [T](\hat{a}) \\
& \ast \langle \lambda \pi. \Phi (\hat{\psi} \pi)(\hat{a} \pi) \rangle \} \\
I\{ \ r. \ \exists \hat{b}. \ [L] \ast [T'](\hat{b}) \\
& \ast \langle \lambda \pi. (\hat{\psi} \pi)(\hat{b} \pi) \rangle \}
\end{align*}
\]

An instruction is well-typed if:

Given resources for \( L \) and \( T \)...

...and a precondition from \( \Phi \)...

...executing \( I \) gives back \( L \) and \( T' \)...
An instruction is well-typed if:

\[
\begin{align*}
\left[ L \mid T \vdash I \vdash a. \; T' \leadsto \Phi \right] \triangleq \\
\forall \hat{\Psi}. \left\{ \exists \hat{a}. [L] * [T](\hat{a}) \right. \\
\left. \quad * \langle \lambda \pi. \Phi (\hat{\Psi} \pi) (\hat{a} \pi) \rangle \right\} \\
I \{ \ r. \ \exists \hat{b}. [L] * [T'](\hat{b}) \right. \\
\left. \quad * \langle \lambda \pi. (\hat{\Psi} \pi) (\hat{b} \pi) \rangle \right\}
\end{align*}
\]

Given resources for \(L\) and \(T\)... ...and a precondition from \(\Phi\)... ...executing \(I\) gives back \(L\) and \(T'\)... ...and postcondition \(\Psi\)
RustHornBelt

Example Judgements

\textbf{MUTBOR-BOR}

\[
\alpha \mid a : \text{own} \ T \vdash \& \text{mut} \star a \vdash b. \ a : \spadesuit \alpha \text{own} \ T, b : \& \alpha \text{mut} \ T
\]

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

\textbf{MUTBOR-WRITE}

\[
\alpha \mid b : \& \alpha \text{mut} \ T, c : T \vdash \star b = c \vdash b. \ b : \& \alpha \text{mut} \ T
\]

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

\textbf{MUTBOR-BYE}

\[
\alpha \mid b : \& \alpha \text{mut} \ T \vdash \vdash
\]

\[
\leadsto \lambda \Psi, [b]. \ b.2 = b.1 \rightarrow \Psi []
\]
Adequacy Revisited

**Theorem.** (Adequacy). For any \( \lambda_{\text{Rust}} \) function \( f \) such that
\[
\emptyset \vdash f \vdash x. x : \text{fn}(\emptyset) \rightarrow () \wedge \lambda \Psi, [\cdot]. \Psi [\lambda \Psi', [\cdot]. \Psi' (\cdot)] \text{ holds},
\]
o no execution of \( f \) (with the trivial continuation) ends in a stuck state.

Our \( \lambda_{\text{Rust}} \) contains assertions, theorem implies their validity

Also implies core safety properties like inbounds array accesses
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
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
A different memcpy function:

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

Uses Iterators
An alternative memcpy

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

Uses Iterators
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
        }
    }
}
Rust uses external iterators through the Iterator trait.

Iterators can be composed and abstracted over

Need generic reasoning principles
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: $\rightsquigarrow \subseteq S \times I^* \times S$
- Transitive: $a \overset{s}{\rightsquigarrow} b \land b \overset{t}{\rightsquigarrow} c \rightarrow a \overset{s\cdot t}{\rightsquigarrow} c$, reflexive: $a \overset{e}{\rightsquigarrow} a$
- A set of accepting states: $C \subseteq S$
The **IterMut** Iterator

```rust
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

- Value before iteration
- Value after iteration *(prophecy)*
The **IterMut** Iterator

Value before iteration

Value after iteration *(prophecy)*

`next`
The **IterMut** Iterator

- Value before iteration
- Value after iteration (*prophecy*)

IterMut

next

next
The **IterMut** Iterator

![Diagram showing the IterMut Iterator with 'next' and 'IterMut' nodes connected by arrows.]

next

next

...
The **IterMut** Iterator

The transition relation

\[
\text{it} \overset{\triangleright}{\rightsquigarrow} \text{it'} \overset{\trianglelefteq}{=} \text{tr}(\text{it}) = v \cdot \text{tr}(\text{it'})
\]

where

\[
\text{tr}(s) \overset{\triangleleft}{=} [\text{&mut } s[0], \ldots, \text{&mut } s[|s| - 1]]
\]
The `IterMut` Iterator

Accepting States

completed(s) ≜ |s| = 0
An alternative `memcpy`

With specifications

```rust
#[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
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);
}
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.

$I_0$

$F$
The Map Iterator

$I_0$

$F_0$
The Map Iterator
The Map Iterator

Apply $F$
The Map Iterator

Apply F

\[ I_n \]

\[ F_n \]
The Map Iterator

Apply $F$

$F_n$

$I_n$
The Map Iterator

The produces relation

$I_0$ $F_0$ $I_n$ $F_n$

$F_0$ $F$ $F$ $F$ $F$ $F$ $F$ $F$ $F$ $F_n$

$\triangle$
The Map Iterator

Accepting States

$I \in C \iff I \in C$
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);
}
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

```rust
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);
}
```

Map propagates this through `collect` to `w`
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
The Map Iterator
Side-effects and Preconditions

```rust
def 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

```rust
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.
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);
}
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, ...