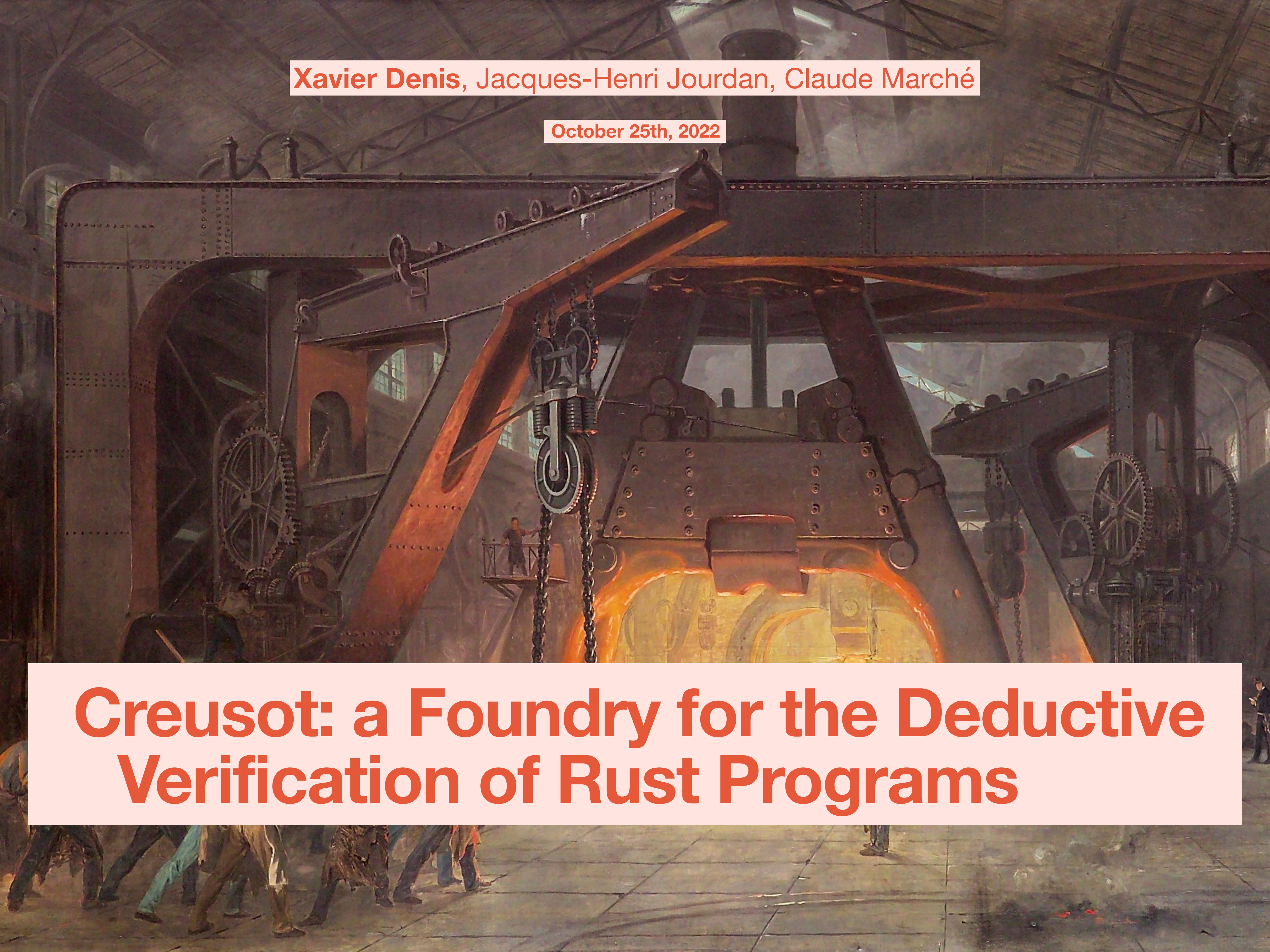


Xavier Denis, Jacques-Henri Jourdan, Claude Marché

October 25th, 2022

Creusot: a Foundry for the Deductive Verification of Rust Programs

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

```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

```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?

Enter Creusot

Based on **RustHorn** and **RustHornBelt**[1]

```
fn use_swap(a: &mut u32, b: &mut u32, c: &mut u32) {
  swap(a, b);
  assert!(a == old(b) && b == old(a) && c == old(c));
}
```

Uses **first-order logic**

Fully handles mutable pointers: even nested in structures

[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?

```
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);
}
```

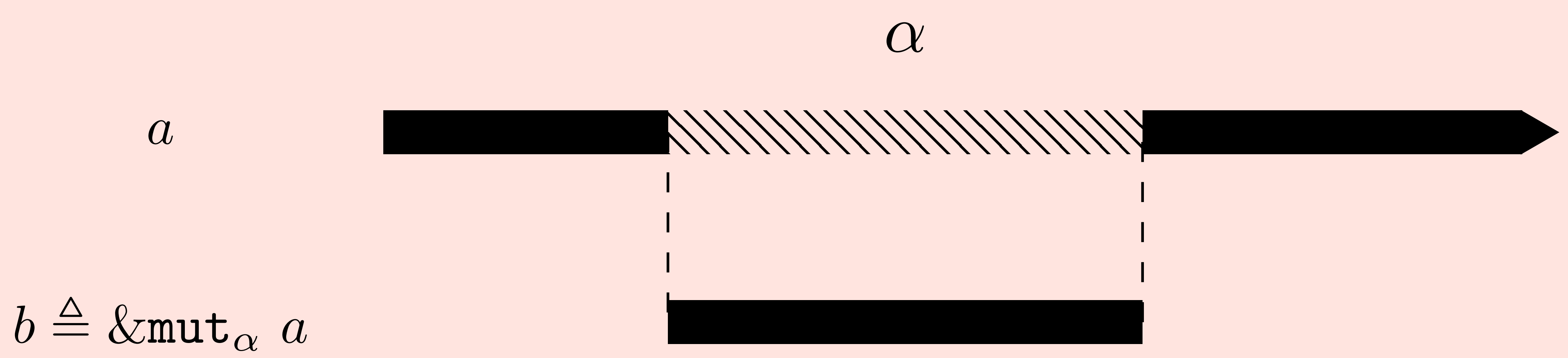
?

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

Prophecies

Synchronizing lender and borrower

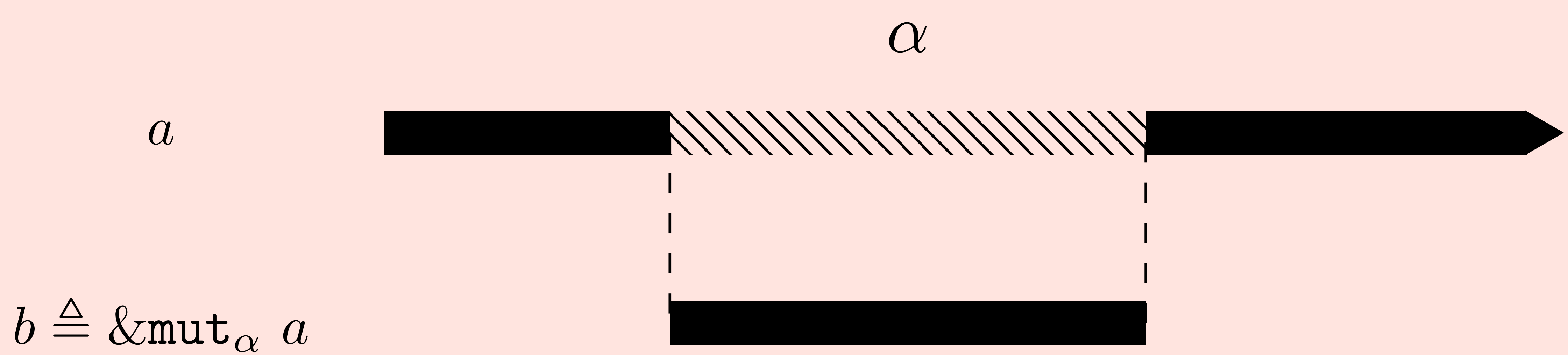
- **Idea:** Model mutable borrows as pair of **current** and **final** values
- We prophetize the final value, which the lender recovers.
- Depends on **uniqueness** and **lifetimes** of mutable borrows



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

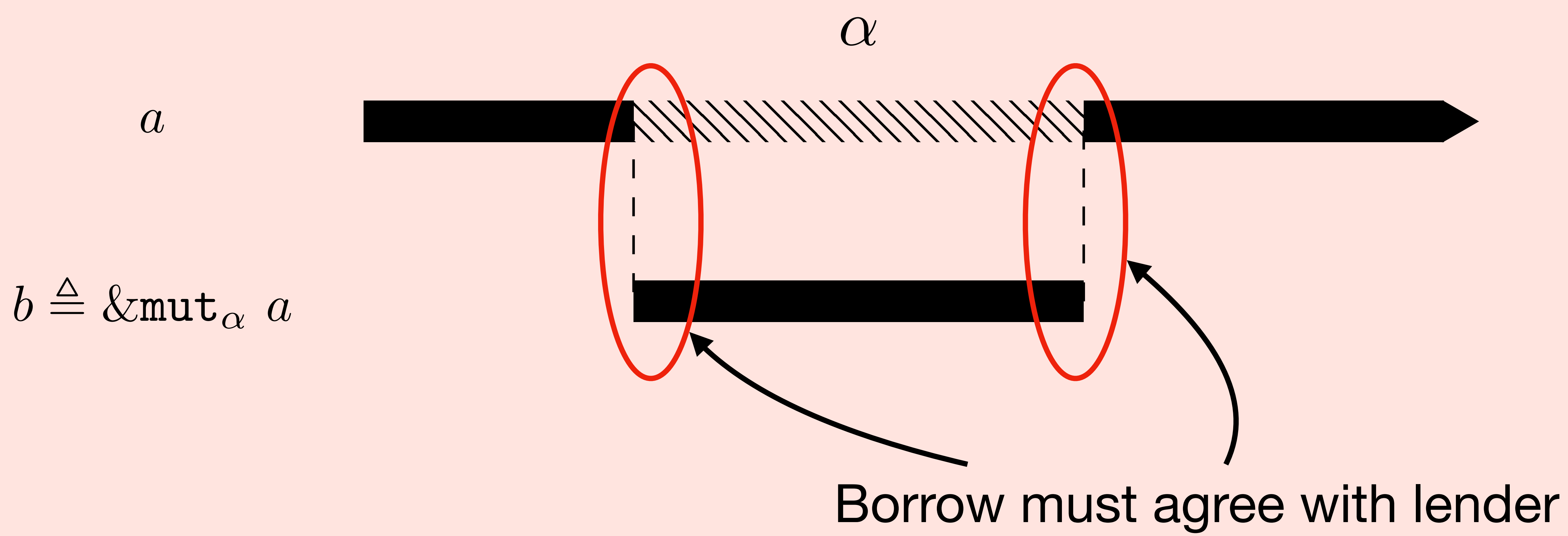


a is inaccessible for the duration of α

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?

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

```ocaml
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 }
```

What can Creusot do?

- **Specifications:** Creusot's specification language *Pearlite* adds:
  - Ghost predicates and functions
  - Contracts: ensures, requires, invariant
  - Laws: Used with *traits*
- **Traits:** Creusot adds support for ad-hoc polymorphism in Rust
  - Verification using *laws* and *contracts*, implementations must verify contract.

Example

Verifying Gnome Sort

```
fn gnome_sort<T: Ord>(v: &mut Vec<T>) {
   let mut i = 0;
   while i < v.len() {
        if i == 0 || v[i - 1] <= v[i] {
            i += 1;
        } else {
            v.swap(i - 1, i);
            i -= 1;
        }
    }
}
```

Verifying Gnome Sort

Specification Helpers

```
#[predicate]
fn sorted_range<T: Ord>(s: Seq<T>, l: Int, u: Int) -> bool {
    pearlite! {
        forall<i : Int, j : Int> l <= i && i < j && j < u ==>
            s[i] <= s[j]
    }
}

#[predicate]
fn sorted<T: Ord>(s: Seq<T>) -> bool {
    sorted_range(s, 0, s.len())
}
```

Verifying Gnome Sort

Specification Helpers

The `pearlite` macro gives access to custom syntax

```
#[predicate]
fn sorted_range<T: Ord>(s: Seq<T>, l: Int, u: Int) -> bool {
    pearlite! {
        forall<i : Int, j : Int> l <= i && i < j && j < u ==>
            s[i] <= s[j]
    }
}

#[predicate]
fn sorted<T: Ord>(s: Seq<T>) -> bool {
    sorted_range(s, 0, s.len())
}
```

Verifying Gnome Sort

Specification Helpers

```
#[predicate]
fn sorted_range<T: Ord>(s: Seq<T>, l: Int, u: Int) -> bool {
    pearlite! {
        forall<i : Int, j : Int> l <= i && i < j && j < u ==>
            s[i] <= s[j]
    }
}

#[predicate]
fn sorted<T: Ord>(s: Seq<T>) -> bool {
    sorted_range(s, 0, s.len())
}
```

Verifying Gnome Sort

Specification Helpers

```
#[predicate]
fn sorted_range<T: Ord>(s: Seq<T>, l: Int, u: Int) -> bool {
    pearlite! {
        forall<i : Int, j : Int> l <= i && i < j && j < u ==>
            s[i] <= s[j]
    }
}

#[predicate]
fn sorted<T: Ord>(s: Seq<T>) -> bool {
    sorted_range(s, 0, s.len())
}
```

??

Verifying Traits

A sketch for Ord

```
pub trait Ord {
    fn cmp(&self, rhs: &Self) -> Ordering;

    ...
}
```

Verifying Traits

A sketch for Ord

```
pub trait Ord {
    #[logic]
    fn cmp_log(self, _: Self) -> Ordering;

    ...

    #[ensures(result == self.cmp_log(*rhs))]
    fn cmp(&self, rhs: &Self) -> Ordering;

    #[law]
    #[requires(x.cmp_log(y) == o)]
    #[requires(y.cmp_log(z) == o)]
    #[ensures(x.cmp_log(z) == o)]
    fn trans(x: Self, y: Self, z: Self, o: Ordering);

    ...
}
```

Verifying Gnome Sort

Top level specification

```
fn gnome_sort<T: Ord>(v: &mut Vec<T>) {
    let mut i = 0;
    while i < v.len() {
        if i == 0 || v[i - 1] <= v[i] {
            i += 1;
        } else {
            v.swap(i - 1, i);
            i -= 1;
        }
    }
}
```

Verifying Gnome Sort

Top level specification

```
#[ensures(sorted(@^v))]
#[ensures((@^v).permutation_of(@*v))]
fn gnome_sort<T: Ord>(v: &mut Vec<T>) {
    let mut i = 0;
    while i < v.len() {
        if i == 0 || v[i - 1] <= v[i] {
            i += 1;
        } else {
            v.swap(i - 1, i);
            i -= 1;
        }
    }
}
```

Verifying Gnome Sort

Top level specification

`@` *'model'* is sugar for the logical model, here `Seq<T>`

```
#[ensures(sorted(@^v))]
#[ensures((@^v).permutation_of(@*v))]
fn gnome_sort<T: Ord>(v: &mut Vec<T>) {
    let mut i = 0;
    while i < v.len() {
        if i == 0 || v[i - 1] <= v[i] {
            i += 1;
        } else {
            v.swap(i - 1, i);
            i -= 1;
        }
    }
}
```

Model Types

- Pearlite is deeply integrated with Rust's type system
- We use traits to encode the *model value* pattern for specification
- This powers our `@` syntax

```
pub trait Model {
    type ModelTy;
    #[logic]
    fn model(self)
      -> Self::ModelTy;
}
```

```
pub impl<T> Model for Vec<T> {
    type ModelTy = Seq<T>;

    #[logic]
    #[trusted]
    fn model(self) -> Self::ModelTy {
        absurd
    }
}
```

Verifying Gnome Sort

Top level specification

`@` *‘model’* is sugar for the logical model, here `Seq<T>`

```
#[ensures(sorted(@^v))]
#[ensures((@^v).permutation_of(@*v))]
fn gnome_sort<T: Ord>(v: &mut Vec<T>) {
     let mut i = 0;
    while i < v.len() {
         if i == 0 || v[i - 1] <= v[i] {
             i += 1;
         } else {
             v.swap(i - 1, i);
             i -= 1;
         }
     }
}
```

Verifying Gnome Sort

Top level specification

`@` *'model'* is sugar for the logical model, here `Seq<T>`

^ the *'final'* value of borrow

```
#[ensures(sorted(@^v))]
#[ensures((@^v).permutation_of(@*v))]
fn gnome_sort<T: Ord>(v: &mut Vec<T>) {
     let mut i = 0;
    while i < v.len() {
        if i == 0 || v[i - 1] <= v[i] {
            i += 1;
        } else {
            v.swap(i - 1, i);
            i -= 1;
        }
    }
}
```

Verifying Gnome Sort

Top level specification

```
#[ensures(sorted(@^v))]
#[ensures((@^v).permutation_of(@*v))]
fn gnome_sort<T: Ord>(v: &mut Vec<T>) {
    let mut i = 0;
    #[invariant(sorted, sorted_range(@v, 0, @i))]
    #[invariant(permutation, (@*v).permutation_of(@*old(v)))]
    while i < v.len() {
        if i == 0 || v[i - 1] <= v[i] {
            i += 1;
        } else {
            v.swap(i - 1, i);
            i -= 1;
        }
    }
}
```

Evaluation

Selected from paper

	LOC	Spec LOC	Time (s)
Gnome Sort	11	17	2.06
Knapsack 0/1	32	106	5.96
Sparse Array	47	75	4.86
Filter Vector	21	39	0.98
HashMap	50	111	5.43

Applications

CreuSAT written and proved over 6 months by Sarek Skotåm

- Totals 3.5kloc of code and proofs with 2:1 overhead
- Proofs pass in ~3 minutes
- https://github.com/sarsko/CreuSAT

Future Work

Creusot is just beginning

For now we have laid a *foundation* for verification. Time to build up!

Closures & Iterators: Now supported in Creusot

Ghost Code: The combination of *ownership types* and *ghost code* offers potential for *lightweight resource algebras.*

Concurrency: Linear *RAs may enable powerful concurrent reasoning.*

Unsafe: Explore the limits of the prophetic model