OXIDE: THE ESSENCE OF RUST

Aaron J. Weiss
Northeastern University

> Rust's rich type system and ownership model guarantee memory-safety and thread-safety — enabling you to eliminate many classes of bugs at compile-time.
>
> – *the official Rust website*

BORROW CHECKING

WHAT IS A BORROW CHECKER?

```rust
struct State { ... }

fn main() {
    let mut state = State { ... };
    let init = read_state(&state);
    update_state(&mut state);
    let fin = read_state(&state);
    consume_state(state);

    // cannot use `state` anymore
}
```

WHAT IS A BORROW CHECKER?

```
struct State { … }

fn main() {
    let mut state = State { … };
    let init = read_state(&state);
    update_state(&mut state);
    let fin = read_state(&state);
    consume_state(state);

    // cannot use `state` anymore
}
```

borrow

WHAT IS A BORROW CHECKER?

```rust
struct State { ... }

fn main() {
    let mut state = State { ... };
    let init = read_state(&state);
    update_state(&mut state);
    let fin = read_state(&state);
    consume_state(state);

    // cannot use `state` anymore
}
```

borrow

mutable borrow

WHAT IS A BORROW CHECKER?

```
struct State { … }

fn main() {
    let mut state = State { … };
    let init = read_state(&state);
    update_state(&mut state);
    let fin = read_state(&state);
    consume_state(state);

    // cannot use `state` anymore
}
```

borrow

mutable borrow

move

WHAT IS A BORROW CHECKER?

```
fn update_state(state: &mut State) {
    if should_reset(state) {
        *state = State { … };
    } else {
        (*state).count += 1
    }
}
```

A BORROW CHECKER IS...

🧾

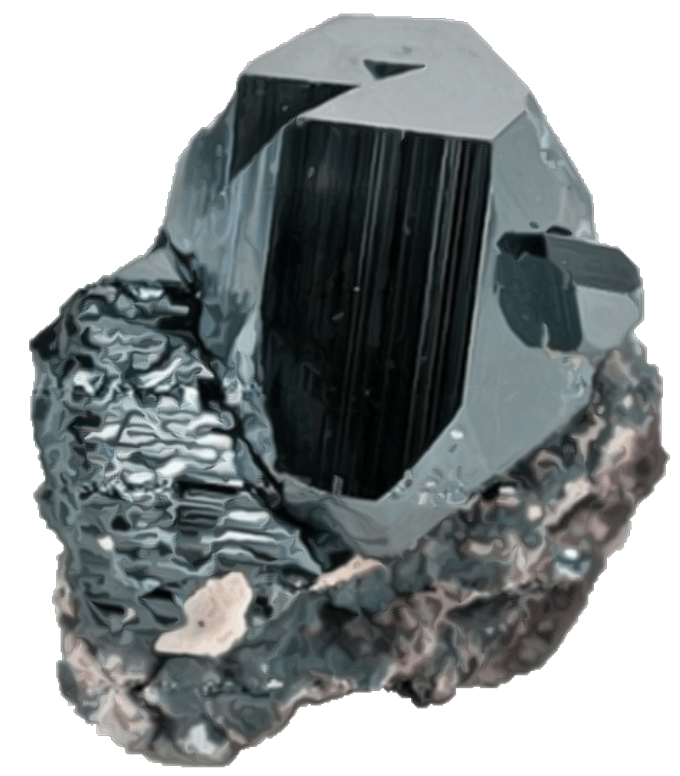
Ownership

&

Flexible Alias Protection

📖 ⊕ 🖋

OXIDE IS OUR EFFORT TO FORMALIZE BORROW CHECKING



OUR EXAMPLE PROGRAM IN OXIDE

```
struct State { ... }

fn main() {
    letprov<'a, 'b, 'c> {
        let state = State { ... };
        let init = read_state::<'a>(&'a shrd state);
        update_state::<'b>(&'b uniq state);
        let fin = read_state::<'c>(&'c shrd state);
        consume_state(state);
    }
}
```

OUR EXAMPLE PROGRAM IN OXIDE

```
struct State { ... }

fn main() {
    letprov<'a, 'b, 'c> {
        let state = State { ... };
        let init = read_state::<'a>(&'a shrd state);
        update_state::<'b>(&'b uniq state);
        let fin = read_state::<'c>(&'c shrd state);
        consume_state(state);
    }
}
```

OUR EXAMPLE PROGRAM IN OXIDE

```
struct State { ... }

fn main() {
    letprov<'a, 'b, 'c> {
        let state = State { ... };
        let init = read_state::<'a>(&'a shrd state);
        update_state::<'b>(&'b uniq state);
        let fin = read_state::<'c>(&'c shrd state);
        consume_state(state);
    }
}
```

OUR EXAMPLE PROGRAM IN OXIDE

```
struct State { ... }

fn main() {
    letprov<'a, 'b, 'c> {
        let state = State { ... };
        let init = read_state::<'a>(&'a shrd state);
        update_state::<'b>(&'b uniq state);
        let fin = read_state::<'c>(&'c shrd state);
        consume_state(state);
    }
}
```

OUR EXAMPLE PROGRAM IN OXIDE

```
struct State { ... }

fn main() {
    letprov<'a, 'b, 'c> {
        let state = State { ... };
        let init = read_state::<'a>(&'a shrd state);
        update_state::<'b>(&'b uniq state);
        let fin = read_state::<'c>(&'c shrd state);
        consume_state(state);
    }
}
```

PROVENANCES AS A 'POINTS-TO' ANALYSIS

A reference with type…

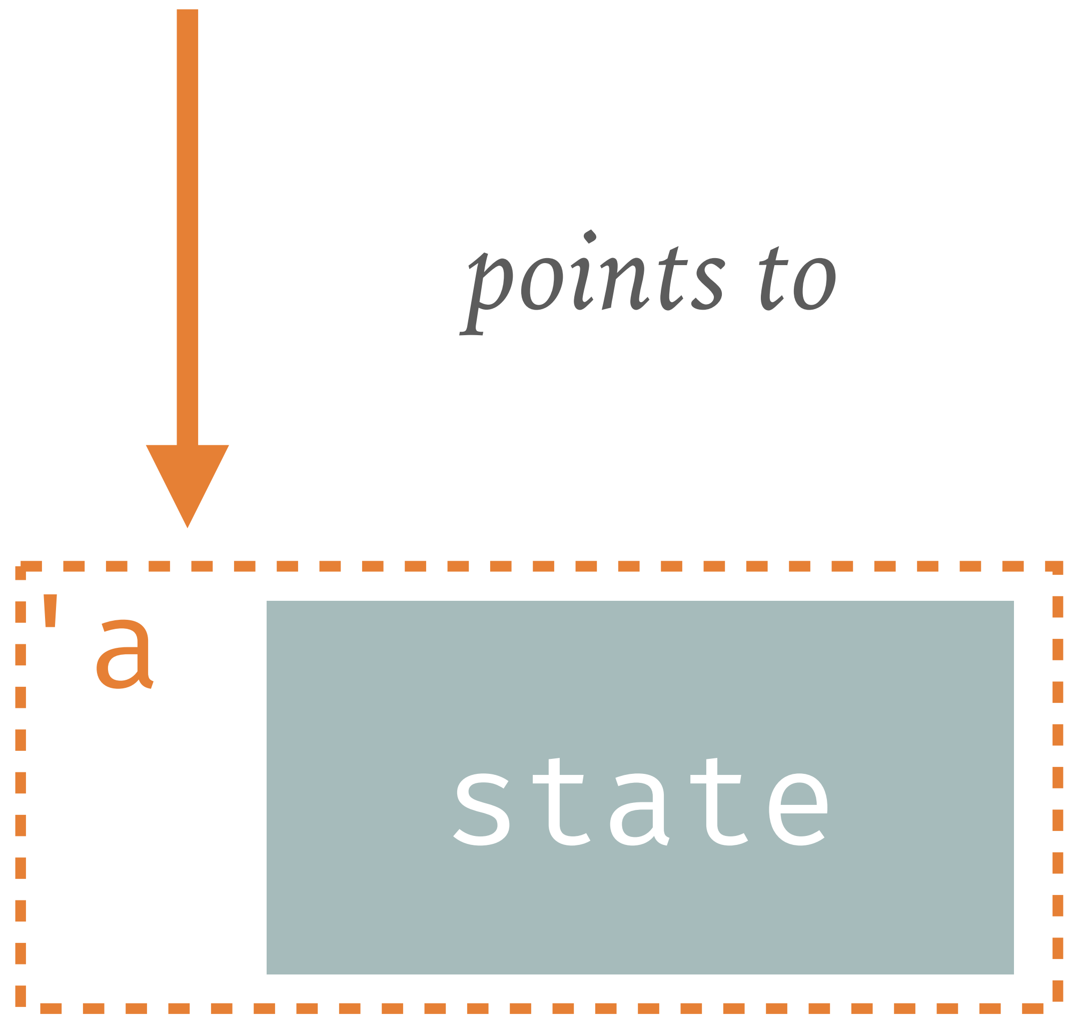
`&'a shrd state`   `&'b uniq state`

PROVENANCES AS A 'POINTS-TO' ANALYSIS

A reference with type…

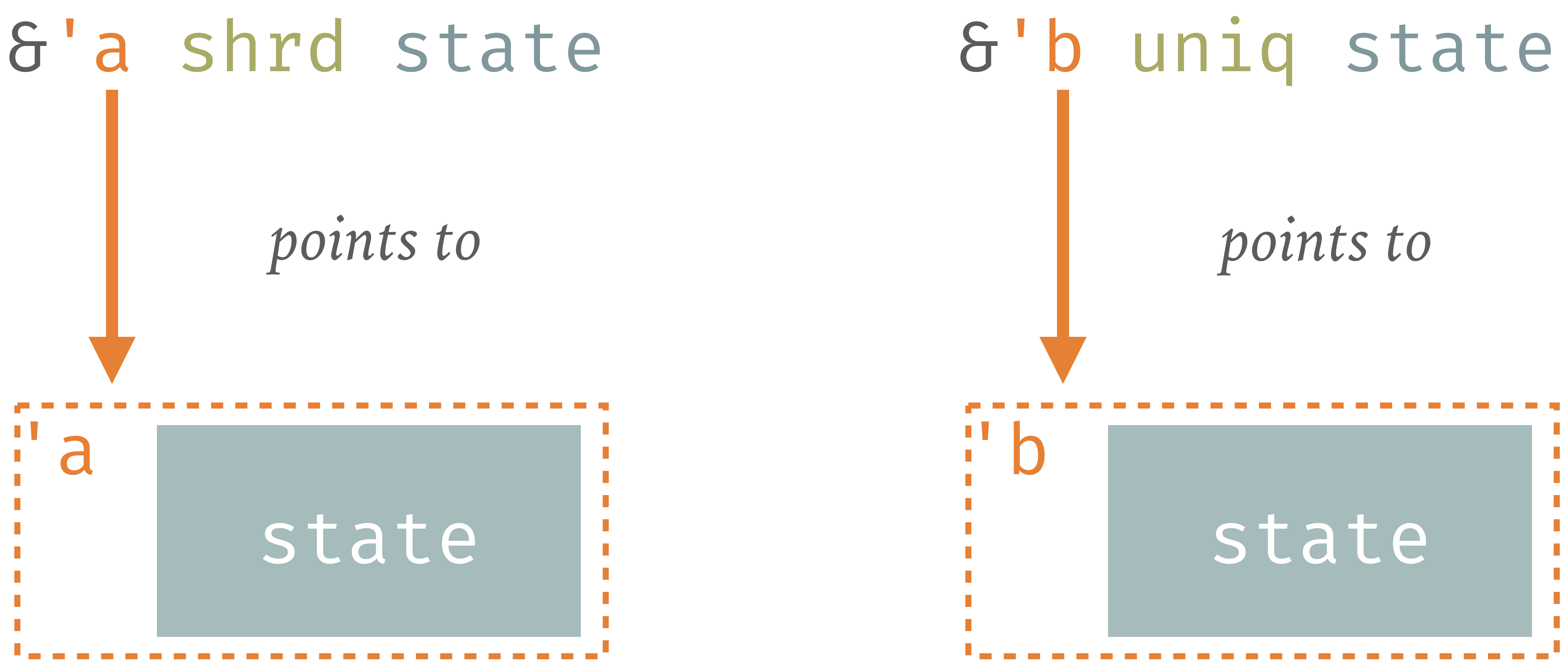
`&'a shrd state`

`&'b uniq state`



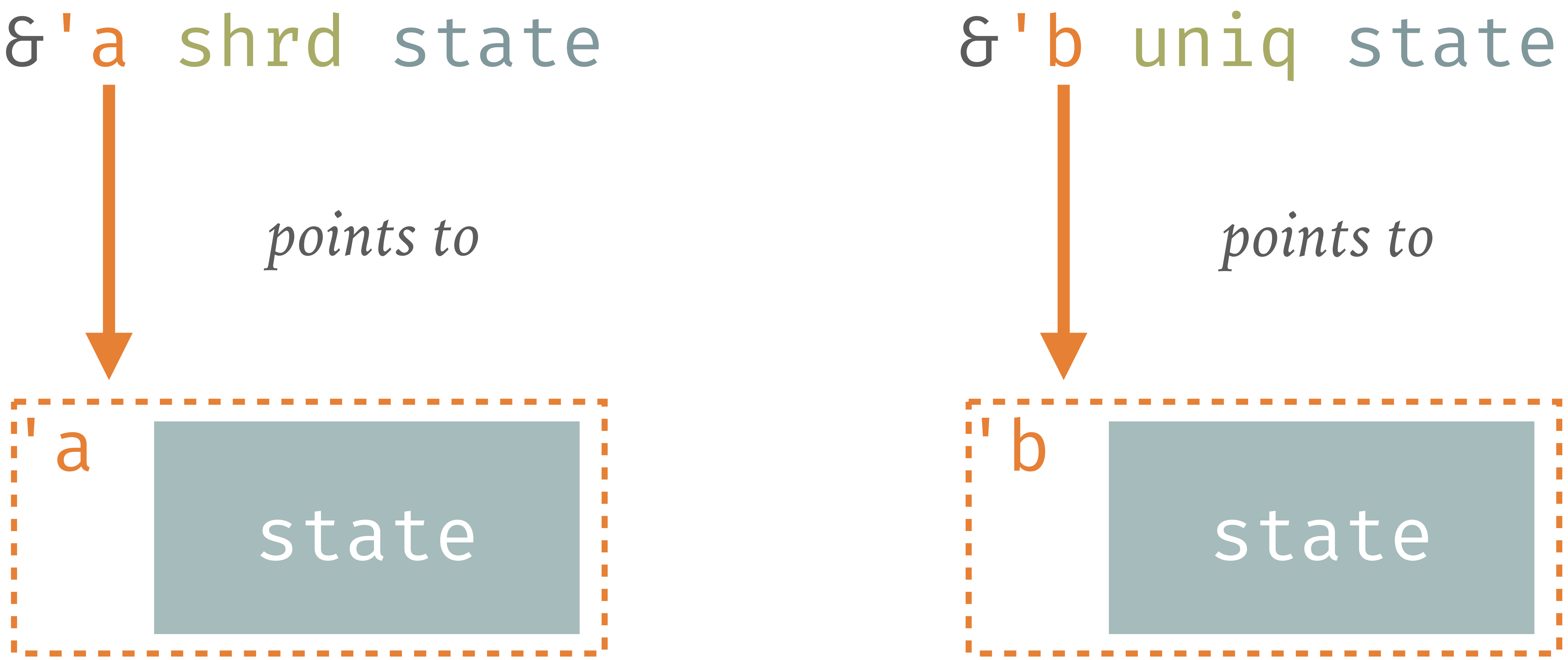
PROVENANCES AS A ‘POINTS-TO’ ANALYSIS

A reference with type…



PROVENANCES AS A 'POINTS-TO' ANALYSIS

A reference with type…



$$\Gamma ::= \bullet \mid \Gamma \natural \mathcal{F}$$

$$\mathcal{F} ::= \bullet \mid \mathcal{F},\ x : T \mid \mathcal{F},\ 'a \longmapsto \{\ p_1 \dots p_n\ \}$$

TYPECHECKING A MOVE EXPRESSION

```
struct State { ... }

fn main() {
    letprov<'a, 'b, 'c> {
        let state = State { ... };
        let init = read_state::<'a>(&'a shrd state);
        update_state::<'b>(&'b uniq state);
        let fin = read_state::<'c>(&'c shrd state);
        consume_state(state);
    }
}
```

TYPECHECKING A MOVE EXPRESSION

```
consume_state(state);
```

TYPECHECKING A MOVE EXPRESSION

```
consume_state(state);
```

$$\frac{\texttt{state}\ \textit{is not aliased} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta;\ \Gamma \vdash \texttt{state} : \texttt{State}}$$

TYPECHECKING A BORROW EXPRESSION

```
struct State { ... }

fn main() {
    letprov<'a, 'b, 'c> {
        let state = State { ... };
        let init = read_state::<'a>(&'a shrd state);
        update_state::<'b>(&'b uniq state);
        let fin = read_state::<'c>(&'c shrd state);
        consume_state(state);
    }
}
```

TYPECHECKING A BORROW EXPRESSION

```
update_state::<'b>(&'b uniq state);
```

TYPECHECKING A BORROW EXPRESSION

```
update_state::<'b>(&'b uniq state);
```

$$\frac{\texttt{state}\ \textit{is not aliased} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta;\, \Gamma \vdash \texttt{\&'a uniq state} : \texttt{\&'a uniq State}}$$

TYPECHECKING A BORROW EXPRESSION

$$\frac{\texttt{state}\ \textit{is not uniquely aliased} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta;\ \Gamma \vdash \texttt{\&'a shrd state} : \texttt{\&'a shrd State}}$$

```
update_state::<'b>(&'b uniq state);
```

$$\frac{\texttt{state}\ \textit{is not aliased} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta;\ \Gamma \vdash \texttt{\&'a uniq state} : \texttt{\&'a uniq State}}$$

OWNERSHIP SAFETY: “IS NOT ALIASED”

$$\Delta;\ \Gamma \vdash_{\texttt{uniq}} x \Rightarrow \{\ \ldots\ \} \qquad \Delta;\ \Gamma \vdash_{\texttt{shrd}} x \Rightarrow \{\ \ldots\ \}$$

OWNERSHIP SAFETY: "IS NOT ALIASED"

$\Delta;\ \Gamma \vdash_{\texttt{uniq}} x \Rightarrow \{\ \dots\ \}$ $\quad$ $\Delta;\ \Gamma \vdash_{\texttt{shrd}} x \Rightarrow \{\ \dots\ \}$

$\omega ::= uniq \mid shrd$

$\pi ::= x \mid \pi\,.\,n \mid \pi\,.f$

$$\Delta;\ \Gamma \vdash_{\omega} \pi \Rightarrow \{\ p_1\ \dots\ p_n\ \}$$

OWNERSHIP SAFETY: "IS NOT ALIASED"

$$\Delta;\ \Gamma \vdash_{\texttt{uniq}} x \Rightarrow \{\ \dots\ \}$$

$$\Delta;\ \Gamma \vdash_{\texttt{shrd}} x \Rightarrow \{\ \dots\ \}$$

$$\omega ::= uniq \mid shrd$$

$$\pi ::= x \mid \pi\,.\,n \mid \pi\,.f$$

$$\Delta;\ \Gamma \vdash_{\omega} \pi \Rightarrow \{\ p_1\ \dots\ p_n\ \}$$

places

OWNERSHIP SAFETY: "IS NOT ALIASED"

$$\Delta;\ \Gamma \vdash_{\texttt{uniq}} x \Rightarrow \{\ \ldots\ \}$$

$$\Delta;\ \Gamma \vdash_{\texttt{shrd}} x \Rightarrow \{\ \ldots\ \}$$

$\Downarrow$

$$\omega ::= uniq \mid shrd$$

$$\pi ::= x \mid \pi\,.\,n \mid \pi\,.\,f$$

$$\Delta;\ \Gamma \vdash_{\omega} \pi \Rightarrow \{\ p_1\ \ldots\ p_n\ \}$$

places (π)

place expressions ($p_1 \ldots p_n$)

THE STORY SO FAR

$$\frac{\texttt{state}\ \textit{is not aliased} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta;\ \Gamma \vdash \texttt{state} : \texttt{State}}$$

$$\frac{\texttt{state}\ \textit{is not aliased} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta;\ \Gamma \vdash \texttt{\&'a uniq state : \&'a uniq State}}$$

$$\frac{\texttt{state}\ \textit{is not uniquely aliased} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta;\ \Gamma \vdash \texttt{\&'a shrd state : \&'a shrd State}}$$

THE STORY SO FAR

$$\frac{\Delta; \Gamma \vdash_{\texttt{uniq}} \texttt{state} \Rightarrow \{\texttt{state}\} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta; \Gamma \vdash \texttt{state} : \texttt{State}}$$

$$\frac{\Delta; \Gamma \vdash_{\texttt{uniq}} \texttt{state} \Rightarrow \{\texttt{state}\} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta; \Gamma \vdash \texttt{\&'a uniq state : \&'a uniq State}}$$

$$\frac{\Delta; \Gamma \vdash_{\texttt{shrd}} \texttt{state} \Rightarrow \{\texttt{state}\} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta; \Gamma \vdash \texttt{\&'a shrd state : \&'a shrd State}}$$

A BIT OF A SNAG

$$\frac{\Delta; \Gamma \vdash_{\texttt{uniq}} \texttt{state} \Rightarrow \{\texttt{state}\} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta; \Gamma \vdash \texttt{state} : \texttt{State}}$$

Can we use `state` *again?*

A BIT OF A SNAG

$$\frac{\Delta; \Gamma \vdash_{\mathtt{uniq}} \mathtt{state} \Rightarrow \{\mathtt{state}\} \qquad \Gamma(\mathtt{state}) = \mathtt{State}}{\Delta; \Gamma \vdash \mathtt{state} : \mathtt{State}}$$

Can we use `state` *again?* ✗

A CONVENTIONAL APPROACH... ?

Convention: $\vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$

A CONVENTIONAL APPROACH... ?

Convention: $\vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$

Rust:

```
struct Point(i32, i32)

let pt = Point(5, 6);
...
add_one(pt.0);
...
add_one(pt.1);
```

AN (UN)CONVENTIONAL APPROACH

Environment passing!

$$\frac{\Delta;\ \Gamma \vdash_{\mathtt{uniq}} \mathtt{state} \Rightarrow \{\mathtt{state}\} \qquad \Gamma(\mathtt{state}) = \mathtt{State}}{\Delta;\ \Gamma \vdash \mathtt{state} : \mathtt{State}}$$

AN (UN)CONVENTIONAL APPROACH

Environment passing!

$$\frac{\Delta;\ \Gamma \vdash_{\texttt{uniq}} \texttt{state} \Rightarrow \{\,\texttt{state}\,\} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta;\ \Gamma \vdash \texttt{state} : \texttt{State}}$$

AN (UN)CONVENTIONAL APPROACH

Environment passing!

$$\frac{\Delta;\ \Gamma \vdash_{\mathtt{uniq}} \mathtt{state} \Rightarrow \{\mathtt{state}\} \qquad \Gamma(\mathtt{state}) = \mathtt{State}}{\Delta;\ \Gamma \vdash \mathtt{state} : \mathtt{State} \Rightarrow \Gamma[\mathtt{state} \longmapsto \mathtt{State}^{\dagger}]}$$

EXAMPLE: PROJECTING OUT OF A TUPLE

EXAMPLE: PROJECTING OUT OF A TUPLE

```
•; x : (i32, i32)
```

EXAMPLE: PROJECTING OUT OF A TUPLE

$\bullet; \; x : (\text{i32}, \text{i32}) \vdash$

EXAMPLE: PROJECTING OUT OF A TUPLE

$$\bullet;\ x : (\texttt{i32}, \texttt{i32}) \vdash x.0 : \texttt{i32}$$

EXAMPLE: PROJECTING OUT OF A TUPLE

$$\bullet;\ \mathtt{x} : \mathtt{(i32,\ i32)} \vdash \mathtt{x.0} : \mathtt{i32} \Rightarrow$$

EXAMPLE: PROJECTING OUT OF A TUPLE

$$\bullet;\ \mathtt{x} : \mathtt{(i32, i32)} \vdash \mathtt{x.0} : \mathtt{i32} \Rightarrow \mathtt{x} : \mathtt{(i32^{\dagger}, i32)}$$

GOOD FOR PROVENANCE TRACKING, TOO!

$$\frac{\Delta;\ \Gamma \vdash_{\texttt{uniq}} \texttt{state} \Rightarrow \{\texttt{state}\} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta;\ \Gamma \vdash \texttt{\&'a uniq state : \&'a uniq State}}$$

$$\frac{\Delta;\ \Gamma \vdash_{\texttt{shrd}} \texttt{state} \Rightarrow \{\texttt{state}\} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta;\ \Gamma \vdash \texttt{\&'a shrd state : \&'a shrd State}}$$

GOOD FOR PROVENANCE TRACKING, TOO!

$$\frac{\Delta; \Gamma \vdash_{\texttt{uniq}} \texttt{state} \Rightarrow \{\texttt{state}\} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta; \Gamma \vdash \texttt{\&'a uniq state : \&'a uniq State} \Rightarrow \Gamma[\texttt{'a} \mapsto \{\texttt{state}\}]}$$

$$\frac{\Delta; \Gamma \vdash_{\texttt{shrd}} \texttt{state} \Rightarrow \{\texttt{state}\} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta; \Gamma \vdash \texttt{\&'a shrd state : \&'a shrd State}}$$

GOOD FOR PROVENANCE TRACKING, TOO!

$$\frac{\Delta;\ \Gamma \vdash_{\texttt{uniq}} \texttt{state} \Rightarrow \{\texttt{state}\} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta;\ \Gamma \vdash \texttt{\&'a uniq state : \&'a uniq State} \Rightarrow \Gamma[\texttt{'a} \mapsto \{\texttt{state}\}]}$$

$$\frac{\Delta;\ \Gamma \vdash_{\texttt{shrd}} \texttt{state} \Rightarrow \{\texttt{state}\} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Delta;\ \Gamma \vdash \texttt{\&'a shrd state : \&'a shrd State} \Rightarrow \Gamma[\texttt{'a} \mapsto \{\texttt{state}\}]}$$

INTRODUCTIONS, IT'S A PLEASURE TO MEET YOU!

```
struct State { ... }

fn main() {
    letprov<'a, 'b, 'c> {
        let state = State { ... };
        let init = read_state::<'a>(&'a shrd state);
        update_state::<'b>(&'b uniq state);
        let fin = read_state::<'c>(&'c shrd state);
        consume_state(state);
    }
}
```

INTRODUCTIONS, IT'S A PLEASURE TO MEET YOU!

```
letprov<'a, 'b, 'c>      let state = State { … };
```

INTRODUCTIONS, IT'S A PLEASURE TO MEET YOU!

```
letprov<'a, 'b, 'c>        let state = State { ... };
```

$$\frac{\text{'a } \textit{is not in } \Gamma \qquad \Delta;\ \Gamma, \text{'a} \longmapsto \{\} \vdash e : T \Rightarrow \Gamma', \text{'a} \longmapsto \{\ldots\}}{\Delta;\ \Gamma \vdash \texttt{letprov<'a> \{ e \}} : T \Rightarrow \Gamma'}$$

INTRODUCTIONS, IT'S A PLEASURE TO MEET YOU!

```
letprov<'a, 'b, 'c>        let state = State { ... };
```

$$\frac{\texttt{'a}\ \textit{is not in}\ \Gamma \qquad \Delta;\ \Gamma, \texttt{'a} \longmapsto \{\} \vdash \texttt{e} : \texttt{T} \Rightarrow \Gamma', \texttt{'a} \longmapsto \{...\}}{\Delta;\ \Gamma \vdash \texttt{letprov<'a> \{ e \}} : \texttt{T} \Rightarrow \Gamma'}$$

$$\frac{\Delta;\ \Gamma \vdash \texttt{State \{ ... \}} : \texttt{State} \Rightarrow \Gamma \qquad \texttt{state}\ \textit{is not in}\ \Gamma \qquad \Delta;\ \Gamma, \texttt{state} : \texttt{State} \vdash \texttt{e} : \texttt{T} \Rightarrow \Gamma', \texttt{state} : ...}{\Delta;\ \Gamma \vdash \texttt{let state = State \{ ... \}; e} : \texttt{()} \Rightarrow \Gamma'}$$

SEQUENCING IS STRAIGHTFORWARD

Flow environments forward!

$$\frac{\Delta;\ \Gamma \vdash e_1 : T_1 \Rightarrow \Gamma_1 \qquad \Delta;\ \Gamma_1 \vdash e_2 : T_2 \Rightarrow \Gamma_2}{\Delta;\ \Gamma \vdash e_1;\ e_2 : T_2 \Rightarrow \Gamma_2}$$

SEQUENCING IS STRAIGHTFORWARD

Flow environments forward!

$$\frac{\Delta;\ \Gamma \vdash \texttt{e}_1 : \texttt{T}_1 \Rightarrow \Gamma_1 \qquad \Delta;\ \Gamma_1 \vdash \texttt{e}_2 : \texttt{T}_2 \Rightarrow \Gamma_2}{\Delta;\ \Gamma \vdash \texttt{e}_1;\ \texttt{e}_2 : \texttt{T}_2 \Rightarrow \Gamma_2}$$

$$\frac{\texttt{'a} \textit{ is not in } \Gamma \qquad \Delta;\ \Gamma, \texttt{'a} \longmapsto \{\} \vdash \texttt{e} : \texttt{T} \Rightarrow \Gamma', \texttt{'a} \longmapsto \{\ldots\}}{\Delta;\ \Gamma \vdash \texttt{letprov<'a> \{ e \}} : \texttt{T} \Rightarrow \Gamma'}$$

$$\frac{\Delta;\ \Gamma \vdash \texttt{State \{ ... \}} : \texttt{State} \Rightarrow \Gamma \qquad \texttt{state} \textit{ is not in } \Gamma \qquad \Delta;\ \Gamma, \texttt{state} : \texttt{State} \vdash \texttt{e} : \texttt{T} \Rightarrow \Gamma', \texttt{state} : \ldots}{\Delta;\ \Gamma \vdash \texttt{let state = State \{ ... \}; e} : \texttt{()} \Rightarrow \Gamma'}$$

SEQUENCING IS STRAIGHTFORWARD

Flow environments forward!

$$\frac{\Delta;\ \Gamma \vdash e_1 : T_1 \Rightarrow \Gamma_1 \qquad \Delta;\ \Gamma_1 \vdash e_2 : T_2 \Rightarrow \Gamma_2}{\Delta;\ \Gamma \vdash e_1;\ e_2 : T_2 \Rightarrow \Gamma_2}$$

'a *is not in* Γ

$$\frac{\Delta;\ \Gamma, \texttt{'a} \longmapsto \{\} \vdash e : T \Rightarrow \Gamma', \texttt{'a} \longmapsto \{\ldots\}}{\Delta;\ \Gamma \vdash \texttt{letprov<'a> \{ e \}} : T \Rightarrow \Gamma'}$$

Environment ordering matters!

state *is not in* Γ

$$\frac{\Delta;\ \Gamma \vdash \texttt{State \{ ... \}} : \texttt{State} \Rightarrow \Gamma \qquad \Delta;\ \Gamma, \texttt{state} : \texttt{State} \vdash e : T \Rightarrow \Gamma', \texttt{state} : \ldots}{\Delta;\ \Gamma \vdash \texttt{let state = State \{ ... \}; e} : \texttt{()} \Rightarrow \Gamma'}$$

BRANCHES CAUSE INFORMATION LOSS

$$\frac{\Delta;\ \Gamma \vdash \texttt{e}_1 : \texttt{Bool} \Rightarrow \Gamma_1 \qquad \Delta;\ \Gamma_1 \vdash \texttt{e}_2 : \texttt{T} \Rightarrow \Gamma_2 \qquad \Delta;\ \Gamma_1 \vdash \texttt{e}_3 : \texttt{T} \Rightarrow \Gamma_3}{\Delta;\ \Gamma \vdash \texttt{if e}_1\ \{\ \texttt{e}_2\ \}\ \texttt{else}\ \{\ \texttt{e}_3\ \} : \texttt{T} \Rightarrow \Gamma_2 \Cup \Gamma_3}$$

BRANCHES CAUSE INFORMATION LOSS

$$\frac{\Delta;\ \Gamma \vdash \texttt{e}_1 : \texttt{Bool} \Rightarrow \Gamma_1 \qquad \Delta;\ \Gamma_1 \vdash \texttt{e}_2 : \texttt{T} \Rightarrow \Gamma_2 \qquad \Delta;\ \Gamma_1 \vdash \texttt{e}_3 : \texttt{T} \Rightarrow \Gamma_3}{\Delta;\ \Gamma \vdash \texttt{if}\ \texttt{e}_1\ \{\ \texttt{e}_2\ \}\ \texttt{else}\ \{\ \texttt{e}_3\ \} : \texttt{T} \Rightarrow \Gamma_2 \Cup \Gamma_3}$$

BRANCHES CAUSE INFORMATION LOSS

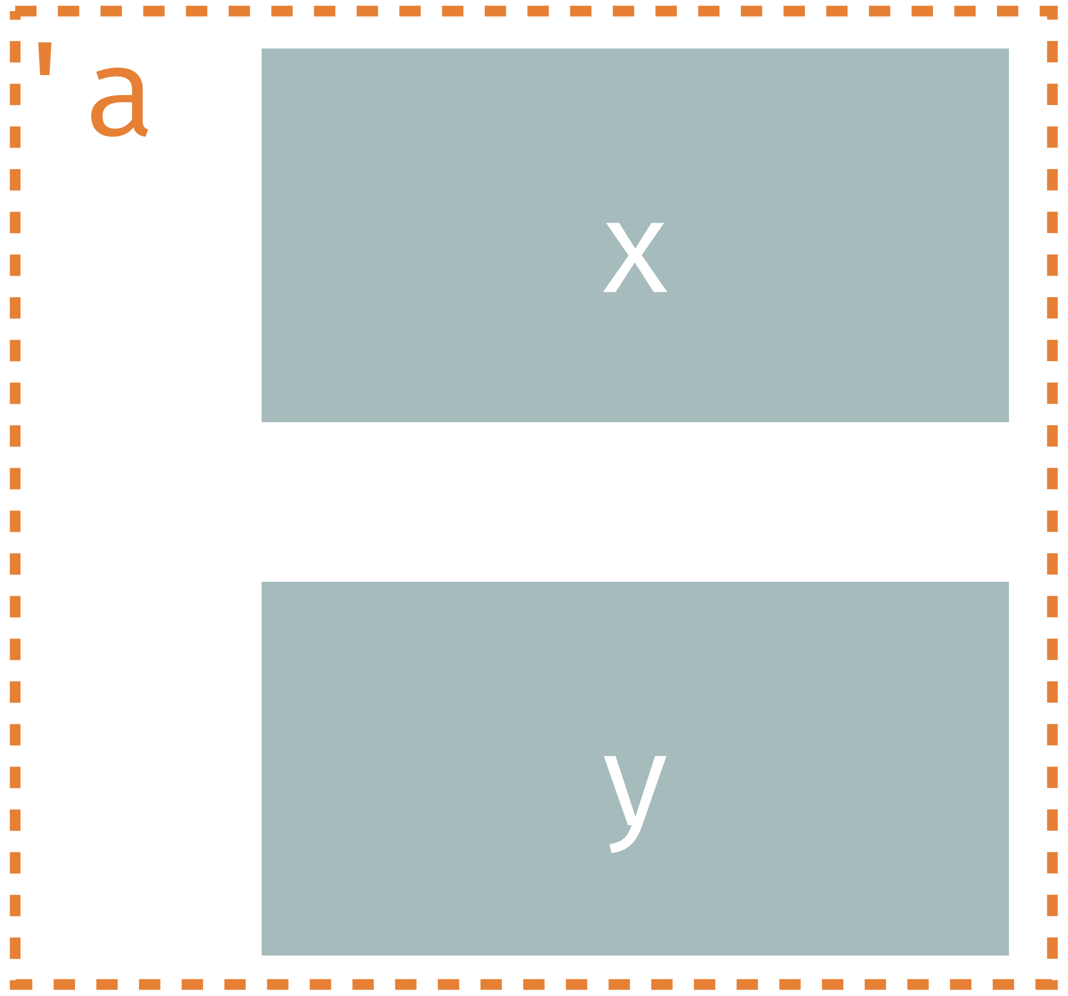
$$\frac{\Delta;\Gamma \vdash e_1 : \texttt{Bool} \Rightarrow \Gamma_1 \qquad \Delta;\Gamma_1 \vdash e_2 : \texttt{T} \Rightarrow \Gamma_2 \qquad \Delta;\Gamma_1 \vdash e_3 : \texttt{T} \Rightarrow \Gamma_3}{\Delta;\Gamma \vdash \texttt{if}\ e_1\ \{\ e_2\ \}\ \texttt{else}\ \{\ e_3\ \} : \texttt{T} \Rightarrow \Gamma_2 \Cup \Gamma_3}$$

```
if cond {
    &'a uniq x
} else {
    &'a uniq y
} // 'a ⟼ {x, y}
```

BRANCHES CAUSE INFORMATION LOSS

$$\frac{\Delta;\ \Gamma \vdash e_1 : \texttt{Bool} \Rightarrow \Gamma_1 \qquad \Delta;\ \Gamma_1 \vdash e_2 : \texttt{T} \Rightarrow \Gamma_2 \qquad \Delta;\ \Gamma_1 \vdash e_3 : \texttt{T} \Rightarrow \Gamma_3}{\Delta;\ \Gamma \vdash \texttt{if}\ e_1\ \{\ e_2\ \}\ \texttt{else}\ \{\ e_3\ \} : \texttt{T} \Rightarrow \Gamma_2 \Cup \Gamma_3}$$

```
if cond {
    &'a uniq x
} else {
    &'a uniq y
} // 'a ⟼ {x, y}
```



SOME SIMPLE SUBTYPING

Types can differ in their provenances!

$$\Delta;\ \Gamma \vdash T_1 \ <:\ T_2$$

SOME SIMPLE SUBTYPING

Types can differ in their provenances!

Combine provenances when they do!

$$\Delta;\ \Gamma \vdash \mathsf{T}_1 \ <:\ \mathsf{T}_2 \Rightarrow \Gamma'$$

SOME SIMPLE SUBTYPING

Types can differ in their provenances!

Combine provenances when they do!

$$\Delta;\ \Gamma \vdash \texttt{T}_1 \ \texttt{<:}\ \texttt{T}_2 \Rightarrow \Gamma'$$

$$\frac{\Delta;\ \Gamma \vdash \texttt{'a} \ \texttt{:>}\ \texttt{'b} \Rightarrow \Gamma' \qquad \Delta;\ \Gamma \vdash \texttt{State} \ \texttt{<:}\ \texttt{State} \Rightarrow \Gamma'}{\Delta;\ \Gamma \vdash \texttt{\&'a shrd State} \ \texttt{<:}\ \texttt{\&'b shrd State} \Rightarrow \Gamma'}$$

SUBTYPING BY EXAMPLE

```
if cond {
    &'a uniq x
    // 'a ⟼ {x}, 'b ⟼ {}
} else {
    &'b uniq y
    // 'a ⟼ {}, 'b ⟼ {y}
} // 'a ⟼ {x, y}, 'b ⟼ {x, y}
```

ASSIGNMENT “REGAINS” INFORMATION

ASSIGNMENT “REGAINS” INFORMATION

$$\bullet;\ x : \texttt{i32},\ y : \texttt{i32},\ \texttt{'a} \mapsto \{\ x,\ y\ \},\ z : \&\texttt{'a uniq i32} \vdash$$

ASSIGNMENT “REGAINS” INFORMATION

$$\bullet;\ x : \texttt{i32},\ y : \texttt{i32},\ \texttt{'a} \mapsto \{\ x,\ y\ \},\ z : \texttt{\&'a uniq i32} \vdash$$

$$*z := x : ()$$

ASSIGNMENT "REGAINS" INFORMATION

$$\bullet;\ x : \texttt{i32},\ y : \texttt{i32},\ \texttt{'a} \mapsto \{\ x,\ y\ \},\ z : \&\texttt{'a uniq i32} \vdash$$

$$*z := x : ()$$

$$\Rightarrow$$

ASSIGNMENT “REGAINS” INFORMATION

$$\bullet; \; x : \texttt{i32},\; y : \texttt{i32},\; \texttt{'a} \mapsto \{\, x, y \,\},\; z : \&\texttt{'a uniq i32} \vdash$$

$$*z := x : ()$$

$$\Rightarrow x : \texttt{i32},\; y : \texttt{i32},\; \texttt{'a} \mapsto \{\, x \,\},\; z : \&\texttt{'a uniq i32}$$

FUNCTIONS AND APPLICATION

```
struct State { ... }

fn main() {
    letprov<'a, 'b, 'c> {
        let state = State { ... };
        let init = read_state::<‘a>(&'a shrd state);
        update_state::<'b>(&'b uniq state);
        let fin = read_state::<'c>(&'c state);



}
```

FUNCTIONS AND APPLICATION

```
read_state::<‘a>(&'a shrd state)
```

FUNCTIONS AND APPLICATION

```
read_state::<‘a>(&'a shrd state)
```

$$\frac{\Delta;\,\Gamma \vdash \texttt{read\_state : ∀<'p>(\&'p shrd State) → T} \Rightarrow \Gamma_f \qquad \Delta;\,\Gamma_f \vdash \texttt{\&'a shrd state :\&'a shrd State} \Rightarrow \Gamma'}{\Delta;\,\Gamma \vdash \texttt{read\_state::<'a>(\&'a shrd state) : T} \Rightarrow \Gamma'}$$

FUNCTIONS AND APPLICATION

```
read_state::<‘a>(&'a shrd state)
```

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \texttt{read\_state} : \forall\texttt{<'p>(\&'p shrd State)} \to \texttt{T} \Rightarrow \Gamma_f \\ \Delta; \Gamma_f \vdash \texttt{\&'a shrd state} : \texttt{\&'a shrd State} \Rightarrow \Gamma'\end{array}}{\Delta; \Gamma \vdash \texttt{read\_state::<'a>(\&'a shrd state)} : \texttt{T} \Rightarrow \Gamma'}$$

$$\frac{}{\Delta; \Gamma \vdash \texttt{read\_state} : \forall\texttt{<'p>(\&'p shrd State)} \to \texttt{T} \Rightarrow \Gamma_f}$$

FUNCTIONS AND APPLICATION

`read_state::<‘a>(&'a shrd state)`

$$\frac{\Delta;\ \Gamma \vdash \texttt{read\_state : ∀<'p>(&'p shrd State) → T} \Rightarrow \Gamma_f \qquad \Delta;\ \Gamma_f \vdash \texttt{&'a shrd state :&'a shrd State} \Rightarrow \Gamma'}{\Delta;\ \Gamma \vdash \texttt{read\_state::<'a>(&'a shrd state) : T} \Rightarrow \Gamma'}$$

$$\frac{\Sigma(\texttt{read\_state}) = \texttt{fn update\_state<'p>(state: &'p shrd State) → T \{ e \}}}{\Sigma;\ \Delta;\ \Gamma \vdash \texttt{read\_state : ∀<'p>(&'p shrd State) → T} \Rightarrow \Gamma_f}$$

FUNCTIONS AND APPLICATION

```
read_state::<'a>(&'a shrd state)
```

$$\frac{\begin{array}{c}\Sigma; \Delta; \Gamma \vdash \texttt{read\_state} : \forall\texttt{<'p>(\&'p shrd State)} \to \texttt{T} \Rightarrow \Gamma_f \\ \Sigma; \Delta; \Gamma_f \vdash \texttt{\&'a shrd state} : \texttt{\&'a shrd State} \Rightarrow \Gamma'\end{array}}{\Sigma; \Delta; \Gamma \vdash \texttt{read\_state::<'a>(\&'a shrd state)} : \texttt{T} \Rightarrow \Gamma'}$$

$$\frac{\Sigma(\texttt{read\_state}) = \texttt{fn update\_state<'p>(state: \&'p shrd State)} \to \texttt{T \{ e \}}}{\Sigma; \Delta; \Gamma \vdash \texttt{read\_state} : \forall\texttt{<'p>(\&'p shrd State)} \to \texttt{T} \Rightarrow \Gamma_f}$$

CLOSURES MOVE THEIR FREE VARIABLES

$$\frac{\textit{free-vars}(\texttt{|y: i32| -> i32 \{ x + y \}}) = \texttt{x} \qquad \mathscr{F}_c = \texttt{x : i32} \qquad \Sigma;\ \Delta;\ \Gamma \natural \mathscr{F}_c, \texttt{y : i32} \vdash \texttt{x + y : i32} \Rightarrow \Gamma' \natural \mathscr{F}}{\Sigma;\ \Delta;\ \Gamma \vdash \texttt{|y: i32| -> i32 \{ x + y \} : (i32)} \xrightarrow{\mathscr{F}_c} \texttt{i32} \Rightarrow \Gamma'}$$

CLOSURES MOVE THEIR FREE VARIABLES

$$\frac{\textit{free-vars}(|\texttt{y: i32}| \rightarrow \texttt{i32 \{ x + y \}}) = \texttt{x} \qquad \mathscr{F}_c = \texttt{x : i32} \qquad \Sigma;\ \Delta;\ \Gamma \natural \mathscr{F}_c, \texttt{y : i32} \vdash \texttt{x + y : i32} \Rightarrow \Gamma' \natural \mathscr{F}}{\Sigma;\ \Delta;\ \Gamma \vdash |\texttt{y: i32}| \rightarrow \texttt{i32 \{ x + y \} : (i32)} \overset{\mathscr{F}_c}{\rightarrow} \texttt{i32} \Rightarrow \Gamma'}$$

$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \texttt{add\_x : (i32)} \overset{\mathscr{F}_c}{\rightarrow} \texttt{i32} \Rightarrow \Gamma_f \qquad \Sigma;\ \Delta;\ \Gamma_f \vdash \texttt{5 : i32} \Rightarrow \Gamma_f}{\Sigma;\ \Delta;\ \Gamma \vdash \texttt{add\_x(5) : i32} \Rightarrow \Gamma_f}$$

LET'S STEP BACK A BIT

REBORROWING

```
struct Point(i32, i32)

letprov<'r, 'x, 'y> {
    let pt = Point(5, 6);
    let r = &'r uniq pt;  // 'r ⟼ { pt }
    let x = &'x uniq (*r).0; // 'x ⟼ { pt, (*r).0 }
    let y = &'y uniq (*r).1; // 'y ⟼ { pt, (*r).1 }
    r
}
```

REBORROWING

```
struct Point(i32, i32)

letprov<'r, 'x, 'y> {
    let pt = Point(5, 6);
    let r = &'r uniq pt;  // 'r ⟼ { pt }
    let x = &'x uniq (*r).0; // 'x ⟼ { pt, (*r).0 }
    let y = &'y uniq (*r).1; // 'y ⟼ { pt, (*r).1 }
    r
}
```

$$\frac{\Delta;\, \Gamma \vdash_{\texttt{uniq}} \texttt{(*r).0} \Rightarrow \{\, \texttt{pt, (*r).0} \,\} \qquad \Gamma(\texttt{pt.0}) = \texttt{i32}}{\Delta;\, \Gamma \vdash \texttt{\&'x uniq (*r).0} : \texttt{\&'x uniq Point} \Rightarrow \Gamma[\texttt{'x} \longmapsto \{\, \texttt{pt, (*r).0} \,\}]}$$

NON-LEXICAL LIFETIMES IN OXIDE

```
struct Point(i32, i32)

fn main() {
    letprov<'x, 'y> {
        let pt = Point(6, 9);
        let x = &'x uniq pt;
        drop::<&'x uniq Point>(x);
        let y = &'y uniq pt;
    }
}
```

NON-LEXICAL LIFETIMES IN OXIDE

```
struct Point(i32, i32)

fn main() {
    letprov<'x, 'y> {
        let pt = Point(6, 9);
        let x = &'x uniq pt;
        // drop::<&'x uniq Point>(x);
        let y = &'y uniq pt;
    }
}
```

PROVING TYPE SAFETY FOR OXIDE

Progress

Preservation

PROVING TYPE SAFETY FOR OXIDE

Progress

If $\Sigma; \bullet; \Gamma \vdash \mathsf{e} : \mathsf{T} \Rightarrow \Gamma'$ *and* $\Sigma \vdash \sigma : \Gamma$*, then*
e *is a value,* e *is an error term, or*
$\exists \sigma', \mathsf{e}'. \Sigma \vdash (\sigma; \mathsf{e}) \rightarrow (\sigma'; \mathsf{e}')$

Preservation

PROVING TYPE SAFETY FOR OXIDE

Progress

If $\Sigma;\ \bullet;\ \Gamma \vdash e : \tau \Rightarrow \Gamma'$ *and* $\Sigma \vdash \sigma : \Gamma$, *then*

e *is a value,* e *is an error term, or*

$\exists \sigma', e'.\ \Sigma \vdash (\sigma;\ e) \to (\sigma';\ e')$

Preservation

If $\Sigma;\ \bullet;\ \Gamma \vdash e : \tau \Rightarrow \Gamma',\quad \Sigma \vdash \sigma : \Gamma,$

$\Sigma \vdash (\sigma;\ e) \to (\sigma';\ e')$, *and* $\Sigma;\ \Gamma \vDash \sigma$, *then*

$\exists \Gamma_i.\ \Sigma;\ \bullet;\ \Gamma_i \vdash e' : \tau \Rightarrow \Gamma'\ \ \Sigma \vdash \sigma' : \Gamma_i$, *and* $\Sigma;\ \Gamma_i \vDash \sigma'$

ONCE MORE, WITH FEELING

$$\frac{\Delta;\ \Gamma \vdash_{\omega} \texttt{state} \Rightarrow \{\texttt{state}\} \qquad \Gamma(\texttt{state}) = \texttt{State}}{\Sigma;\ \Delta;\ \Gamma \vdash \texttt{\&'a}\ \omega\ \texttt{state} : \texttt{\&'a}\ \omega\ \texttt{State} \Rightarrow \Gamma[\texttt{'a} \longmapsto \{\texttt{state}\}]}$$

$$\frac{\texttt{'a}\ \textit{is not in}\ \Gamma \qquad \Sigma;\ \Delta;\ \Gamma, \texttt{'a} \longmapsto \{\} \vdash \texttt{e} : \texttt{T} \Rightarrow \Gamma', \texttt{'a} \longmapsto \{\ldots\}}{\Sigma;\ \Delta;\ \Gamma \vdash \texttt{letprov<'a> \{ e \}} : \texttt{T} \Rightarrow \Gamma'}$$

$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \texttt{e}_1 : \texttt{T}_1 \Rightarrow \Gamma_1 \qquad \Sigma;\ \Delta;\ \Gamma_1 \vdash \texttt{e}_2 : \texttt{T}_2 \Rightarrow \Gamma_2}{\Sigma;\ \Delta;\ \Gamma \vdash \texttt{e}_1\texttt{;}\ \texttt{e}_2 : \texttt{T}_2 \Rightarrow \Gamma_2}$$

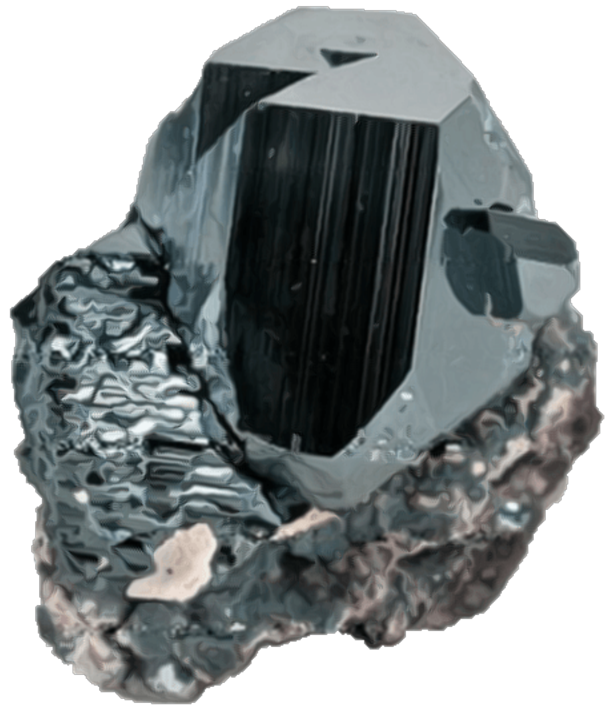
$$\frac{\Sigma;\ \Delta;\ \Gamma \vdash \texttt{State \{ ... \}} : \texttt{State} \Rightarrow \Gamma \qquad \texttt{state}\ \textit{is not in}\ \Gamma \qquad \Sigma;\ \Delta;\ \Gamma, \texttt{state} : \texttt{State} \vdash \texttt{e} : \texttt{T} \Rightarrow \Gamma', \texttt{state} : \ldots}{\Sigma;\ \Delta;\ \Gamma \vdash \texttt{let state = State \{ ... \}; e} : \texttt{()} \Rightarrow \Gamma'}$$

$$\frac{\Sigma(\texttt{read\_state}) = \texttt{fn update\_state<'p>(state: \&'p shrd State)} \rightarrow \texttt{T \{ e \}}}{\Sigma;\ \Delta;\ \Gamma \vdash \texttt{read\_state} : \forall\texttt{<'p>(\&'p shrd State)} \rightarrow \texttt{T} \Rightarrow \Gamma_f}$$

$$\frac{\textit{free-vars}(\texttt{|y: i32|} \rightarrow \texttt{i32 \{ x + y \}}) = \texttt{x} \qquad \mathscr{F}_c = \texttt{x : i32} \qquad \Sigma;\ \Delta;\ \Gamma \natural \mathscr{F}_c, \texttt{y : i32} \vdash \texttt{x + y : i32} \Rightarrow \Gamma' \natural \mathscr{F}}{\Sigma;\ \Delta;\ \Gamma \vdash \texttt{|y: i32|} \rightarrow \texttt{i32 \{ x + y \}} : \texttt{(i32)} \overset{\mathscr{F}_c}{\rightarrow} \texttt{i32} \Rightarrow \Gamma'}$$

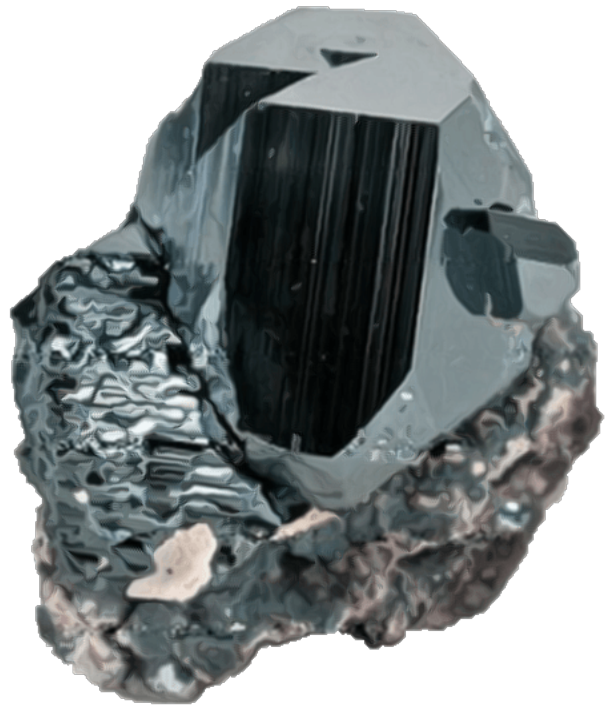
TAKEAWAYS

TAKEAWAYS



OXIDE IS A FORMALIZATION OF BORROW CHECKING

TAKEAWAYS

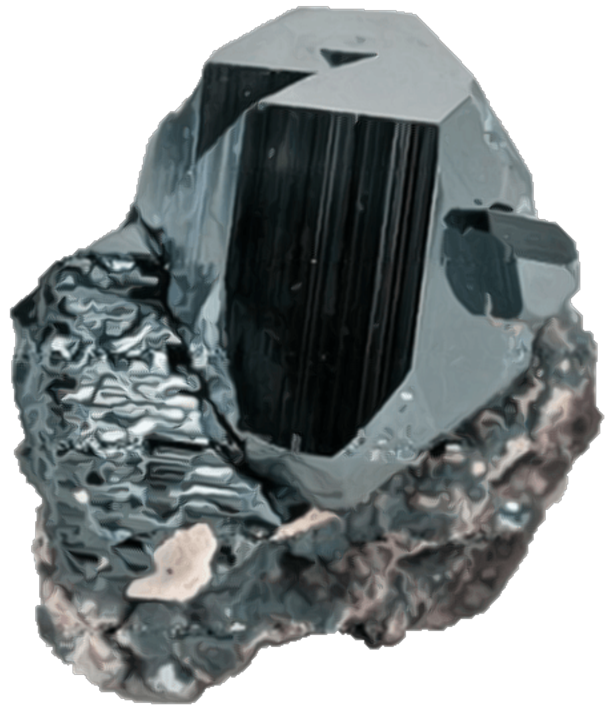


OXIDE IS A FORMALIZATION OF BORROW CHECKING

BORROW CHECKING COMBINES OWNERSHIP & ALIAS PROTECTION

⊕

TAKEAWAYS



OXIDE IS A FORMALIZATION OF BORROW CHECKING

BORROW CHECKING COMBINES OWNERSHIP & ALIAS PROTECTION

 ⊕ 

BORROW CHECKING IS SAFE