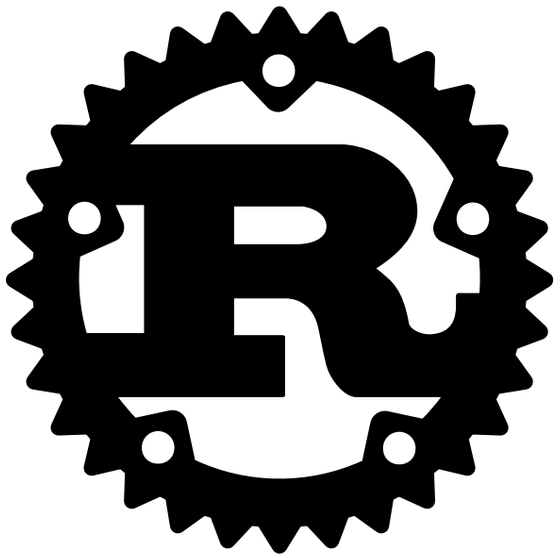
RUST DISTILLED:
AN EXPRESSIVE TOWER OF LANGUAGES

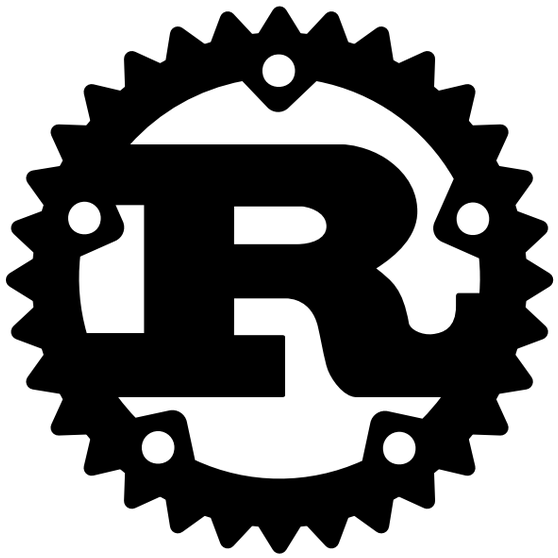
Aaron Weiss, *Daniel Patterson, Amal Ahmed*
Northeastern University and Inria Paris



“

Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.

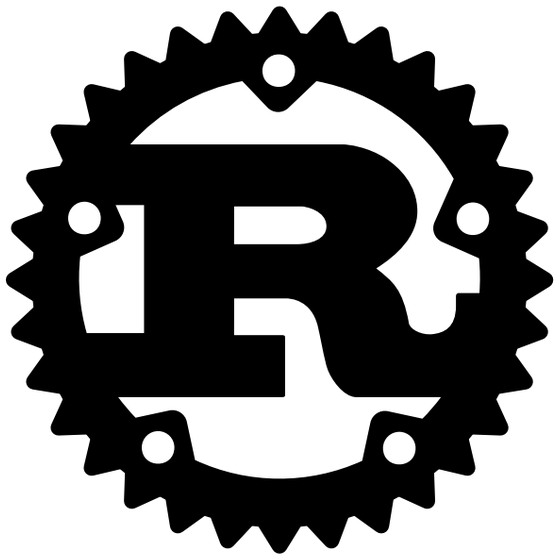
– *the official Rust website*



“

Rust is a *systems programming* language that runs blazingly fast, prevents segfaults, and guarantees thread safety.

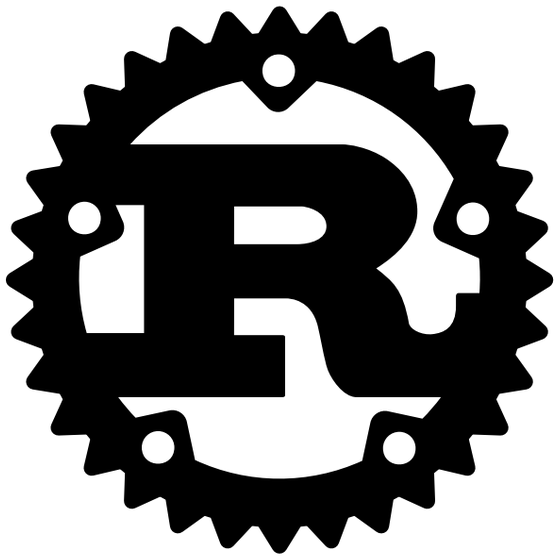
– *the official Rust website*



“

Rust is a systems programming language that runs *blazingly fast*, prevents segfaults, and guarantees thread safety.

– *the official Rust website*



“

Rust is a systems programming language that runs blazingly fast, *prevents segfaults*, and *guarantees thread safety.*

– *the official Rust website*

Memory safety without garbage collection

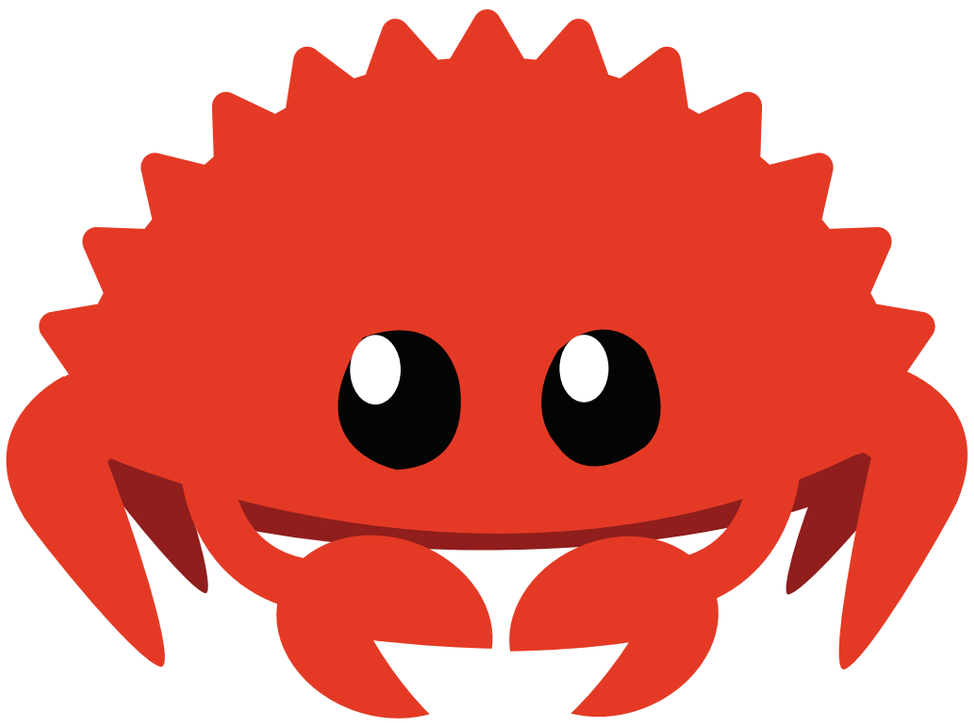
Abstraction without overhead

Concurrency without data races

Stability without stagnation

Hack without fear.

WE HAVE CUTE CRABS



... BUT HOW?

... BUT HOW?

Ownership

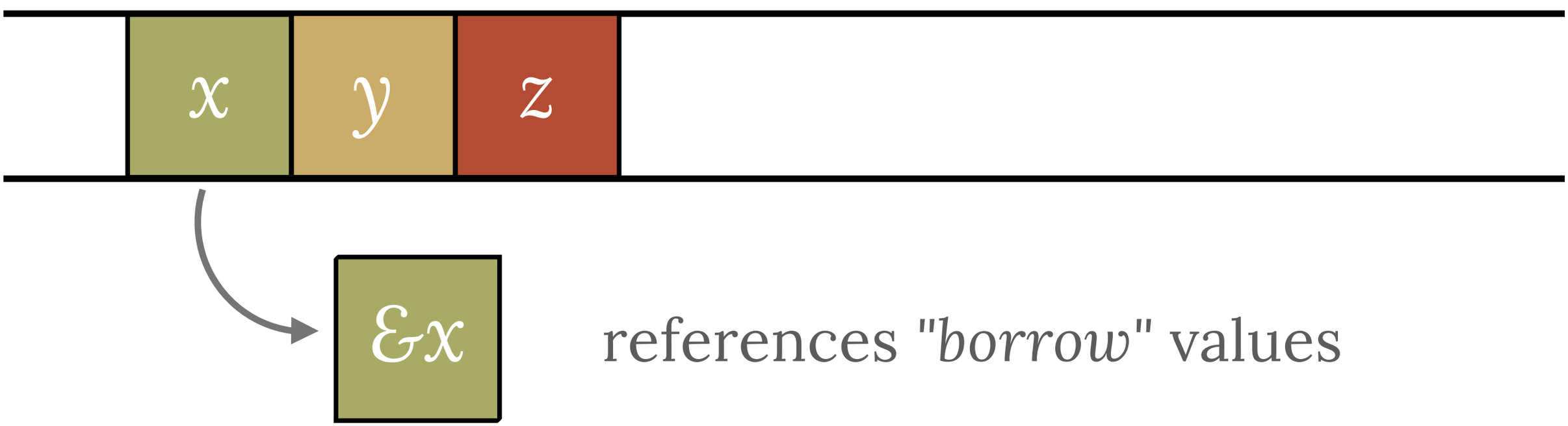
identifiers "*own*" values

... BUT HOW?

Ownership

identifiers "*own*" values

Borrowing



references "*borrow*" values

A PROGRAM IN RUST

```rust
extern crate irc;
use irc::client::prelude::*;

fn main() -> irc::error::Result<()> {
    let config = Config { ... };
    let mut reactor = IrcReactor::new()?;
    let client = reactor.prepare_client_and_connect(&config)?;
    client.identify()?;

    reactor.register_client_with_handler(client, |client, message| {
        print!("{}", message);
        Ok(())
    });

    reactor.run()?;
}
```

A PROGRAM IN RUST

```rust
extern crate irc;
use irc::client::prelude::*;

fn main() -> irc::error::Result<()> {
    let config = Config { ... };
    let mut reactor = IrcReactor::new()?;
    let client = reactor.prepare_client_and_connect(&config)?;
    client.identify()?;

    reactor.register_client_with_handler(client, |client, message| {
        print!("{}", message);
        Ok(())
    });

    reactor.run()?;
}
```

A PROGRAM IN RUST

```rust
extern crate irc;
use irc::client::prelude::*;

fn main() -> irc::error::Result<()> {
    let config = Config { ... };
    let mut reactor = IrcReactor::new()?;
    let client = reactor.prepare_client_and_connect(&config)?;
    client.identify()?;

    reactor.register_client_with_handler(client, |client, message| {
        print!("{}", message);
        Ok(())
    });

    reactor.run()?;
}
```

A PROGRAM IN RUST

```rust
extern crate irc;
use irc::client::prelude::*;

fn main() -> irc::error::Result<()> {
    let config = Config { ... };
    let mut reactor = IrcReactor::new()?;
    let client = reactor.prepare_client_and_connect(&config)?;
    client.identify()?;

    reactor.register_client_with_handler(client, |client, message| {
        print!("{}", message);
        Ok(())
    });

    reactor.run()?;
}
```

A PROGRAM IN RUST

```rust
extern crate irc;
use irc::client::prelude::*;

fn main() -> irc::error::Result<()> {
    let config = Config { ... };
    let mut reactor = IrcReactor::new()?;
    let client = reactor.prepare_client_and_connect(&config)?;
    client.identify()?;

    reactor.register_client_with_handler(client, |client, message| {
        print!("{}", message);
        Ok(())
    });

    reactor.run()?;
}
```

A PROGRAM IN RUST

```rust
extern crate irc;
use irc::client::prelude::*;

fn main() -> irc::error::Result<()> {
    let config = Config { ... };
    let mut reactor = IrcReactor::new()?;
    let client = reactor.prepare_client_and_connect(&config)?;
    client.identify()?;

    reactor.register_client_with_handler(client, |client, message| {
        print!("{}", message);
        Ok(())
    });

    reactor.run()?;
}
```

A PROGRAM IN RUST

```rust
extern crate irc;
use irc::client::prelude::*;

fn main() -> irc::error::Result<()> {
    let config = Config { ... };
    let mut reactor = IrcReactor::new()?;
    let client = reactor.prepare_client_and_connect(&config)?;
    client.identify()?;

    reactor.register_client_with_handler(client, |client, message| {
        print!("{}", message);
        Ok(())
    });

    reactor.run()?;
}
```

A PROGRAM IN RUST

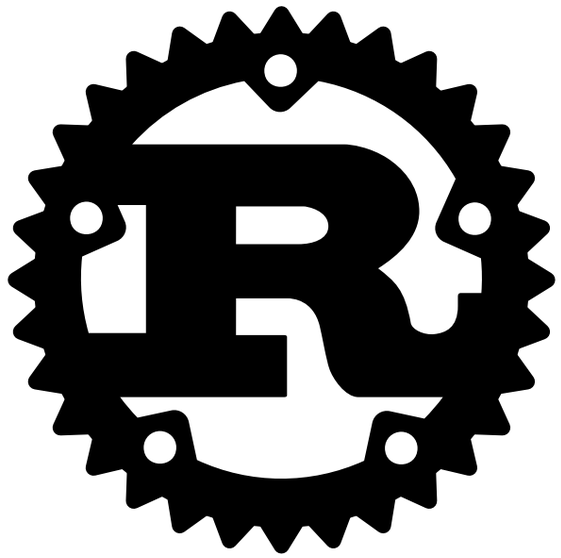
```rust
extern crate irc;
use irc::client::prelude::*;

fn main() -> irc::error::Result<()> {
    let config = Config { ... };
    let mut reactor = IrcReactor::new()?;
    let client = reactor.prepare_client_and_connect(&config)?;
    client.identify()?;

    reactor.register_client_with_handler(client, |client, message| {
        print!("{}", message);
        Ok(())
    });

    reactor.run()?;
}
```

γεια

ελλάδα

THE CURRENT STATE OF AFFAIRS

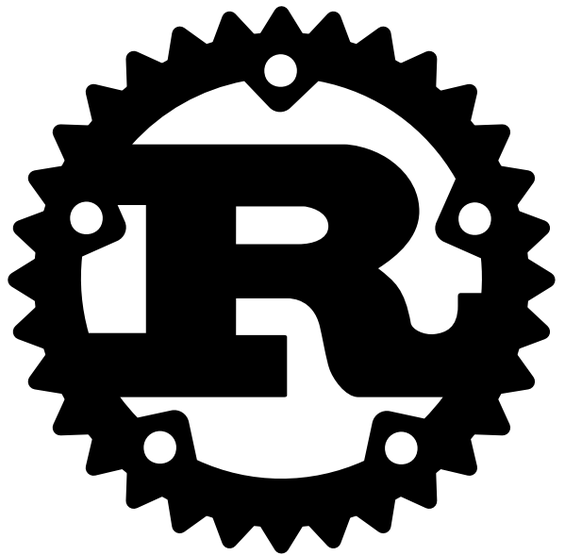
THE CURRENT STATE OF AFFAIRS



RUST

interprocedural static analysis
with *ad-hoc* constraint solving

THE CURRENT STATE OF AFFAIRS



RUST

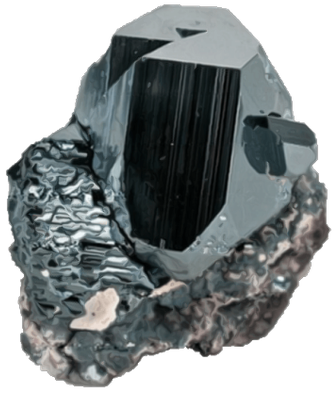
interprocedural static analysis
with *ad-hoc* constraint solving

RUSTBELT (JUNG, JOURDAN, KREBBERS, AND DREYER, POPL '18)

formal language specified in *Iris*
but low-level, in a *CPS-style*.

BUT WE WANT TO DO BETTER

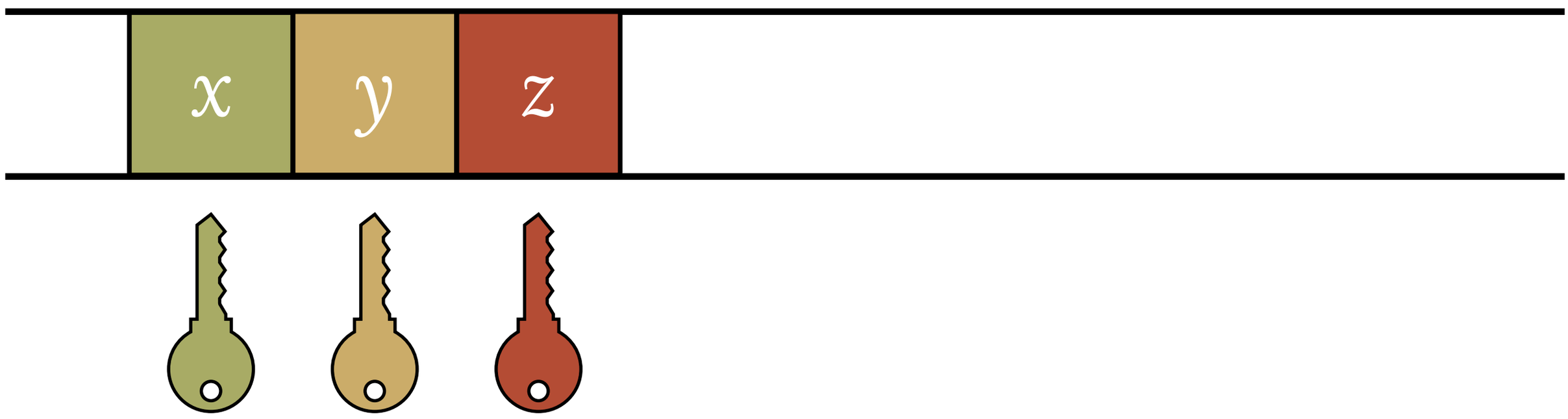
BUT WE WANT TO DO BETTER



Oxide

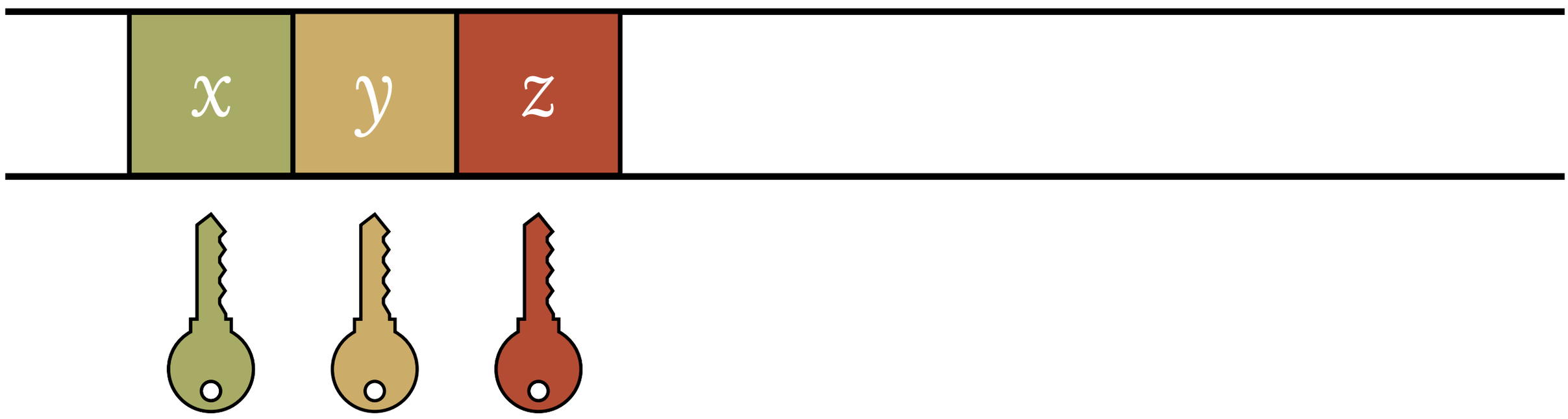
CAPABILITIES FOR OWNERSHIP

CAPABILITIES FOR OWNERSHIP

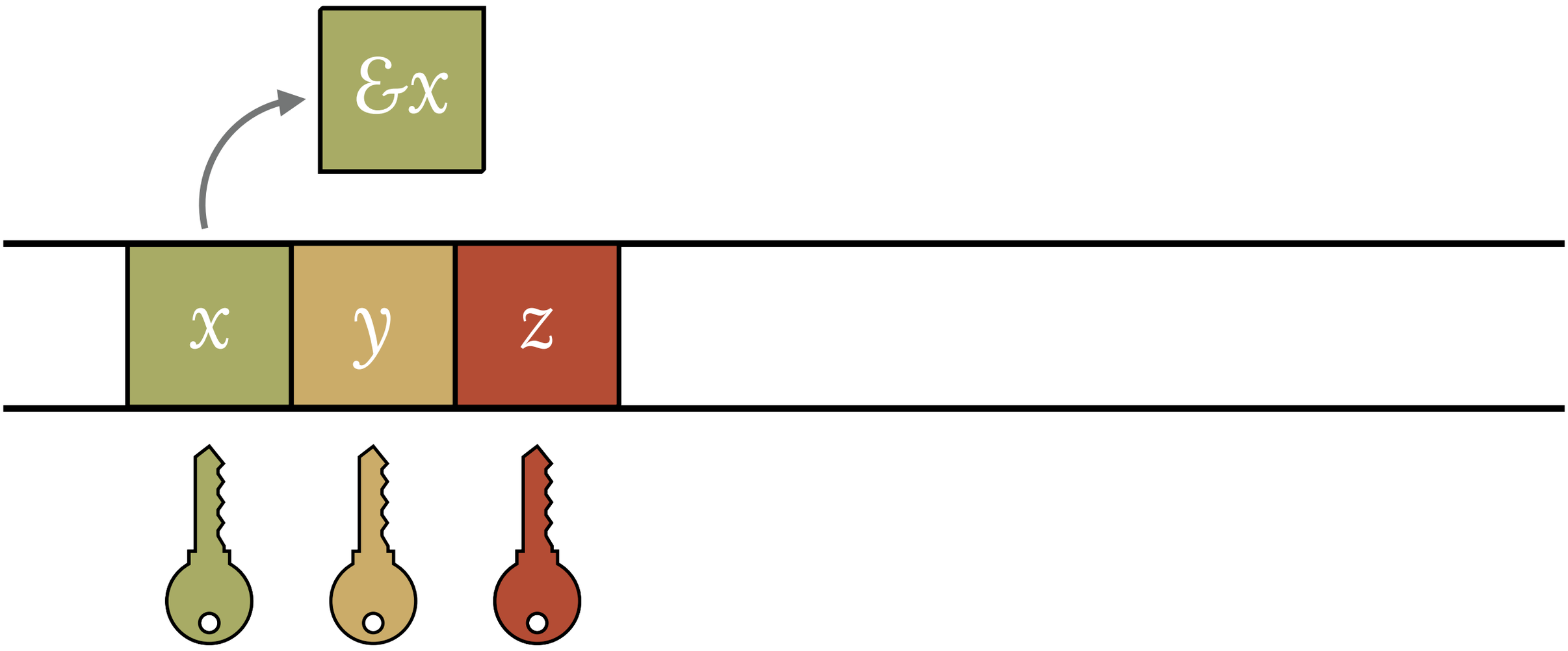


capabilities guard the use of identifiers

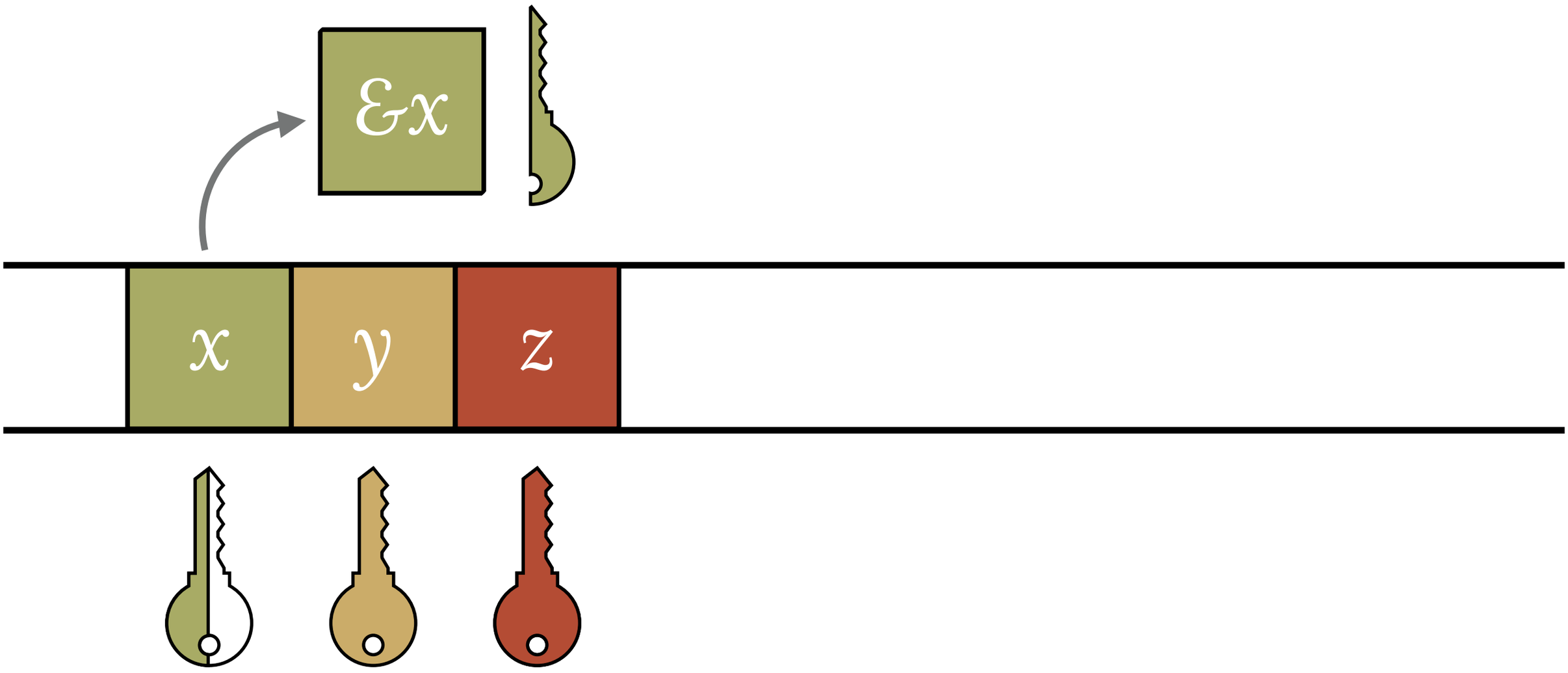
BORROWS BREAK CAPABILITIES INTO FRACTIONS



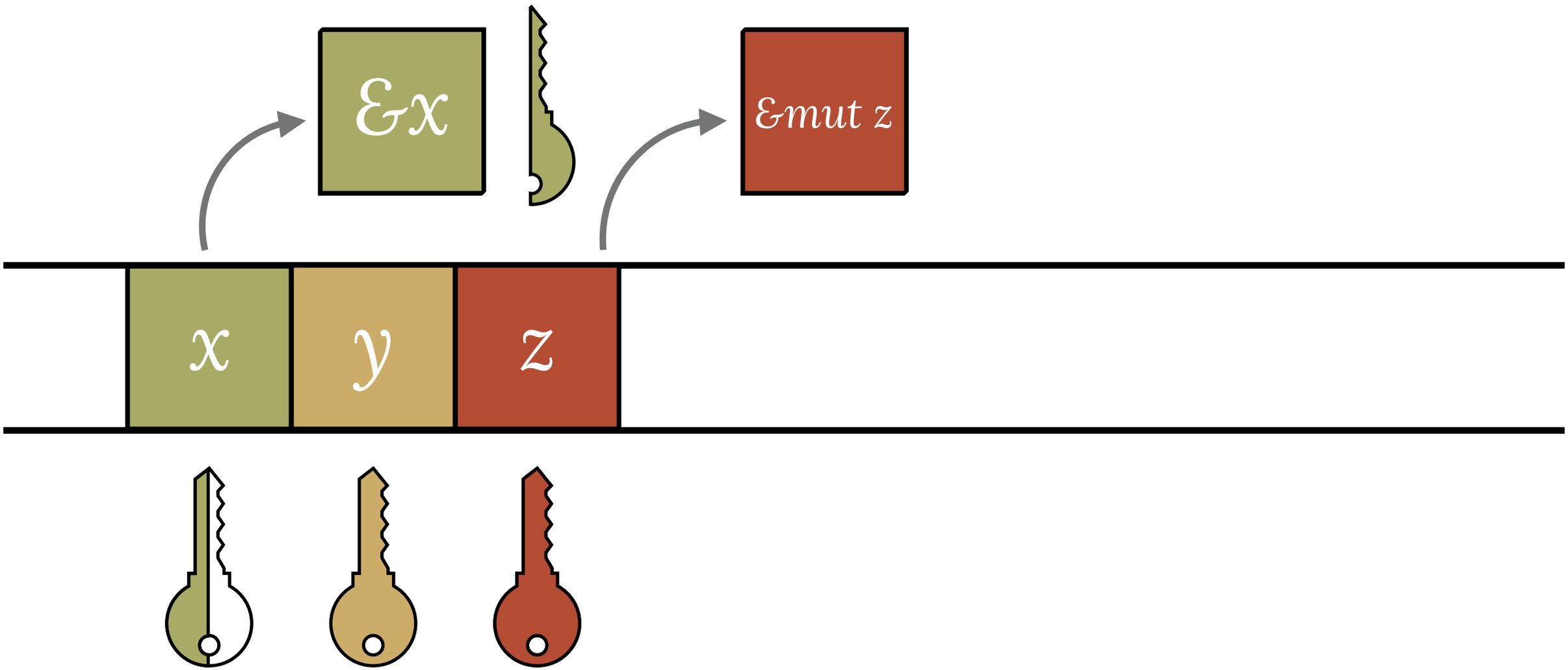
BORROWS BREAK CAPABILITIES INTO FRACTIONS



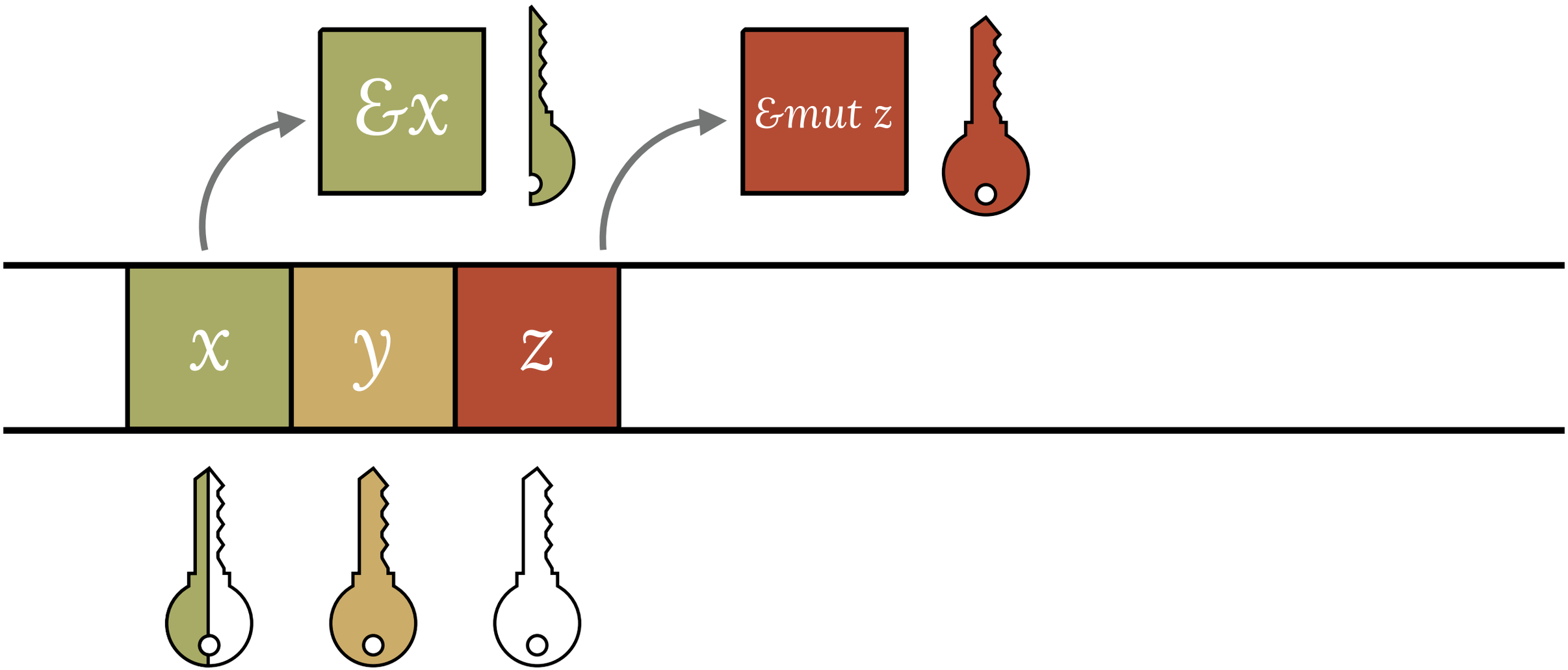
BORROWS BREAK CAPABILITIES INTO FRACTIONS



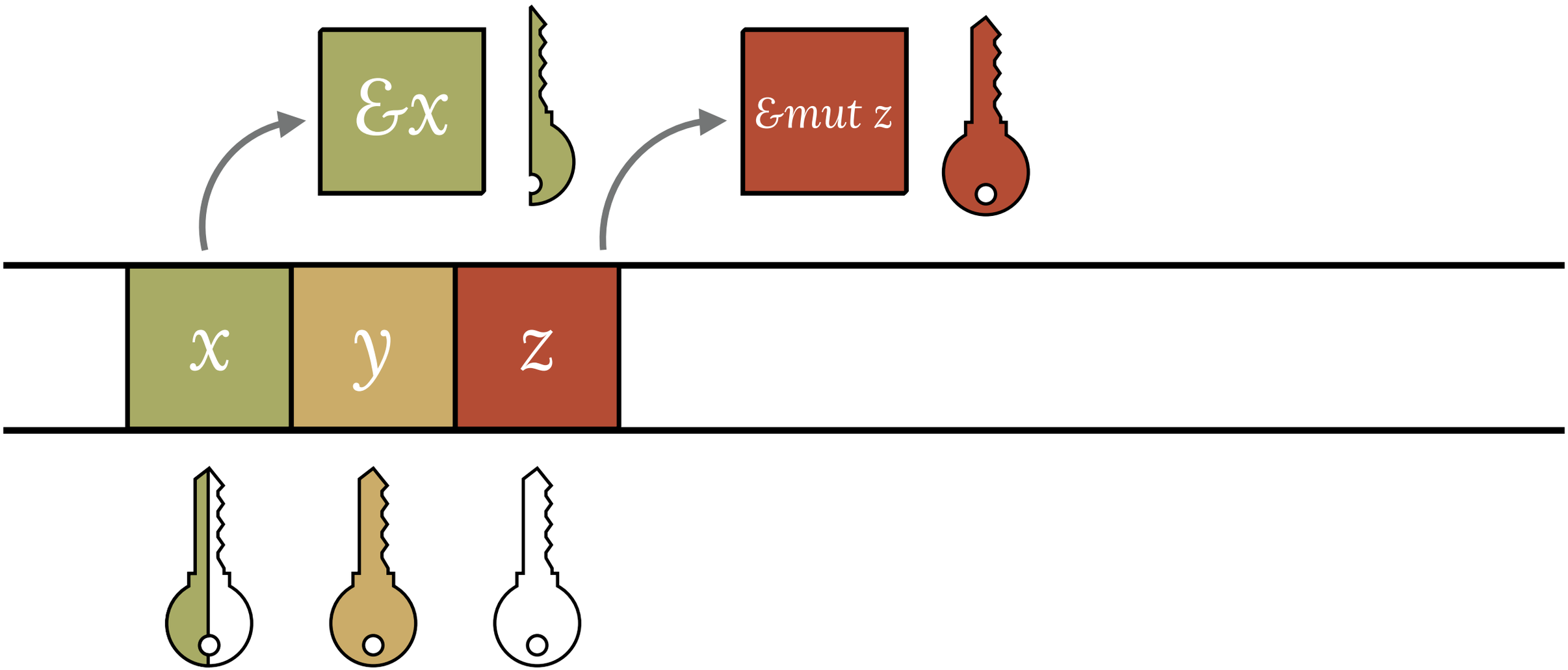
BORROWS BREAK CAPABILITIES INTO FRACTIONS



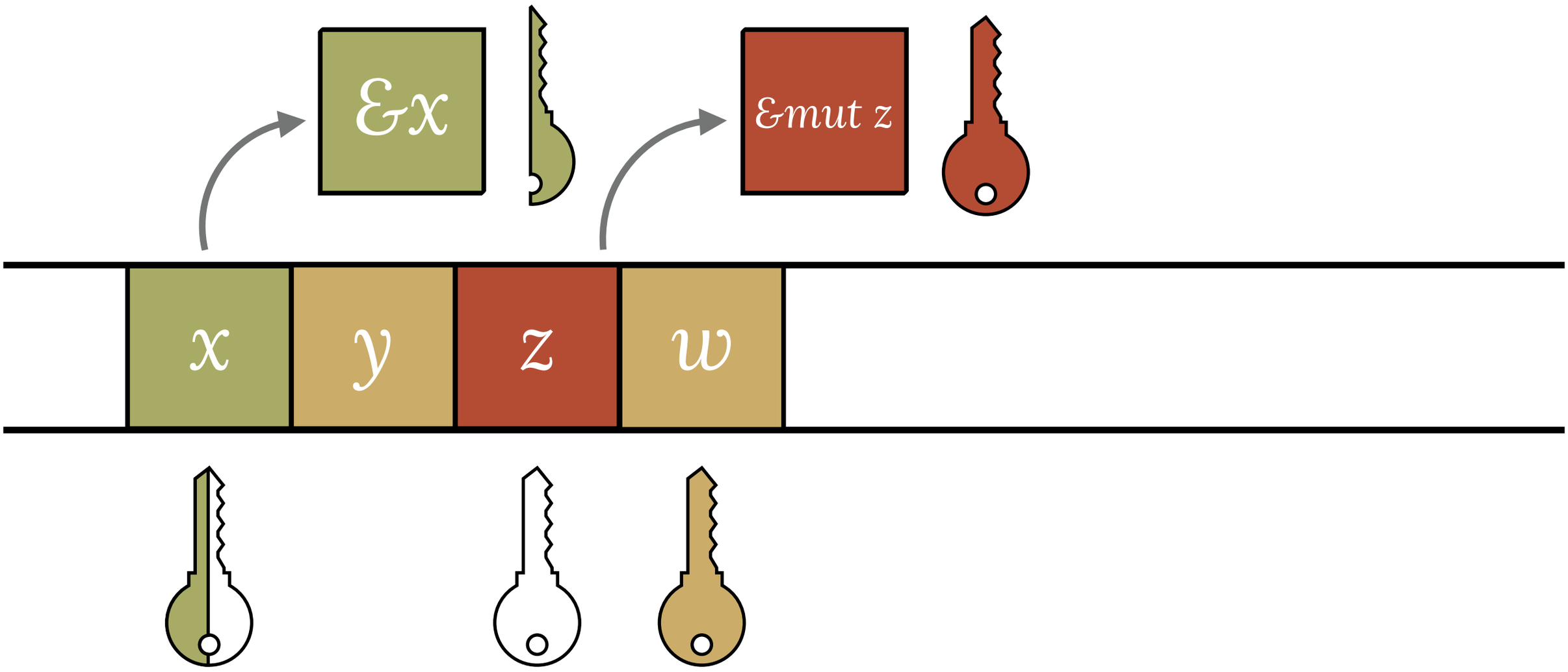
BORROWS BREAK CAPABILITIES INTO FRACTIONS



MOVES TAKE THE CAPABILITY AND THE HOLE



MOVES TAKE THE CAPABILITY AND THE HOLE



WE CALL REFERENCE SITES LOANS

```
extern crate irc;
use irc::client::prelude::*;

fn main() -> irc::error::Result<()> {
    let config = Config { ... };
    let mut reactor = IrcReactor::new()?;
    let client = reactor.prepare_client_and_connect(&config)?;
    client.identify()?;

    reactor.register_client_with_handler(client, |client, message| {
        print!("{}", message);
        Ok(())
    });

    reactor.run()?;
}
```

a loan

WHAT ABOUT LIFETIMES?

WHAT ABOUT LIFETIMES?

```
x : u32
```

WHAT ABOUT LIFETIMES?

```
x : u32

&x : &'x u32
```

WHAT ABOUT LIFETIMES?

```
x : u32

&x : &'x u32
```

To keep type-checking tractable, regions correspond to sets of loans.

TYPE CHECKING

TYPE CHECKING

global context

$$\Sigma$$

TYPE CHECKING

global context

$$\Sigma; \Delta$$

type variable context

TYPE CHECKING

global context

variable context

$$\Sigma; \Delta; \Gamma$$

type variable context

TYPE CHECKING

global context

variable context

$$\Sigma; \Delta; \Gamma; \mathcal{L}$$

loan context

type variable context

TYPE CHECKING

global context

variable context

region context

$$\Sigma; \Delta; \Gamma; \mathcal{L}; P$$

loan context

type variable context

TYPE CHECKING

global context

variable context

region context

$$\Sigma; \Delta; \Gamma; \mathcal{L}; P \vdash e$$

loan context

type variable context

TYPE CHECKING

global context

variable context

region context

$$\Sigma; \Delta; \Gamma; \mathcal{L}; P \vdash e : \tau$$

loan context

type variable context

TYPE CHECKING

global context

variable context

region context

$$\Sigma; \Delta; \Gamma; \mathcal{L}; P \vdash e : \tau \Rightarrow \varepsilon$$

loan context

ownership effects

type variable context

TYPING BORROWS

$$\Sigma; \Delta; \Gamma, x :_f \tau; \mathcal{L}; P \vdash \texttt{\&'a x} :$$

TYPING BORROWS

$$\frac{f \neq 0}{\Sigma; \Delta; \Gamma, x :_f \tau; \mathcal{L}; P \vdash \texttt{\&'a x} :}$$

TYPING BORROWS

$$\frac{f \neq 0}{\Sigma; \Delta; \Gamma, x :_f \tau; \mathcal{L}; P \vdash \texttt{\&'a x : \&\{'a\}}\ \tau}$$

TYPING BORROWS

$$\frac{f \neq 0}{\Sigma; \Delta; \Gamma, x :_f \tau; \mathcal{L}; P \vdash \texttt{\&'a x : \&\{'a\}}\ \tau}$$

$\Rightarrow$ `borrow imm x as 'a`

TYPING BRANCHING

$$\Sigma; \Delta; \Gamma; \mathcal{L}; P \vdash \texttt{if}\ e_1\ \{e_2\}\ \texttt{else}\ \{e_3\} :$$

TYPING BRANCHING

$$\Sigma; \Delta; \Gamma; \mathcal{L}; P \vdash e_1 : \texttt{bool} \Rightarrow \varepsilon_1$$

$$\Sigma; \Delta; \Gamma; \mathcal{L}; P \vdash \texttt{if}\ e_1\ \{e_2\}\ \texttt{else}\ \{e_3\} :$$

TYPING BRANCHING

$$\Sigma; \Delta; \Gamma; \mathcal{L}; P \vdash e_1 : \texttt{bool} \Rightarrow \varepsilon_1$$
$$\Sigma; \Delta; \varepsilon_1(\Gamma); \varepsilon_1(\mathcal{L}); \varepsilon_1(P) \vdash e_2 : \tau_2 \Rightarrow \varepsilon_2$$

$$\Sigma; \Delta; \Gamma; \mathcal{L}; P \vdash \texttt{if}\ e_1\ \{e_2\}\ \texttt{else}\ \{e_3\} :$$

TYPING BRANCHING

$$\Sigma; \Delta; \Gamma; \mathcal{L}; P \vdash e_1 : \texttt{bool} \Rightarrow \varepsilon_1$$
$$\Sigma; \Delta; \varepsilon_1(\Gamma); \varepsilon_1(\mathcal{L}); \varepsilon_1(P) \vdash e_2 : \tau_2 \Rightarrow \varepsilon_2$$
$$\Sigma; \Delta; \varepsilon_1(\Gamma); \varepsilon_1(\mathcal{L}); \varepsilon_1(P) \vdash e_3 : \tau_3 \Rightarrow \varepsilon_3$$

$$\Sigma; \Delta; \Gamma; \mathcal{L}; P \vdash \texttt{if}\ e_1\ \{e_2\}\ \texttt{else}\ \{e_3\} :$$

TYPING BRANCHING

$$\frac{\begin{array}{c}\Sigma; \Delta; \Gamma; \mathcal{L}; P \vdash e_1 : \texttt{bool} \Rightarrow \varepsilon_1 \\ \Sigma; \Delta; \varepsilon_1(\Gamma); \varepsilon_1(\mathcal{L}); \varepsilon_1(P) \vdash e_2 : \tau_2 \Rightarrow \varepsilon_2 \\ \Sigma; \Delta; \varepsilon_1(\Gamma); \varepsilon_1(\mathcal{L}); \varepsilon_1(P) \vdash e_3 : \tau_3 \Rightarrow \varepsilon_3 \\ \tau_2 \sim \tau_3 \Rightarrow \tau \end{array}}{\Sigma; \Delta; \Gamma; \mathcal{L}; P \vdash \texttt{if}\ e_1\ \{e_2\}\ \texttt{else}\ \{e_3\} :}$$

TYPING BRANCHING

$$\frac{\begin{array}{c}\Sigma; \Delta; \Gamma; \mathcal{L}; P \vdash e_1 : \texttt{bool} \Rightarrow \varepsilon_1 \\ \Sigma; \Delta; \varepsilon_1(\Gamma); \varepsilon_1(\mathcal{L}); \varepsilon_1(P) \vdash e_2 : \tau_2 \Rightarrow \varepsilon_2 \\ \Sigma; \Delta; \varepsilon_1(\Gamma); \varepsilon_1(\mathcal{L}); \varepsilon_1(P) \vdash e_3 : \tau_3 \Rightarrow \varepsilon_3 \\ \tau_2 \sim \tau_3 \Rightarrow \tau\end{array}}{\Sigma; \Delta; \Gamma; \mathcal{L}; P \vdash \texttt{if}\ e_1\ \{e_2\}\ \texttt{else}\ \{e_3\} : \tau}$$

TYPING BRANCHING

$$\frac{\begin{array}{c}\Sigma; \Delta; \Gamma; \mathcal{L}; P \vdash e_1 : \texttt{bool} \Rightarrow \varepsilon_1 \\ \Sigma; \Delta; \varepsilon_1(\Gamma); \varepsilon_1(\mathcal{L}); \varepsilon_1(P) \vdash e_2 : \tau_2 \Rightarrow \varepsilon_2 \\ \Sigma; \Delta; \varepsilon_1(\Gamma); \varepsilon_1(\mathcal{L}); \varepsilon_1(P) \vdash e_3 : \tau_3 \Rightarrow \varepsilon_3 \\ \tau_2 \sim \tau_3 \Rightarrow \tau\end{array}}{\Sigma; \Delta; \Gamma; \mathcal{L}; P \vdash \texttt{if}\ e_1\ \{e_2\}\ \texttt{else}\ \{e_3\} : \tau \Rightarrow \varepsilon_1, \varepsilon_2, \varepsilon_3}$$

WE MODEL...

WE MODEL...

```
struct Point { x: u32, y: u32 }
```

WE MODEL...

```
struct Point { x: u32, y: u32 }

enum Option<T> { Some(T), None }
```

WE MODEL...

```
struct Point { x: u32, y: u32 }

enum Option<T> { Some(T), None }
```

if e_1 { e_2 } else { e_3 }

WE MODEL...

```
struct Point { x: u32, y: u32 }

enum Option<T> { Some(T), None }

if e1 { e2 } else { e3 }

|x: u32| { x + x }
```

WE MODEL...

```
struct Point { x: u32, y: u32 }

enum Option<T> { Some(T), None }

if e1 {e2} else {e3}

|x: u32| { x + x }

match opt {
  Some(x) ⇒ x,
  None ⇒ 42,
}
```

WE DO NOT MODEL...

WE DO NOT MODEL...

```
trait Read { ... }
```

WE DO NOT MODEL...

```
trait Read { ... }
```

```
Box::new(Counter::new()).count()
```

```
Box::new(Counter::new()).deref().count()
```

WE DO NOT MODEL...

```
trait Read { ... }
```

```
Box::new(Counter::new()).count()
```

```
Box::new(Counter::new()).deref().count()
```

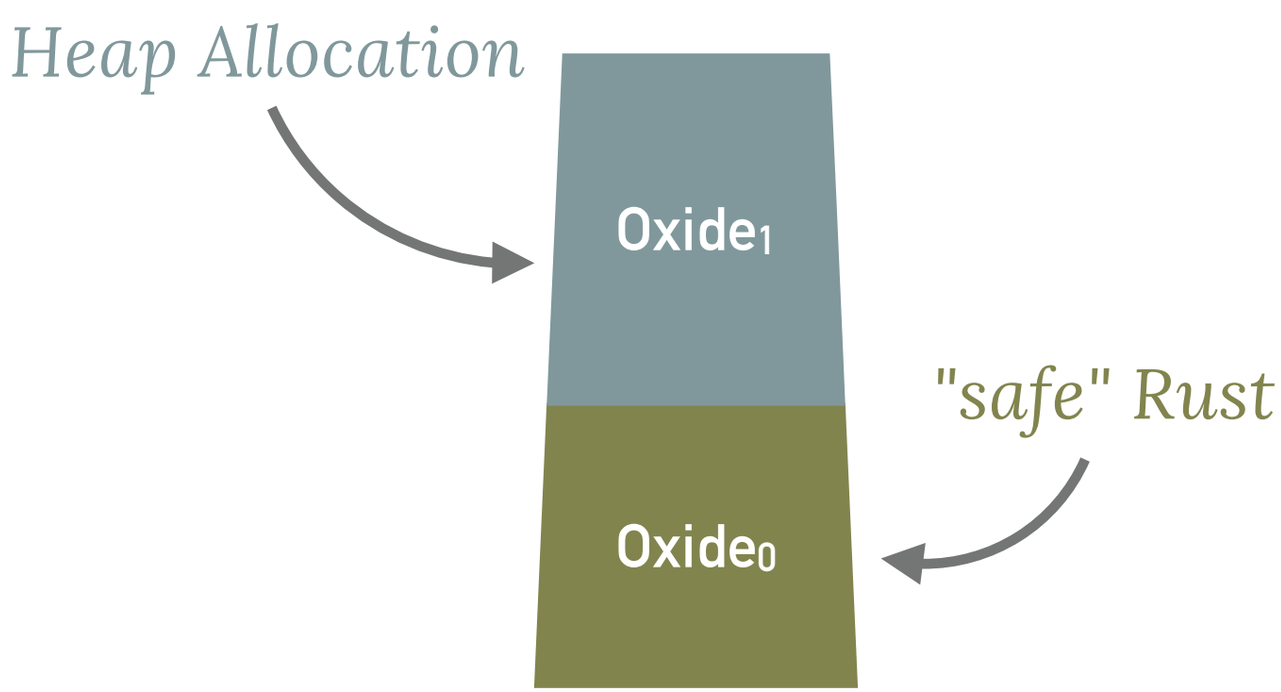
```
(&foo).frobnicate()
```

```
let tmp = &foo;
tmp.frobnicate()
```

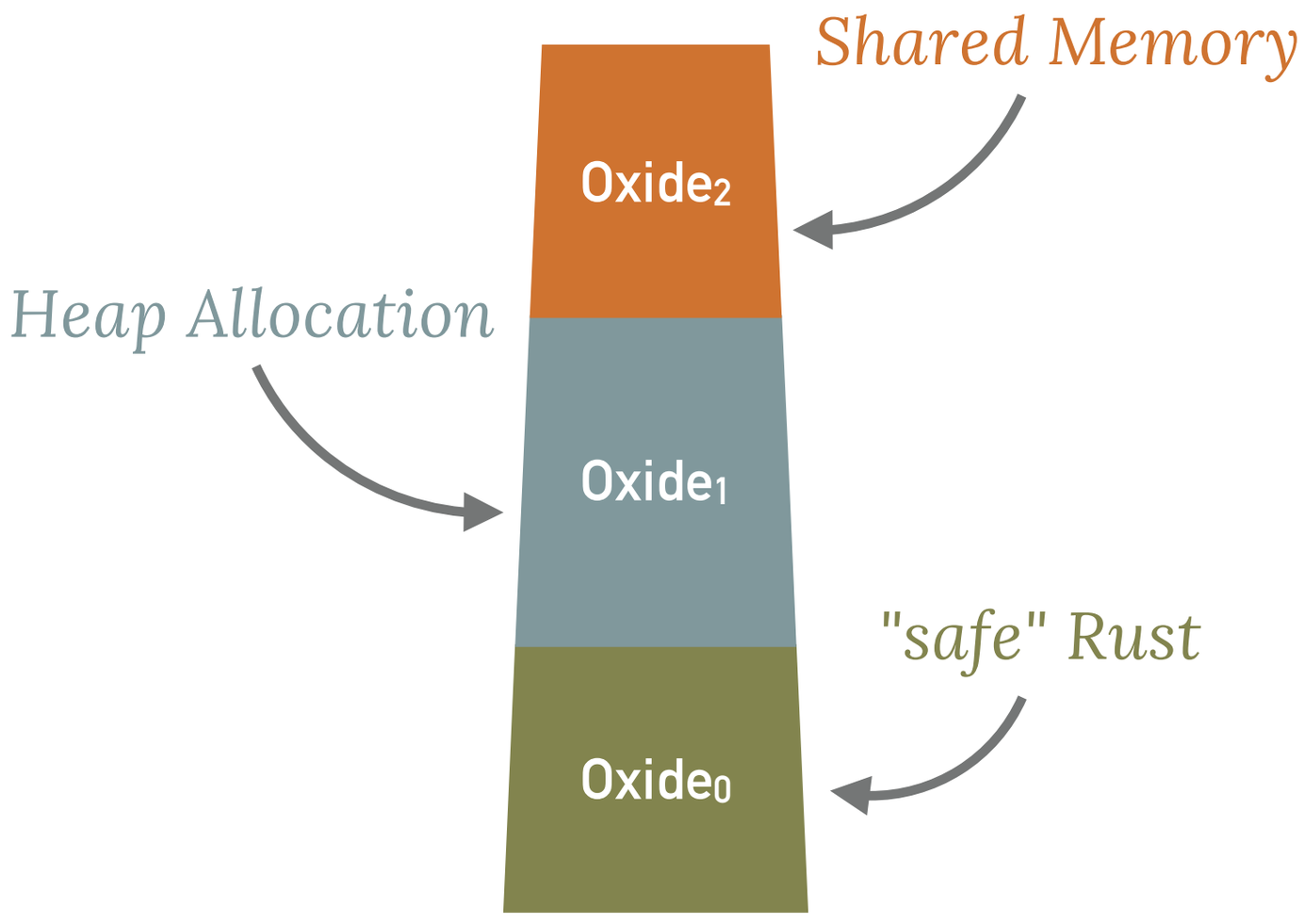
A TOWER OF LANGUAGES

A TOWER OF LANGUAGES

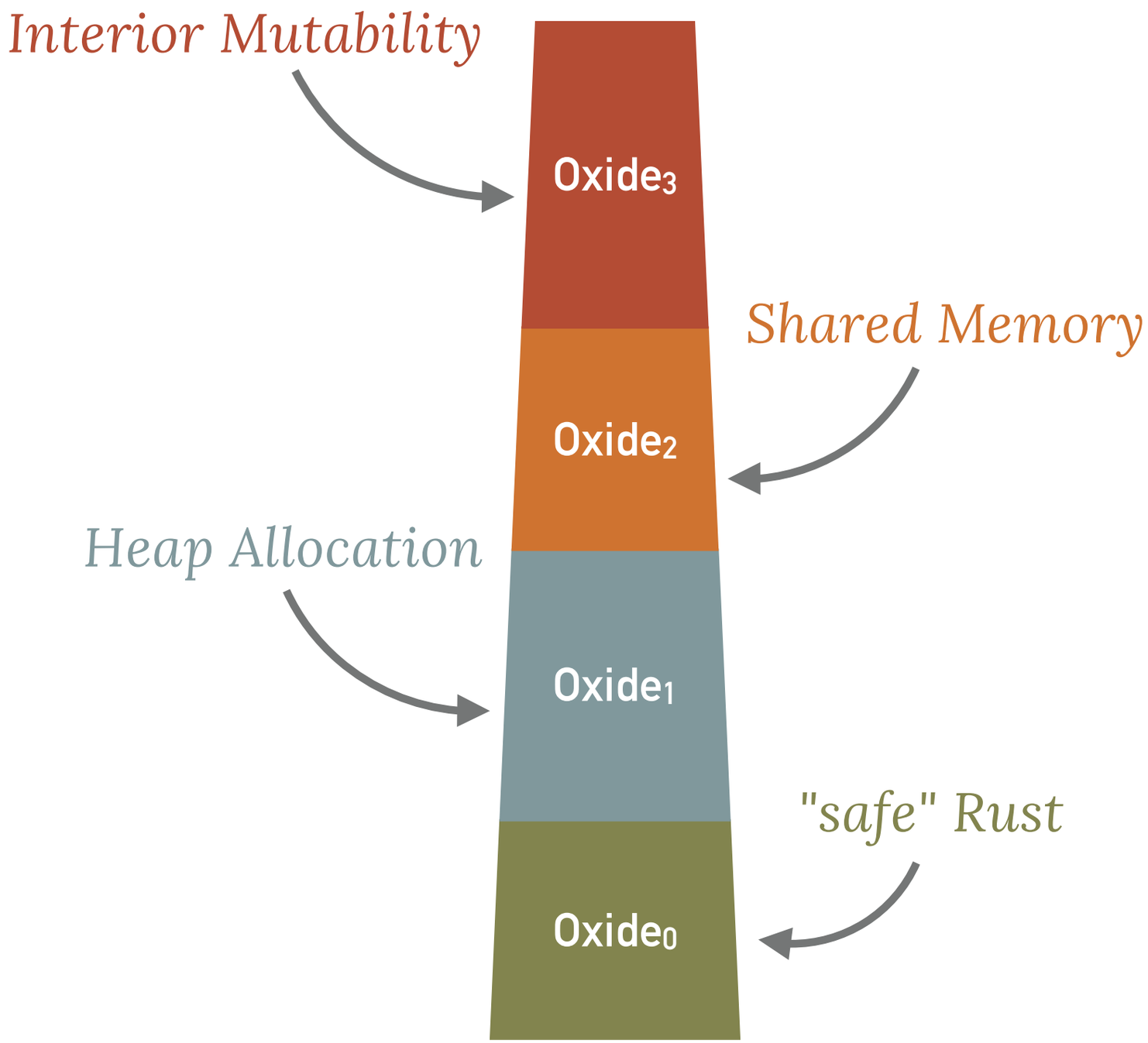
A TOWER OF LANGUAGES



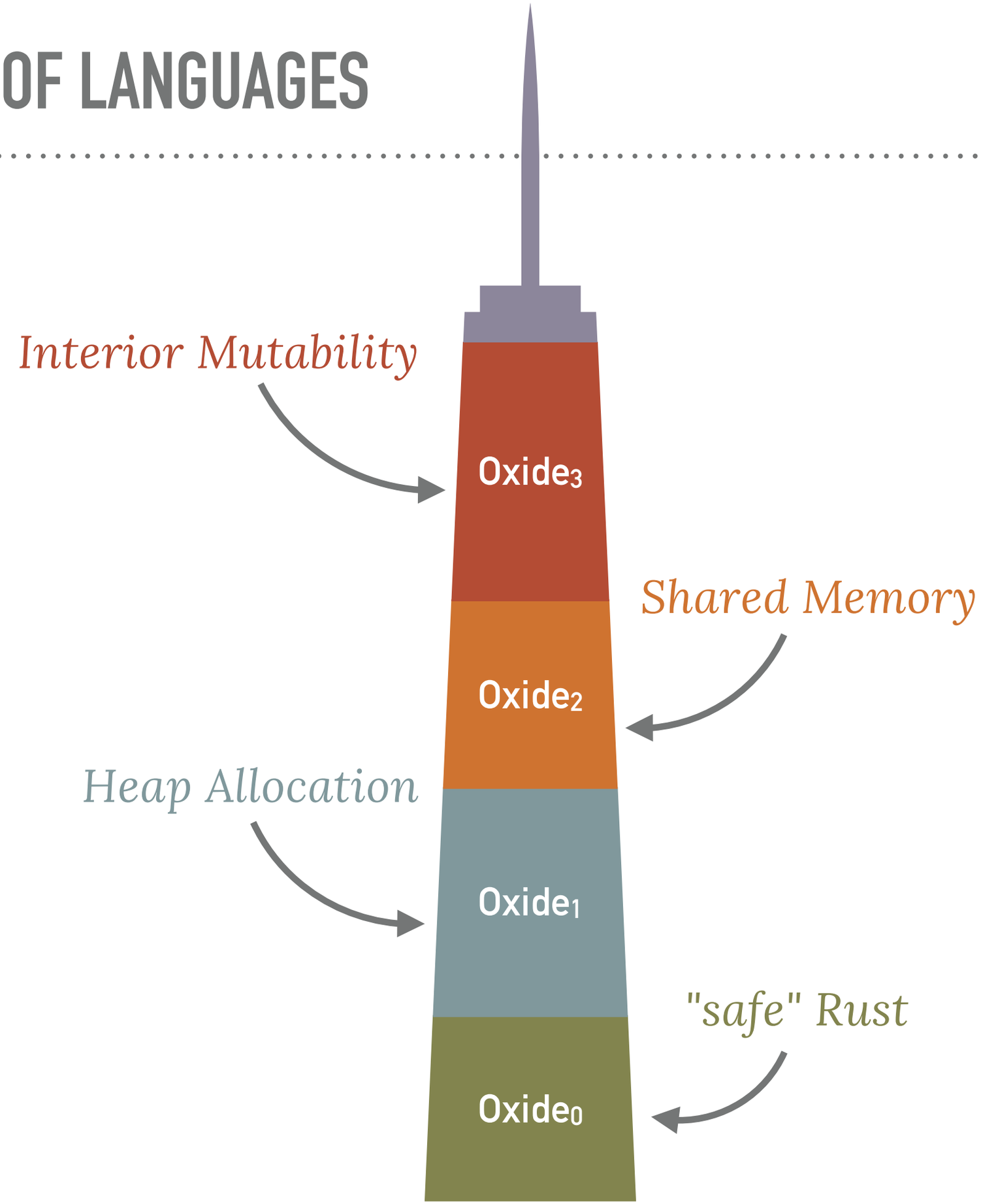
A TOWER OF LANGUAGES



A TOWER OF LANGUAGES



A TOWER OF LANGUAGES



AN EXPRESSIVE TOWER OF EXPRESSIVE POWER

(Felleisen '90)

Expressive power is rooted in observational equivalence.

e_1 e_2

AN EXPRESSIVE TOWER OF EXPRESSIVE POWER

(Felleisen '90)

Expressive power is rooted in observational equivalence.

e_1 e_2

Each new abstraction raises the expressive power
by adding functionality that cannot be observed in lower levels

AN EXPRESSIVE TOWER OF EXPRESSIVE POWER

(Felleisen '90)

Expressive power is rooted in observational equivalence.

$\forall$ C ,

e_1

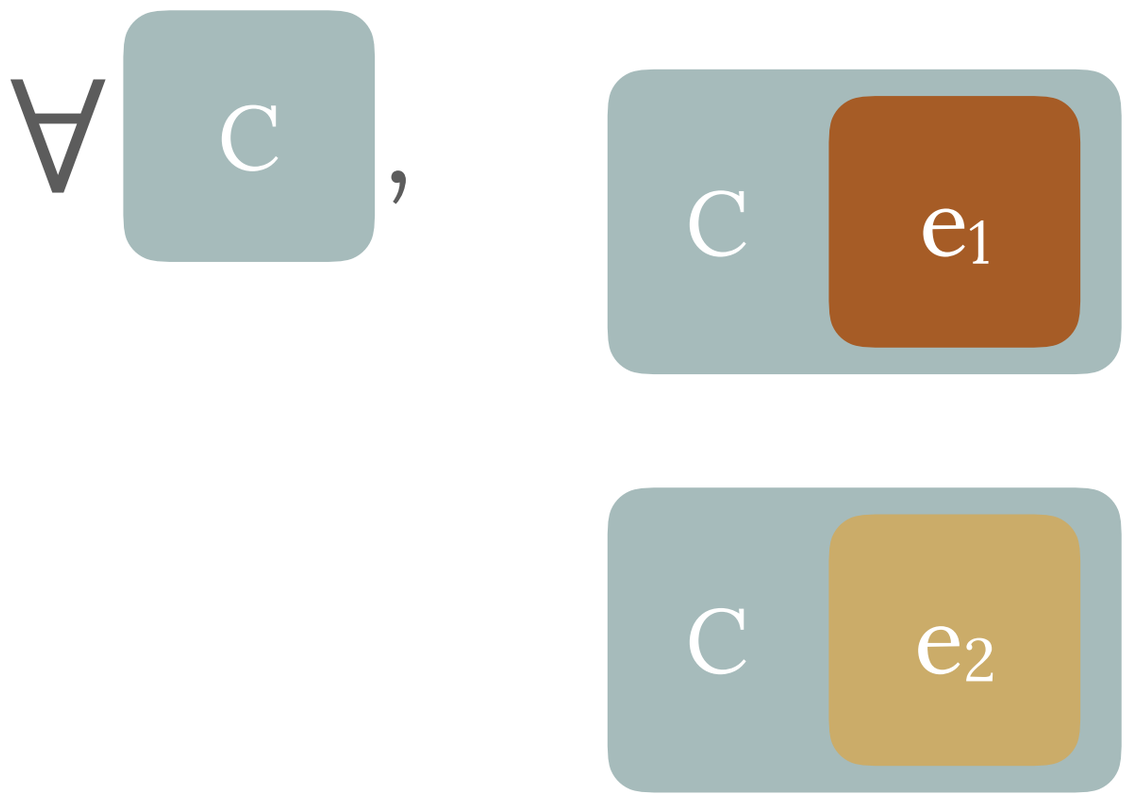
e_2

Each new abstraction raises the expressive power
by adding functionality that cannot be observed in lower levels

AN EXPRESSIVE TOWER OF EXPRESSIVE POWER

(Felleisen '90)

Expressive power is rooted in observational equivalence.



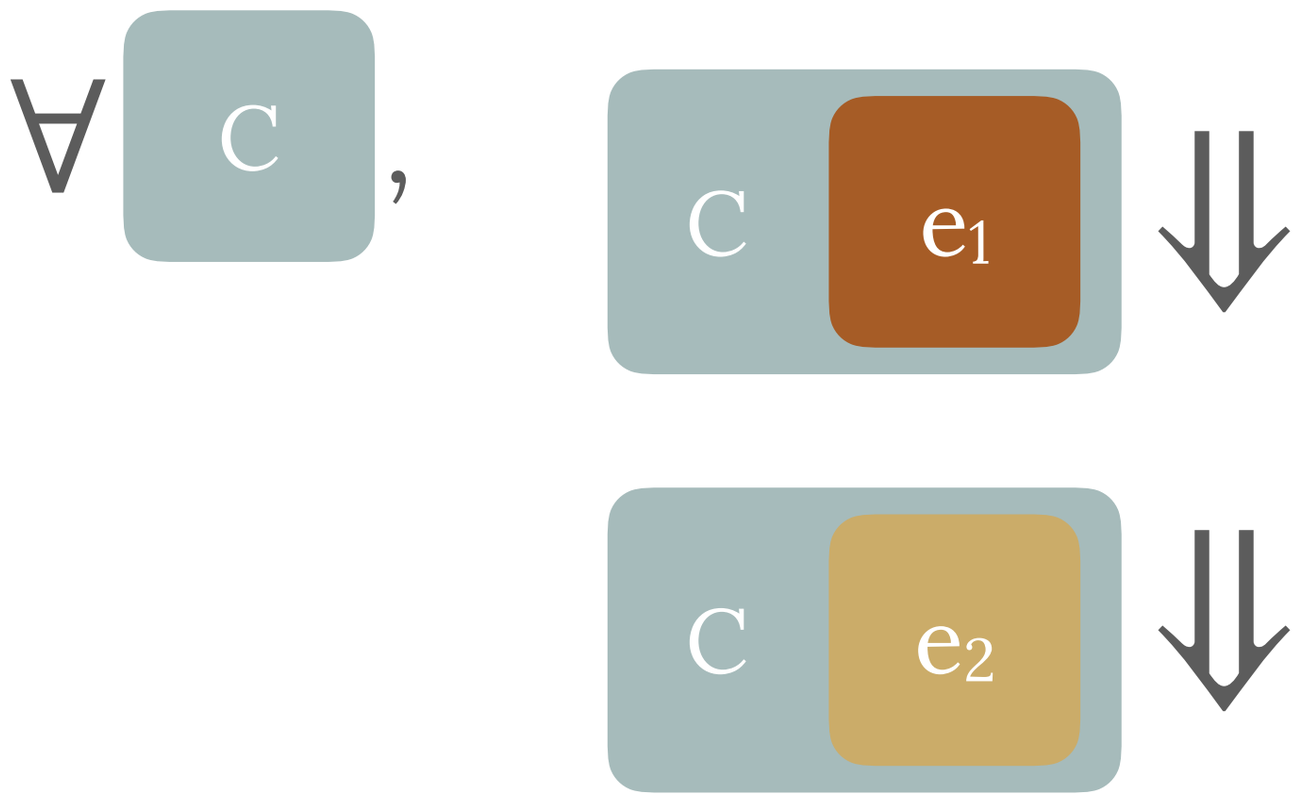
Each new abstraction raises the expressive power

by adding functionality that cannot be observed in lower levels

AN EXPRESSIVE TOWER OF EXPRESSIVE POWER

(Felleisen '90)

Expressive power is rooted in observational equivalence.



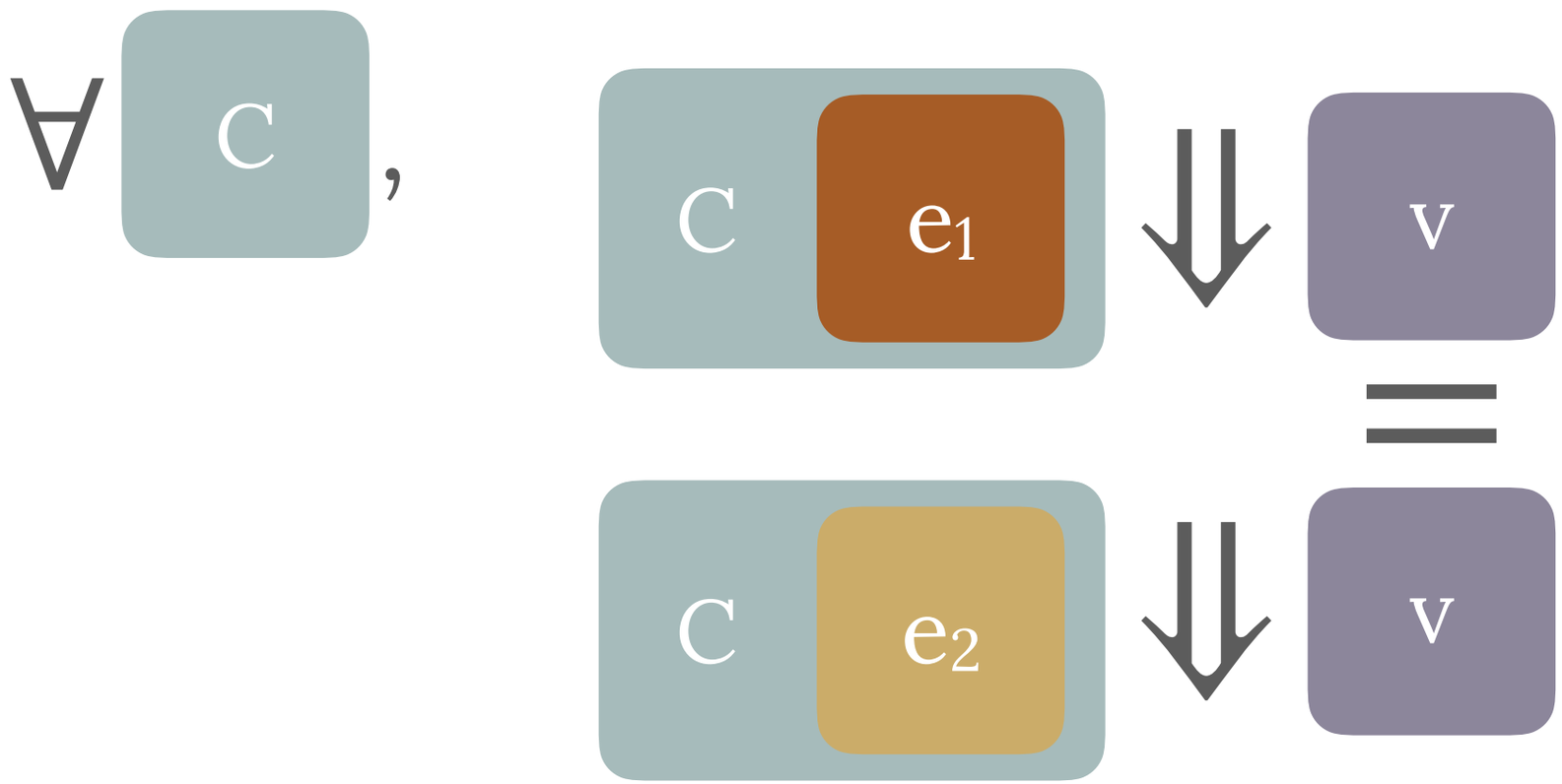
Each new abstraction raises the expressive power

by adding functionality that cannot be observed in lower levels

AN EXPRESSIVE TOWER OF EXPRESSIVE POWER

(Felleisen '90)

Expressive power is rooted in observational equivalence.



Each new abstraction raises the expressive power

by adding functionality that cannot be observed in lower levels

NEXT STEPS

NEXT STEPS

$$\frac{\vdash \text{γεια}}{\vdash \text{ελλάδα}}$$

More formalization...

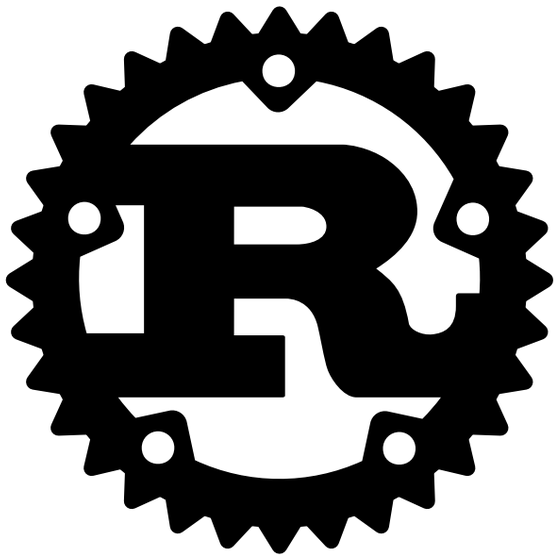
NEXT STEPS

$$\frac{\vdash \text{γεια}}{\vdash \text{ελλάδα}}$$

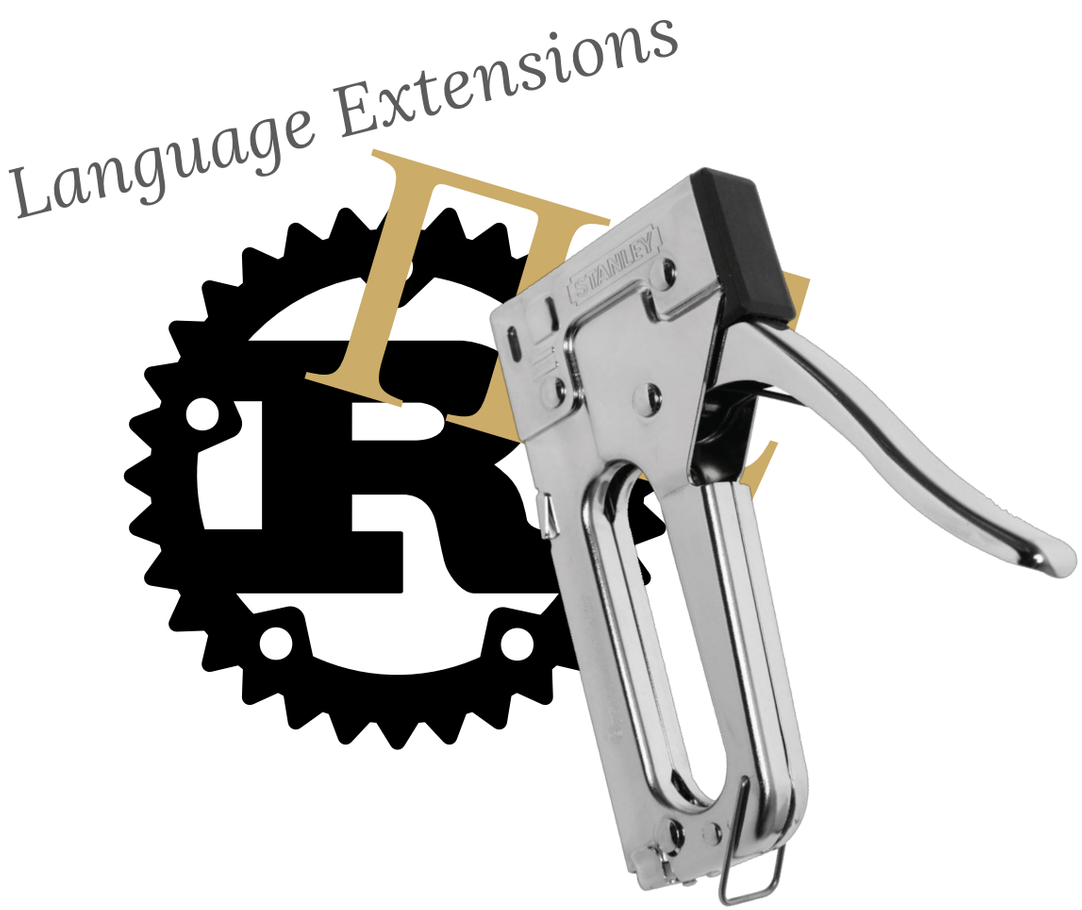
More formalization...

Rust-to-Oxide Compiler

A RUSTY FUTURE

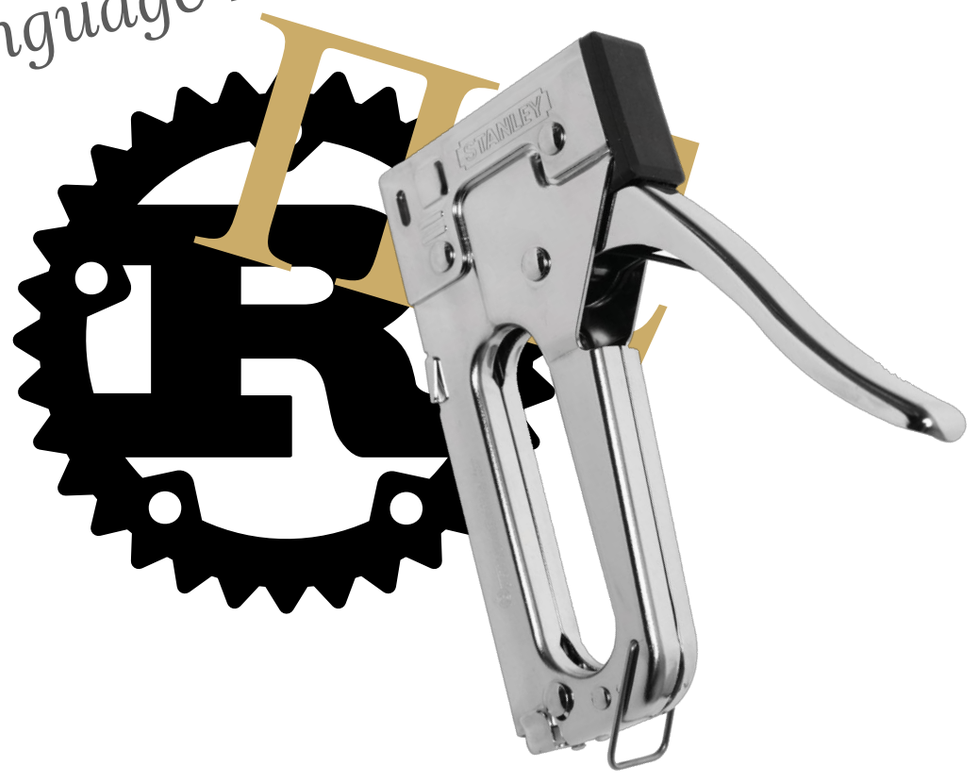


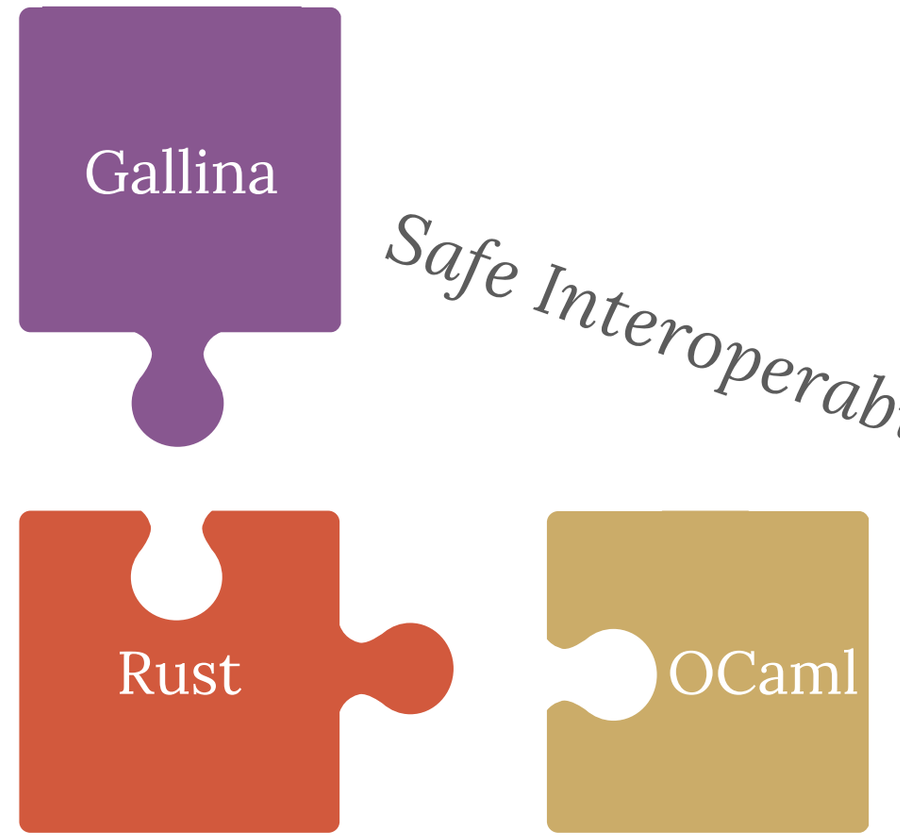
A RUSTY FUTURE



A RUSTY FUTURE

Language Extensions

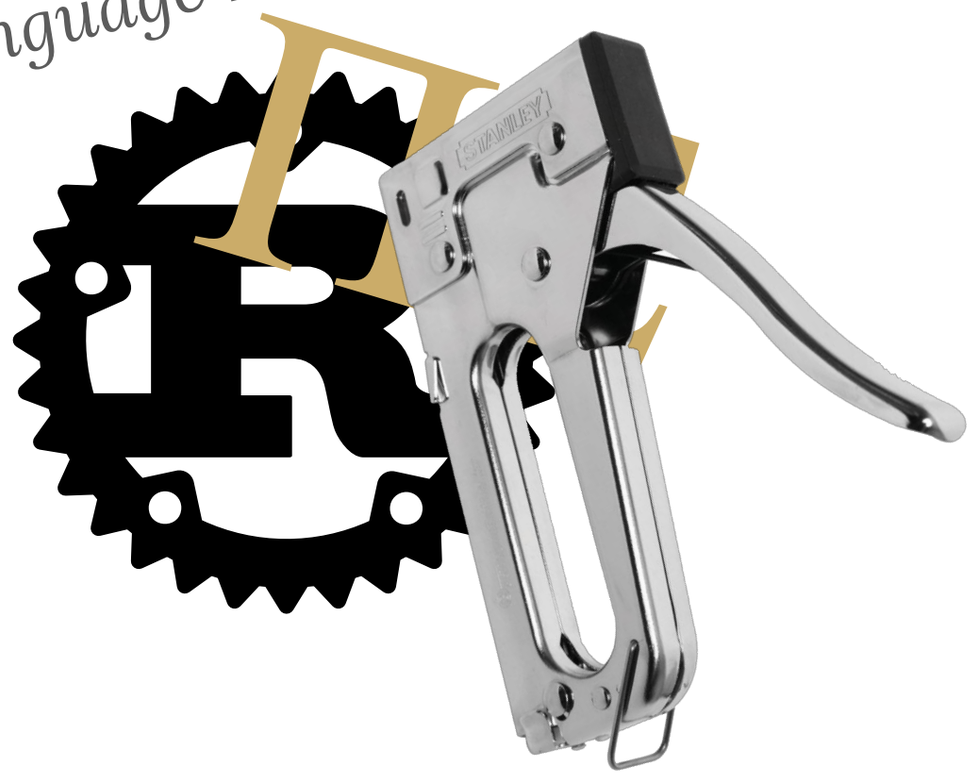




Safe Interoperability

A RUSTY FUTURE

Language Extensions



Gallina

Safe Interoperability

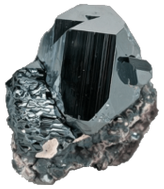
Rust

OCaml

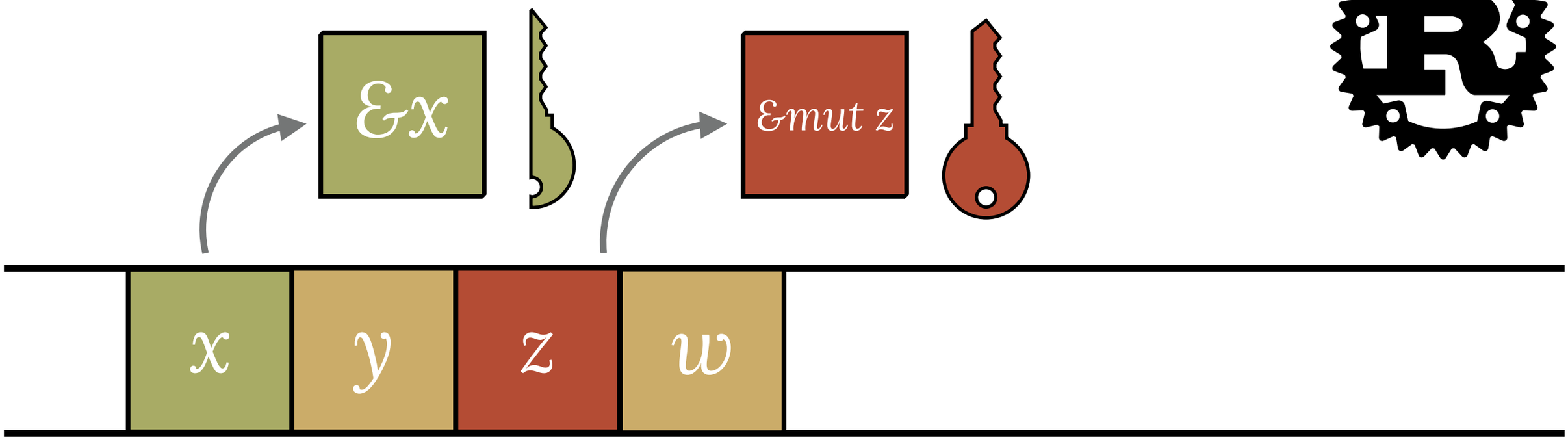
What unsafe code is safe to write?

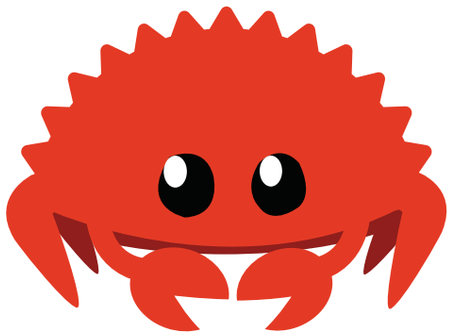
Unsafe Code Guidelines

TAKEAWAYS

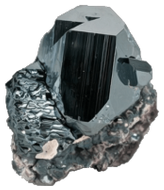


Oxide



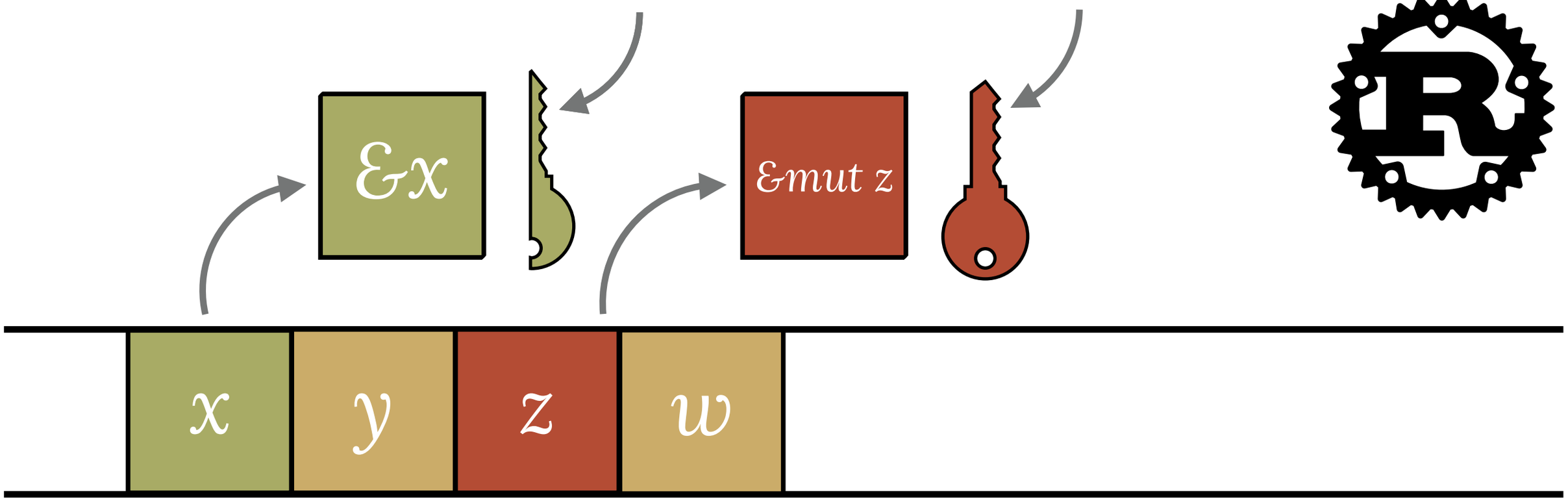


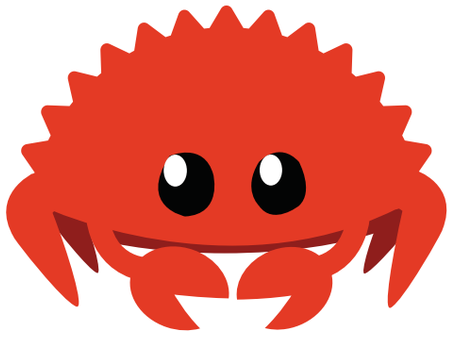
TAKEAWAYS



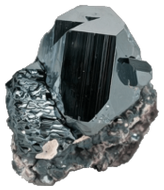
Oxide

Ownership with fractional capabilities



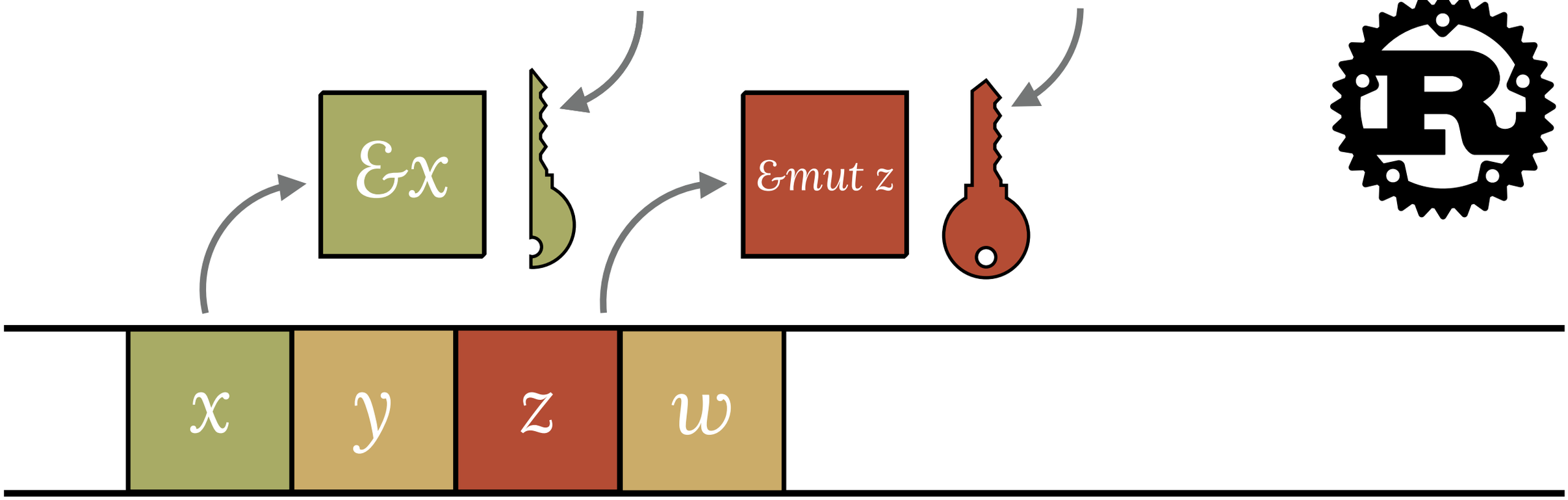


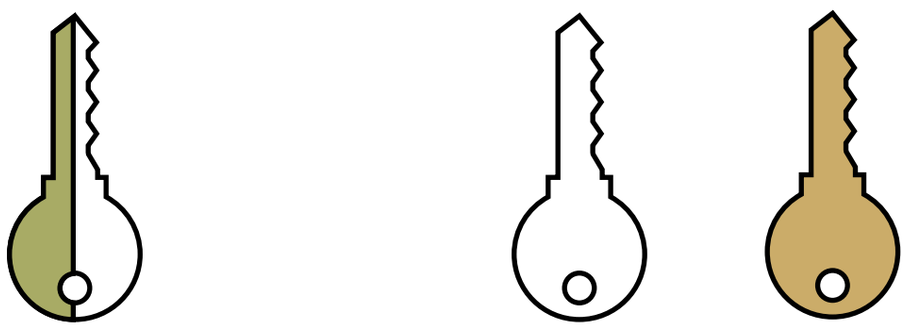
TAKEAWAYS

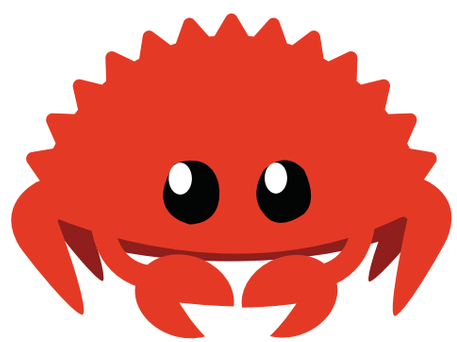


Oxide

Ownership with fractional capabilities

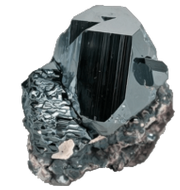




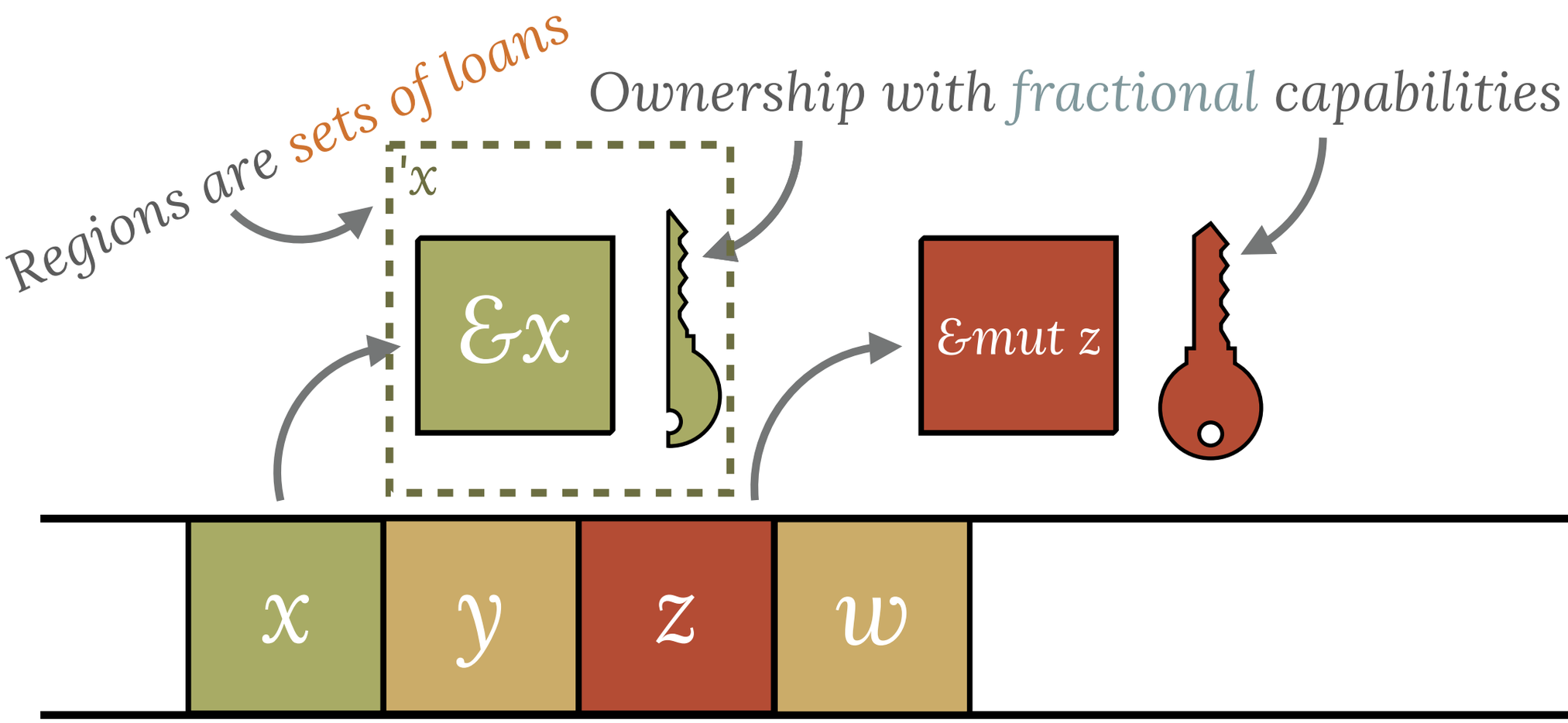


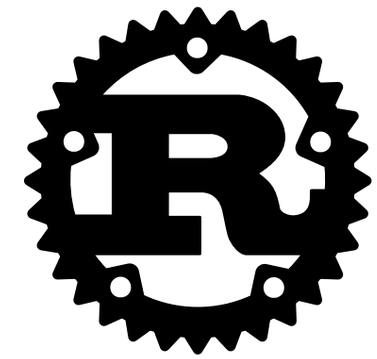
Moves never return their capability

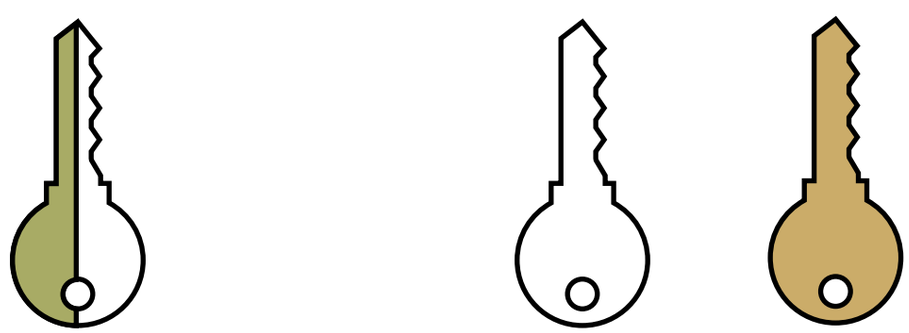
TAKEAWAYS

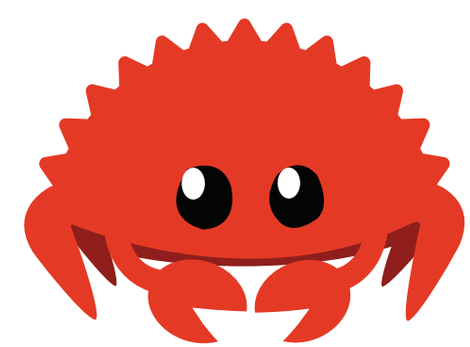


Oxide









Moves never return their capability