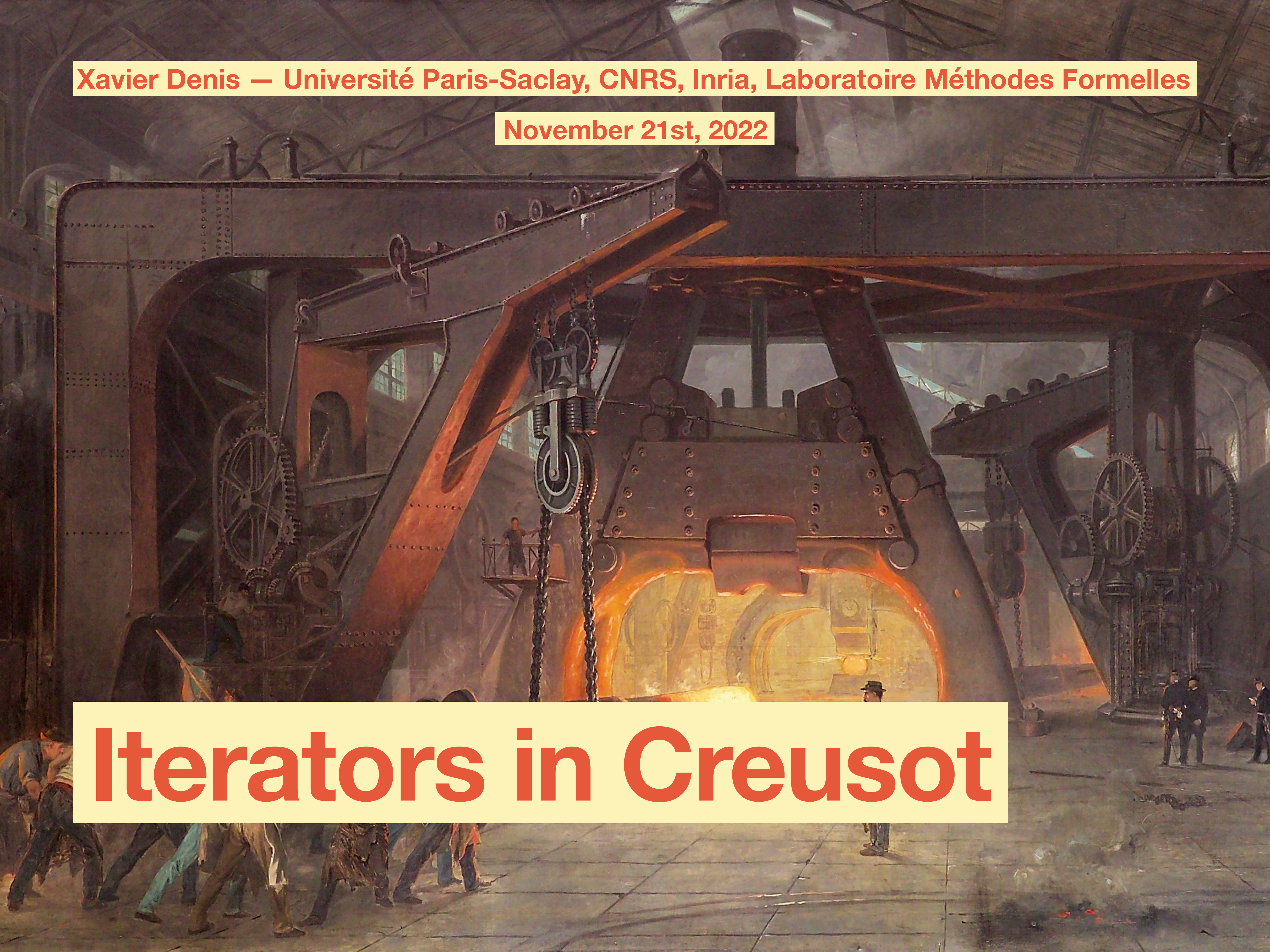


Xavier Denis — Université Paris-Saclay, CNRS, Inria, Laboratoire Méthodes Formelles

November 21st, 2022

Iterators in Creusot

The *pointer problem*

```
void fn use_swap(int* a, int* b, int* c) {
  swap(a, b);
  // What is c here?
}
```

The *pointer problem*

```
void fn use_swap(int* a, int* b, int* c) {
  swap(a, b);
  // What is c here?
}
```

Mutated via global state

The *pointer problem*

May alias

```
void fn use_swap(int* a, int* b, int* c) {
  swap(a, b);
  // What is c here?
}
```

Mutated via global state

The *pointer problem*

May alias

```
void fn use_swap(int* a, int* b, int* c) {
  swap(a, b);
  // What is c here?
}
```

Mutated via global state

Use ***separation logic***?

The *pointer problem*

May alias

```
void fn use_swap(int* a, int* b, int* c) {
  swap(a, b);
  // What is c here?
}
```

Mutated via global state

Use ***separation logic***?

Mixes *memory safety* proof with *functional* proof

The *pointer* problem

May alias

```
void fn use_swap(int* a, int* b, int* c) {
  swap(a, b);
  // What is c here?
}
```

Mutated via global state

Use ***separation logic***?

Mixes *memory safety* proof with *functional* proof

Poor automation, complex logic

The *pointer problem*

May alias

```
void fn use_swap(int* a, int* b, int* c) {
  swap(a, b);
  // What is c here?
}
```

Mutated via global state

Use ***separation logic***?

Mixes *memory safety* proof with *functional* proof

Poor automation, complex logic

To do better we need a new language..

Instead, use Rust

```
fn use_swap(a: &mut u32, b: &mut u32, c: &mut u32) {
  swap(a, b);
  // c is unchanged here
}
```

Instead, use Rust

```
fn use_swap(a: &mut u32, b: &mut u32, c: &mut u32) {
  swap(a, b);
  // c is unchanged here
}
```

Mutability XOR Aliasing: mutable borrows are unique

Ownership typing statically guarantees memory safety

Instead, use Rust

```
fn use_swap(a: &mut u32, b: &mut u32, c: &mut u32) {
  swap(a, b);
  // c is unchanged here
}
```

Mutability XOR Aliasing: mutable borrows are unique

Ownership typing statically guarantees memory safety

How to verify? Separation logic?

Instead, use Rust

```
fn use_swap(a: &mut u32, b: &mut u32, c: &mut u32) {
  swap(a, b);
  // c is unchanged here
}
```

Mutability XOR Aliasing: mutable borrows are unique

Ownership typing statically guarantees memory safety

How to verify? Separation logic?

No! Why prove memory safety twice?

Instead, use Rust

```
fn use_swap(a: &mut u32, b: &mut u32, c: &mut u32) {
  swap(a, b);
  // c is unchanged here
}
```

Mutability XOR Aliasing: mutable borrows are unique

Ownership typing statically guarantees memory safety

How to verify? Separation logic?

No! Why prove memory safety twice?

Today's objective: Verifying Iterators

Or how to write `Vec::len` in O(n) time

```
fn counter() {
    let mut v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w = v
        .iter_mut()
        .map(|x| { cnt += 1; a + 1 })
        .collect();

    assert_eq!(cnt, 4);

    for i in 0..4 {
        assert_eq!(w[i], v[i] + 1);
    }
}
```

Today’s objective: Verifying Iterators

Or how to write `Vec::len` in O(n) time

```
fn counter() {
    let mut v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w = v
        .iter_mut()
        .map(|x| { cnt += 1; a + 1 })
        .collect();

    assert_eq!(cnt, 4);

    for i in 0..4 {
        assert_eq!(w[i], v[i] + 1);
    }
}
```

Mutable Iterators

Today's objective: Verifying Iterators

Or how to write `Vec::len` in O(n) time

```
fn counter() {
    let mut v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w = v
        .iter_mut()
        .map(|x| { cnt += 1; a + 1 })
        .collect();

    assert_eq!(cnt, 4);

    for i in 0..4 {
        assert_eq!(w[i], v[i] + 1);
    }
}
```

Mutable Iterators → `.iter_mut()`

Higher-Order Iterators → `.map(...)`

Today's objective: Verifying Iterators

Or how to write `Vec::len` in O(n) time

```
fn counter() {
    let mut v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w = v
        .iter_mut()
        .map(|x| { cnt += 1; a + 1 })
        .collect();

    assert_eq!(cnt, 4);

    for i in 0..4 {
        assert_eq!(w[i], v[i] + 1);
    }
}
```

Side-Effects

Mutable Iterators

Higher-Order Iterators

Today's objective: Verifying Iterators

Or how to write `Vec::len` in O(n) time

```
fn counter() {
    let mut v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w = v
        .iter_mut()
        .map(|x| { cnt += 1; a + 1 })
        .collect();

    assert_eq!(cnt, 4);

    for i in 0..4 {
        assert_eq!(w[i], v[i] + 1);
    }
}
```

Mutable Iterators

Higher-Order Iterators

Side-Effects

Traversal and collection creation

Today's objective: Verifying Iterators

Or how to write `Vec::len` in O(n) time

```
fn counter() {
    let mut v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w = v
        .iter_mut()
        .map(|x| { cnt += 1; a + 1 })
        .collect();

    assert_eq!(cnt, 4);

    for i in 0..4 {
        assert_eq!(w[i], v[i] + 1);
    }
}
```

Side-Effects

Mutable Iterators

Traversal and collection creation

Higher-Order Iterators

`for`-loops

What are Iterators?

- Rust for-loops are powered using *iterators*.
- Iterators can be created using combinators (map, filter, chain).
- Can be expressed as the following trait:

```
trait Iterator : Sized {
    type Item;

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

- This captures a wide variety of iteration: non-deterministic, effectful, and non-terminating

Challenges

Specifying Iterators

Challenges

Specifying Iterators

- **Key Problem 1:** A specification scheme for iterators
  - Composable & Ergonomic
  - Supports non-determinism and interruptible iteration
  - Supports side-effects and higher-order constructs (map)

Challenges

Specifying Iterators

- **Key Problem 1:** A specification scheme for iterators
  - Composable & Ergonomic
  - Supports non-determinism and interruptible iteration
  - Supports side-effects and higher-order constructs (map)
- **Key Problem 2:** How do we enable users to write expressive invariants which focus on the core of their problem.

Solution

Using **Creusot** we developed a framework to reason about Iterators and their clients.

- **Problem 1:** We view iterators as *state machines* described using
  - $produced$, a transition relation used to describe **next**
  - $completed$, captures the *final* states of the iterator
- **Problem 2:** We support for-loops through a new form of *invariant* which accesses past values of an iterator.
  - Provides invariants for free via iterator.

In this talk

Overview

I. Introduction to **Creusot**
   I. Mutable Value Semantics of Rust
   II. Prophecies
II. Specifying Iterators
   I. General schema
   II. `IterMut`
III. `for`-loops

Introduction to Creusot

Creusot in a nutshell

A highly-automated verification platform for Rust

- Allows user to annotate their programs with specifications

```
#[ensures(^x == * x + 1)]
fn incr(x: &mut u32) {
    *x += 1;
}
```

- Specifications are then checked using automated provers (SMT)
- Provides many features to help write specifications and do proofs

Creusot in a nutshell

How does it work?

- Creusot views Rust programs as *pure, functional* programs
  - Enabled by the *mutable value semantics* of Rust
  - Metatheory formalized in **RustHornBelt**[1]
- Avoids *separation logic* and instead uses *first-order logic*
  - Fully handles mutable borrows: even nested in structures
- By using FOL, get much stronger automation

[1] Matsushita, Denis, Jourdan, Dreyer “RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code”, PLDI’22

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 main () {
  let mut a = 0;
  let x = &mut a;
  let y = &mut 5;
  *x += *y;
  drop(x);
  assert_eq!(a, 5);
}
```

?

Challenge: Synchronizing dataflow between lender and borrower.

Encoding Rust in ML

Mutable References?

```rust
fn main () {
  let mut a = 0;
  let x = &mut a;
  let y = &mut 5;
  *x += *y;
  drop(x);
  assert_eq!(a, 5);
}
```

?

Challenge: Synchronizing dataflow between lender and borrower.

Encoding Rust in ML

Mutable References?

```
fn main () {
  let mut a = 0;
  let x = &mut a;
  let y = &mut 5;
  *x += *y;
  drop(x);
  assert_eq!(a, 5);
}
```

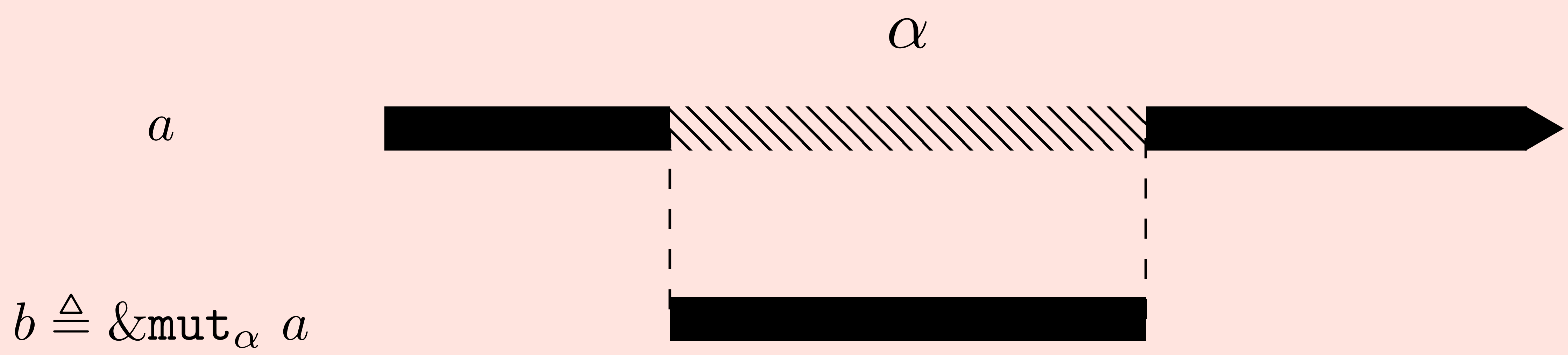
?

Challenge: Synchronizing dataflow between lender and borrower. Solution? *Prophecies*

Prophecies

Synchronizing lender and borrower

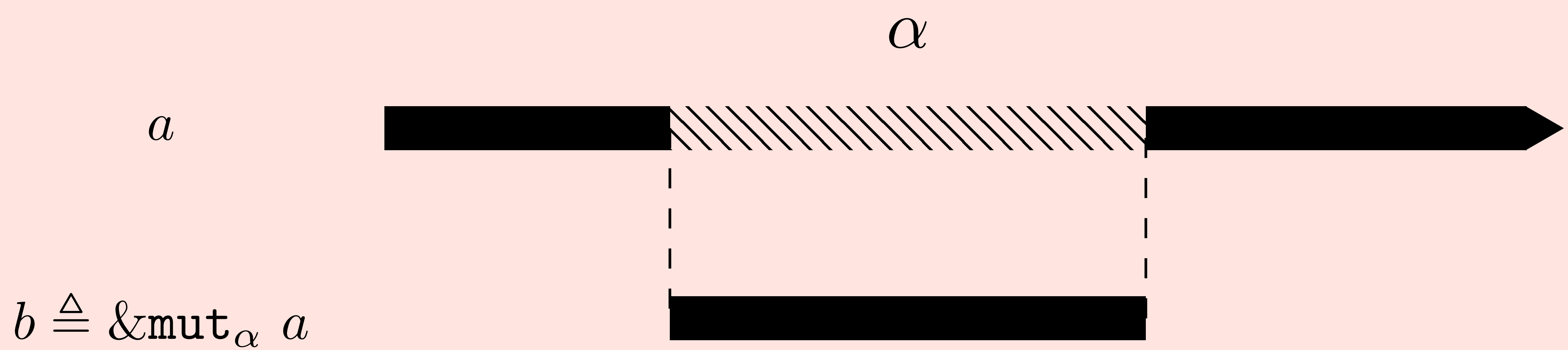
- **Idea:** Model mutable borrows as pair of **current** and **final** values
- We prophetize the final value, which the lender recovers.
- Depends on **uniqueness** and **lifetimes** of mutable borrows



Prophecies

Synchronizing lender and borrower

- **Idea:** Model 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

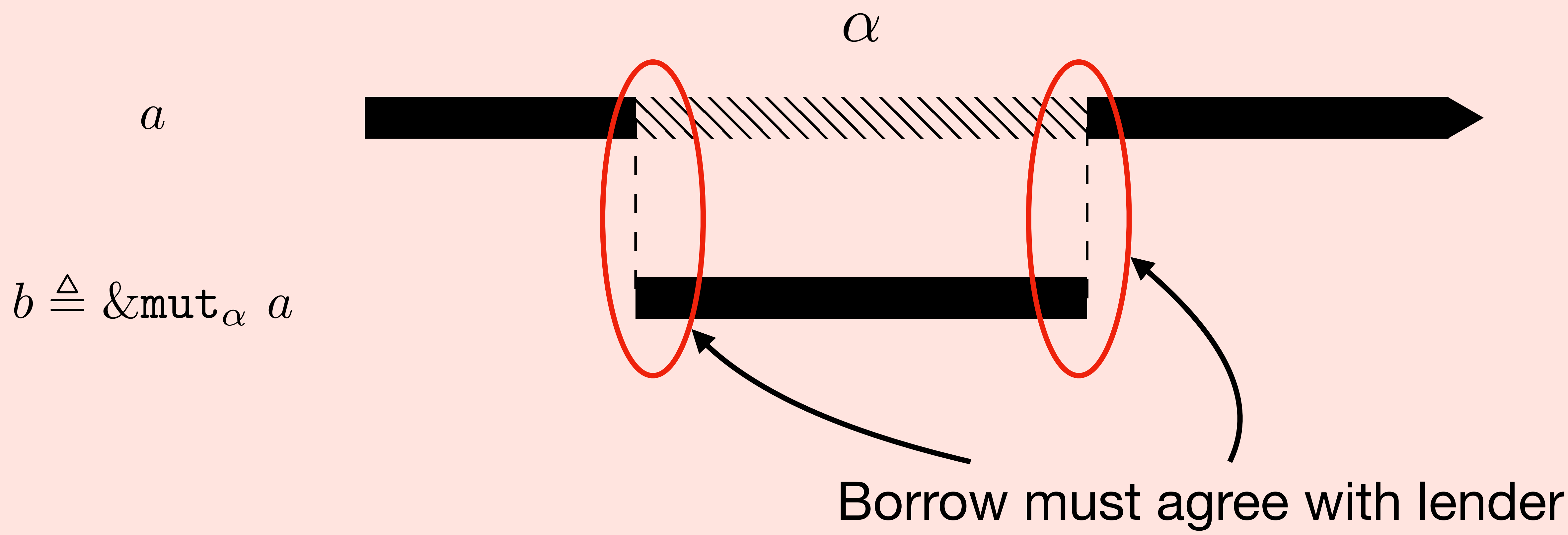


a is inaccessible for the duration of α

Prophecies

Synchronizing lender and borrower

- **Idea:** Model 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

- 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?

```
fn main () {
  let mut a = 0;
  let x = &mut a;
  let y = &mut 5;
  *x += *y;
  drop(x);
  assert_eq!(a, 5);
}
```

```
let main () =
  let a = 0 in
  let x = { cur = a ; fin = any } in
  let a = x.fin in
  let y = { cur = 5; fin = any } in
  let x = { x with cur += y.cur } in
  assume { x.fin = x.cur };
  assert { a = 5 }
```

Encoding Rust in ML

Mutable References?

```
fn main () {
  let mut a = 0;
  let x = &mut a;
  let y = &mut 5;
  *x += *y;
  drop(x);
  assert_eq!(a, 5);
}
```

```
let main () =
  let a = 0 in
  let x = { cur = a ; fin = any } in
  let a = x.fin in
  let y = { cur = 5; fin = any } in
  let x = { x with cur += y.cur } in
  assume { x.fin = x.cur };
  assert { a = 5 }
```

Encoding Rust in ML

Mutable References?

```
fn main () {
  let mut a = 0;
  let x = &mut a;
  let y = &mut 5;
  *x += *y;
  drop(x);
  assert_eq!(a, 5);
}
```

```
let main () =
  let a = 0 in
  let x = { cur = a ; fin = any } in
  let a = x.fin in
  let y = { cur = 5; fin = any } in
  let x = { x with cur += y.cur } in
  assume { x.fin = x.cur };
  assert { a = 5 }
```

Encoding Rust in ML

Mutable References?

```
fn main () {
  let mut a = 0;
  let x = &mut a;
  let y = &mut 5;
  *x += *y;
  drop(x);
  assert_eq!(a, 5);
}
```

```
let main () =
  let a = 0 in
  let x = { cur = a ; fin = any } in
  let a = x.fin in
  let y = { cur = 5; fin = any } in
  let x = { x with cur += y.cur } in
  assume { x.fin = x.cur };
  assert { a = 5 }
```

Specifying Iterators

Modeling Iterators

Produces & Completed

- $produces$ links two states of the iterator using a sequence of items.

- Each call to **`next`** *produces* a new element and updates the state of the iterator

- $produces$ can thus be seen as a transitive, reflexive, *transition relation:*

$$produces(I, [e_0, ..., e_n], I') \triangleq I \xrightarrow[next]{e_0} ... \xrightarrow[next]{e_n} I'$$

- $completed$ takes an iterator and states whether it is finished

Modeling iterators

```
trait Iterator: Sized {
    type Item;

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

Modeling iterators

```
trait Iterator: Sized {
    type Item;

    #[predicate]
    fn completed(self) -> bool;

    #[predicate]
    fn produces(self, visited: Seq<Self::Item>, _: Self) -> bool;

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

Modeling iterators

```
trait Iterator {
    type Item;

    #[predicate]
    fn completed(&mut self) -> bool;

    #[predicate]
    fn produces(self, visited: Seq<Self::Item>, _: Self) -> bool;

    #[law]
    #[ensures(a.produces(Seq::EMPTY, a))]
    fn produces_refl(a: Self);

    #[law]
    #[requires(a.produces(ab, b))]
    #[requires(b.produces(bc, c))]
    #[ensures(a.produces(ab.concat(bc), c))]
    fn produces_trans(a: Self, ab: Seq<Self::Item>, b: Self, bc:
      Seq<Self::Item>, c: Self);

   …
}
```

Modeling iterators

```
trait Iterator {
    type Item;

    #[predicate]
    fn completed(&mut self) -> bool;

    #[predicate]
    fn produces(self, visited: Seq<Self::Item>, _: Self) -> bool;

    ..

    #[ensures(match result {
      None => self.completed(),
      Some(v) => (*self).produces(Seq::singleton(v), ^self)
    })]
    fn next(&mut self) -> Option<Self::Item>;
}
```

Modeling iterators

```
trait Iterator {
    type Item;

    #[predicate]
    fn completed(&mut self) -> bool;

    #[predicate]
    fn produces(self, visited: Seq<Self::Item>, _: Self)

    ..

    #[ensures(match result {
      None => self.completed(),
      Some(v) => (*self).produces(Seq::singleton(v), ^self)
    })]
    fn next(&mut self) -> Option<Self::Item>;
}
```

Accesses the *final* value of a mutable borrow.

Unique to Creusot

IterMut

Next

```
struct IterMut<'a, T> {
    inner: &'a mut [T], // approximately the real thing
}

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

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

IterMut

Completed

```
struct IterMut<'a, T> {
    inner: &'a mut [T], // approximately the real thing
}

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

    #[predicate]
    fn completed(&mut self) -> bool {
        pearlite! { self.resolve() && (@self.inner).ext_eq(Seq::EMPTY) }
    }


    …
}
```

IterMut

Produces

```
struct IterMut<'a, T> {
    inner: &'a mut [T], // approximately the real thing
}

impl<'a, T> Iterator for IterMut<'a, T> {
    …

    #[predicate]
    fn produces(self, visited: Seq<Self::Item>, tl: Self) -> bool {
        self.inner.to_mut_seq().ext_eq(
            visited.concat(tl.inner.to_mut_seq())
        )
    }
    …
}
```

```
fn to_mut_seq(&mut [T]) -> Seq<&mut T>
```

IterMut

Laws

```
struct IterMut<'a, T> {
    inner: &'a mut [T], // approximately the real thing
}

impl<'a, T> Iterator for IterMut<'a, T> {
    …

    #[predicate]
    fn produces(self, visited: Seq<Self::Item>, tl: Self) -> bool { … }

    #[law]
    #[ensures(a.produces(Seq::EMPTY, a))]
    fn produces_refl(a: Self) {}

    #[law]
    #[requires(a.produces(ab, b))]
    #[requires(b.produces(bc, c))]
    #[ensures(a.produces(ab.concat(bc), c))]
    fn produces_trans(a: Self, ab: Seq<Self::Item>, b: Self, bc:
      Seq<Self::Item>, c: Self) {}
   …
}
```

for-loops

Reasoning about for-loops

Desugaring a for-loop

```
fn counter() {
    let mut v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w = v
        .iter_mut()
        .map(|x| { cnt += 1; a + 1 })
        .collect();

    assert_eq!(cnt, 4);

    for i in 0..4 {
        assert_eq!(w[i], v[i] + 1);
    }
}
```

Should be property of the **`Range`** iterator, but how does it work?

Reasoning about for-loops

Desugaring a for-loop

```
fn counter() {
    let mut v = vec![1, 2, 3, 4];
    let mut cnt = 0;
    let w = v
        .iter_mut()
        .map(|x| { cnt += 1; a + 1 })
        .collect();

    assert_eq!(cnt, 4);

    for i in 0..4 {
        assert_eq!(w[i], v[i] + 1);
    }
}
```

Reasoning about for-loops

Desugaring a for-loop

```
fn counter() {
    ...
    let it = 0..4;
    loop {
        match it.next() {
            Some(i) => {
                assert_eq!(w[i], v[i] + 1);
            }
            None => break,
        }
    }
}
```

Reasoning about for-loops

Desugaring a for-loop

```
fn counter() {
    ...
    let it = 0..4;
    loop {
        match it.next() {
            Some(i) => {
                assert_eq!(w[i], v[i] + 1);
            }
            None => break,
        }
    }
}
```

Problem: In the middle of loop, have no idea what the value of `i` is

Reasoning about for-loops

Desugaring a for-loop

```
fn counter() {
    ...
    let it = 0..4;
    #[invariant(??)]
    loop {
        match it.next() {
            Some(i) => {
                assert_eq!(w[i], v[i] + 1);
            }
            None => break,
        }
    }
}
```

Problem: In the middle of loop, have no idea what the value of `i` is

Solution: Add a loop invariant. But… which one? Also, should be *free*, as it is a property of **`Range`**

Reasoning about for-loops

Extending desugaring

```
fn counter() {
    ...
    let it = 0..4;
    let it_old = ghost! { it };
    let mut produced = ghost! { Seq::EMPTY };
    #[invariant(it_old.produces(produced, it))]
    loop {
        match it.next() {
            Some(i) => {
                produced = ghost! { produced.push(i) };
                assert_eq!(w[i], v[i] + 1);
            }
            None => break,
        }
    }
}
```

Extend for-loops with ghost information and add a *structural* invariant which is true for *all* iterators.

Reasoning about for-loops

User invariants

```
#[ensures((*v).len() == (^v).len())]
#[ensures(∀ i, 0 ≤ i < (^v).len() ==> (^v)[i] == 0)]
fn all_zero(v: &mut Vec<u32>) {
    #[invariant(????)]
    for i in v.iter_mut() {
      *i = 0;
    }

}
```

How do we specify the behavior of the loop?

Reasoning about for-loops

User invariants

```
#[ensures((*v).len() == (^v).len())]
#[ensures(∀ i, 0 ≤ i < (^v).len() ==> (^v)[i] == 0)]
fn all_zero(v: &mut Vec<u32>) {
    #[invariant(∀ i, 0 ≤ i < produced.len() ==> ^produced[i] == 0)]
    for i in v.iter_mut() {
      *i = 0;
    }

}
```

for-loop invariants can use **`produced`** to refer to past elements.

Wrap-Up

Recap

- We can model iterators as state machines
  - *produces* acts as the transition relation
  - *completed* describes the final states of the iteration
- Approach is flexible, can express:
  - Mutable iterators
  - Higher order iterators
  - Non-determinism, infinite iteration, etc…

Implementation

- We use this specfication to reason about iterators in Creusot

We've implemented a suite of real-world iterators:

`Once`	`Option`
`Empty`	`Repeat`
`Iter`	`IterMut`
`Map`	`MapExt`
`Skip`	`Take`
`FromIterator`	`IntoIterator`
`collect`	

- Https://github.com/xldenis/creusot