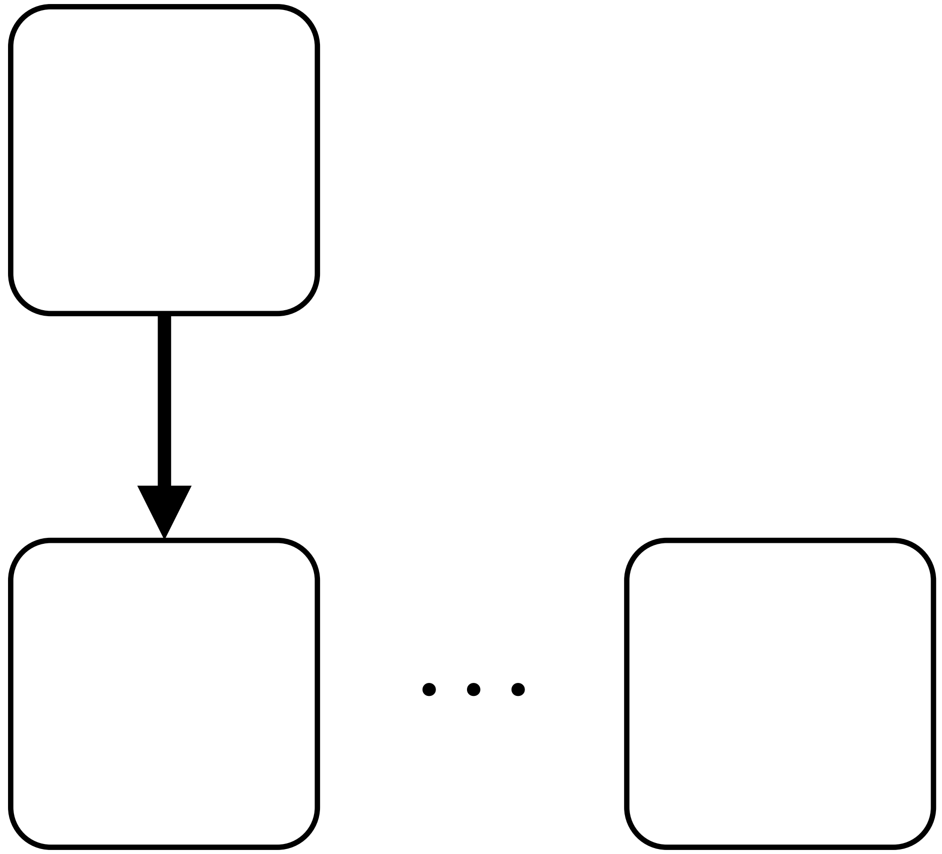
Guaranteeing memory safety in Rust

Nicholas D. Matsakis
Mozilla Research

Hashtable in C / C++

```
template<class K, class V>
struct Hashtable {
    Bucket<K,V> *buckets;
    unsigned num_buckets;
}

template<class K, class V>
struct Bucket {
    bool occupied;
    K key;
    V value;
}
```



Insertion

```
template<class K, class V>
void insert(
  Hashtable<K,V> *table,
  K key,
  V value)
{
  if (table full) {
    table->buckets = realloc(table->buckets, ...);
  }

  unsigned index = find_bucket(table, &key);
  table->buckets[index].key = key;
  table->buckets[index].value = value;
  table->buckets[index].occupied = true;
}
```

Grow arrays as necessary

Initialize the bucket

Assumptions

```
template<class K, class V>
void insert(
  Hashtable<K,V> *table,
  K key,
  V value)
{
  if (table full) {
    table->buckets = realloc(table->buckets, ...);
  }

  unsigned index = find_bucket(table, &key);
  table->buckets[index].key = key;
  table->buckets[index].value = value;
  table->buckets[index].occupied = true;
}
```

Buckets array is unaliased

Never access bucket if occupied is false

Living on the edge

Violating either of these assumptions
can lead to a crash

=>

Crashes lead to exploits.

Privacy?

Q: Doesn't privacy solve this?

A: Not really.

Table lookup

```
template<class K, class V>
V* find(
  Hashtable<K,V> *table,
  K *key)
{
  unsigned index = find_bucket(table, &key);

  if (!table->buckets[index].occupied)
    return NULL;

  return &table->buckets[index].value;
}
```

Find the correct bucket.

Bucket initialized?

Return pointer into the bucket array!

Vulnerability

```
Hashtable<K,V>* table = ...;

V *value = find(table, ...);

insert(table, ...);

use(value);
```

Create alias

Resize bucket array

Dangling pointer

A problem for everyone

Q: Doesn't garbage collection solve this?

A: Not really.

e.g., Java's ConcurrentModificationException

Iteration holds pointers

```
for (std::vector<int>::iterator it = vec.begin();
     it != vec.end();
     ++it)
   std::cout << ' ' << *it;
```

encapsulates a pointer

```
(for-each hashtable (lambda (x) ...))
```

if hashtable is mutable, same problem

Implications

During iteration, the buckets array is aliased.

+

Insertion can resize the array.

=>

Insertion during iteration can allow a foreign website to take over your computer. (e.g., Bug 810718)

Enter: Rust

Think: C++ meets ML/Haskell meets Erlang
(meets Cyclone, ML Kit, and many others)

Credit where credit is due

Rust is the product of a community
too numerous to list here.

https://github.com/mozilla/rust/blob/master/AUTHORS.txt

What Rust has...

– Traits (type classes)

– Ownership (affine types) 1

– Algebraic data types

– Lifetimes (regions) 2

– Actor-style concurrency

What Rust doesn't have...

– Null pointers

– Dangling pointers

– Segmentation faults

– Data races

– Mandatory GC

Why optional GC?

1. Efficiency and predictability

- Avoid unpredictable latency
- Particularly on mobile

2. Memory disjoint tasks

- If you can send memory, you can free it.

Preview: Freezing

```
let mut table: Hashtable<K,V> = ...;

{
  let value = table.find(...);

  table.insert(...);

  match value { ... }
}

table.insert(...);
```

Create hashtable

Find “freezes” hashtable for scope of value

Illegal, hashtable frozen

OK, value is out of scope

Hashtable in Rust

```
struct Hashtable<K,V> {
  buckets: ~[Option<Bucket<K,V>>]
}

struct Bucket<K,V> {
  key: K,
  value: V
}
```

~[T]: Owned pointer to an array

What's in a type

~[T]: Owned pointer to an array

```
~[Option<Bucket<K,V>>]
```

Option<T>: I guess you've seen this before

What is ownership?

In Rust, ownership means:

The Right To Free Memory

$\Rightarrow$

Control over aliasing

Ownership is implicit in C/C++

Some pointers are temporary:

```
memcpy(void *dest, const void *src, uintptr_t count);

V* find(Hashtable<K,V> *table, K *key);
```

Some pointers are not:

```
struct Hashtable {
    Bucket<K,V> *buckets;
    ...
}

void *realloc(void *, ...);
void free(void *);
```

Ownership is explicit in Rust

Temporary pointers are designated &T

Owned pointers are designated ~T

Managed pointers are designated @T

Owned pointers

Owned pointers never alias one another and are automatically freed:

```
{
  let x: ~int = ~22;
  …
}
```

Allocate owned integer

x freed automatically here

Owned pointers are moved

Owned pointers are moved from place to place.

```
fn move_from() {
  let x: ~int = ~22;

  move_to(x);

  printf(“%d”, *x); // Error: x was moved
}

fn move_to(y: ~int) {
  ...
}
```

Moves x into callee’s stack frame

x no longer accessible

Freed by callee

Moving into a data structure

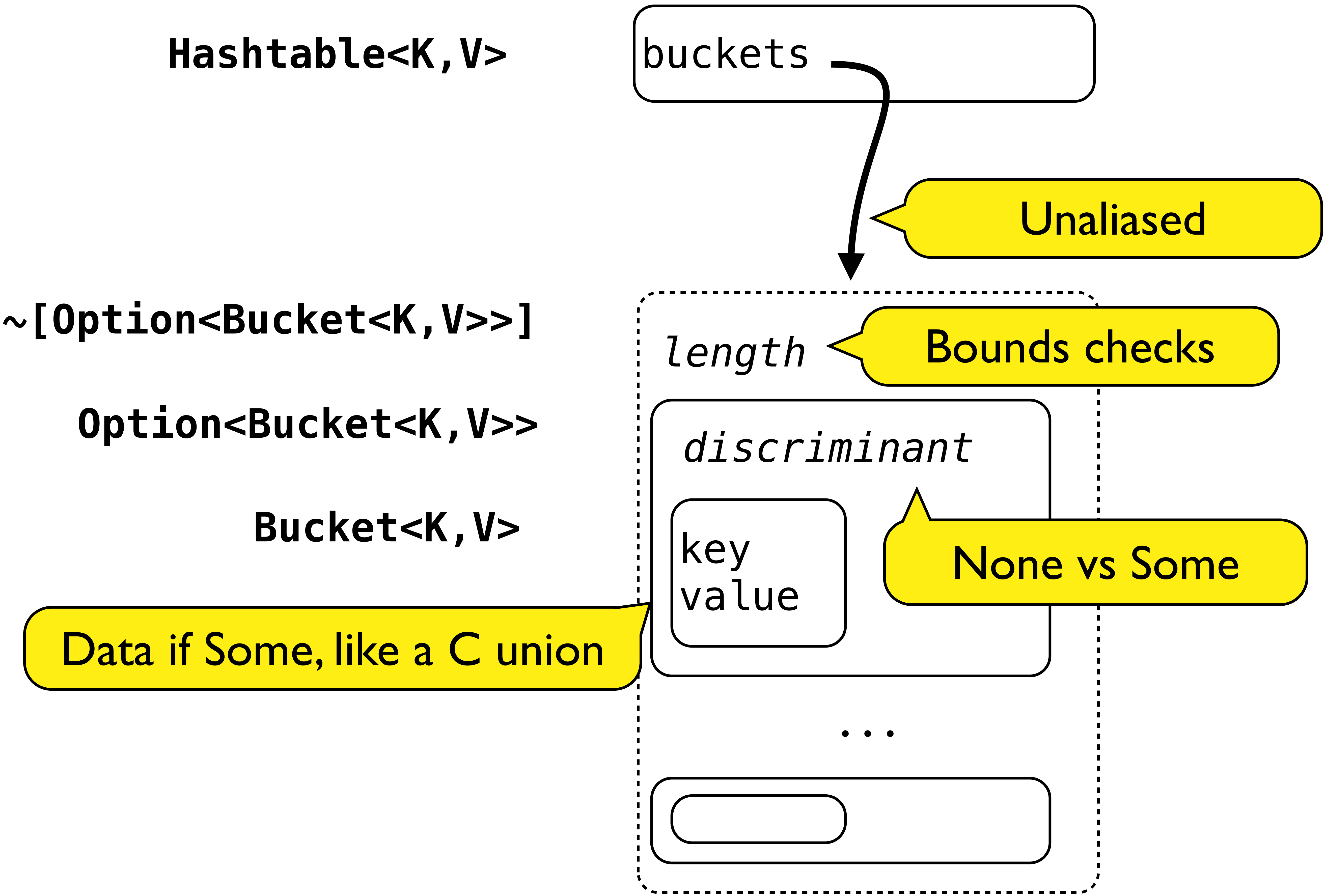
Owned pointers can also be owned by a data structure.

```
let b = ~[];

let table = Hashtable {
  buckets: b
};

b[0] = ...; // Error
table.buckets[0] = ...; // OK
```

Create a fresh array

Move it into the hashtable

Only accessible via table


Hashtable<K,V>
buckets
Unaliased
~[Option<Bucket<K,V>>]
length
Bounds checks
Option<Bucket<K,V>>
discriminant
Bucket<K,V>
key
value
None vs Some
Data if Some, like a C union
. . .

Insertion, first attempt

```
fn insert<K,V>(
  mut table: ~Hashtable<K,V>,
  key: K,
  value: V)
  -> ~Hashtable<K,V>
{
  if (table full) {
    table.buckets = resize_buckets(table.buckets);
  }

  let index = find_bucket(table, &key);
  table.buckets[index] = Some(Bucket {key: key,
                                      value: value});
  return table;
}
```

Take ownership of a hashtable

Safe because of ownership

Update buckets array

Give table back to the caller.

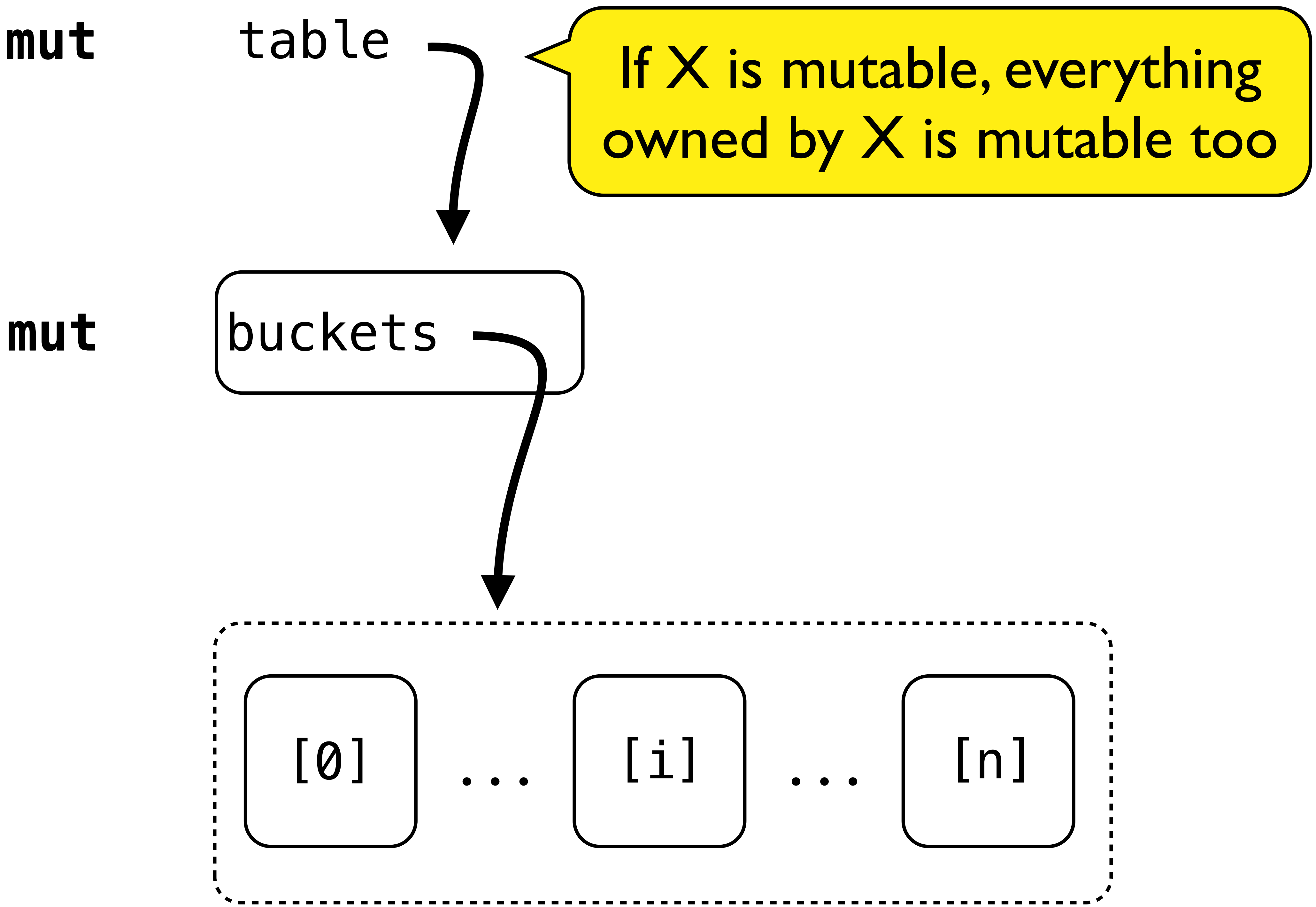
Mutability in Rust

```
let x = 22;
x += 1; // Error

let mut x = 22;
x += 1; // OK

fn update(x: int) {
  x += 1; // Error
}

fn update(mut x: int) {
  x += 1; // OK
}
```

Local variables must be declared as mutable

Parameters too

Mutability and ownership



Inherited mutability

```
fn insert<K,V>(
  mut table: ~Hashtable<K,V>,
  key: K,
  value: V)
  -> ~Hashtable<K,V>
{
  if (table full) {
    table.buckets = resize_buckets(table.buckets);
  }

  let index = find_bucket(table, &key);
  table.buckets[index] = Some(Bucket {key: key,
                                      value: value});
  table
}
```

table is mutable

table.buckets is mutable

table.buckets[index] is mutable

Ownership is explicit in Rust

Temporary pointers are designated &T

Owned pointers are designated ~T

Managed pointers are designated @T

Borrowing

```
fn insert<K,V>(
  table: &mut Hashtable<K,V>,
  key: K,
  value: V)
{
  if (table full) {
    resize_buckets(&mut table.buckets);
  }

  let index = find_bucket(&*table, &key);
  table.buckets[index] = Some(Bucket {key: key,
                                      value: value});
}
```

Mutable borrowed pointer as input

Mutable borrow

Immutable borrows

Borrowing

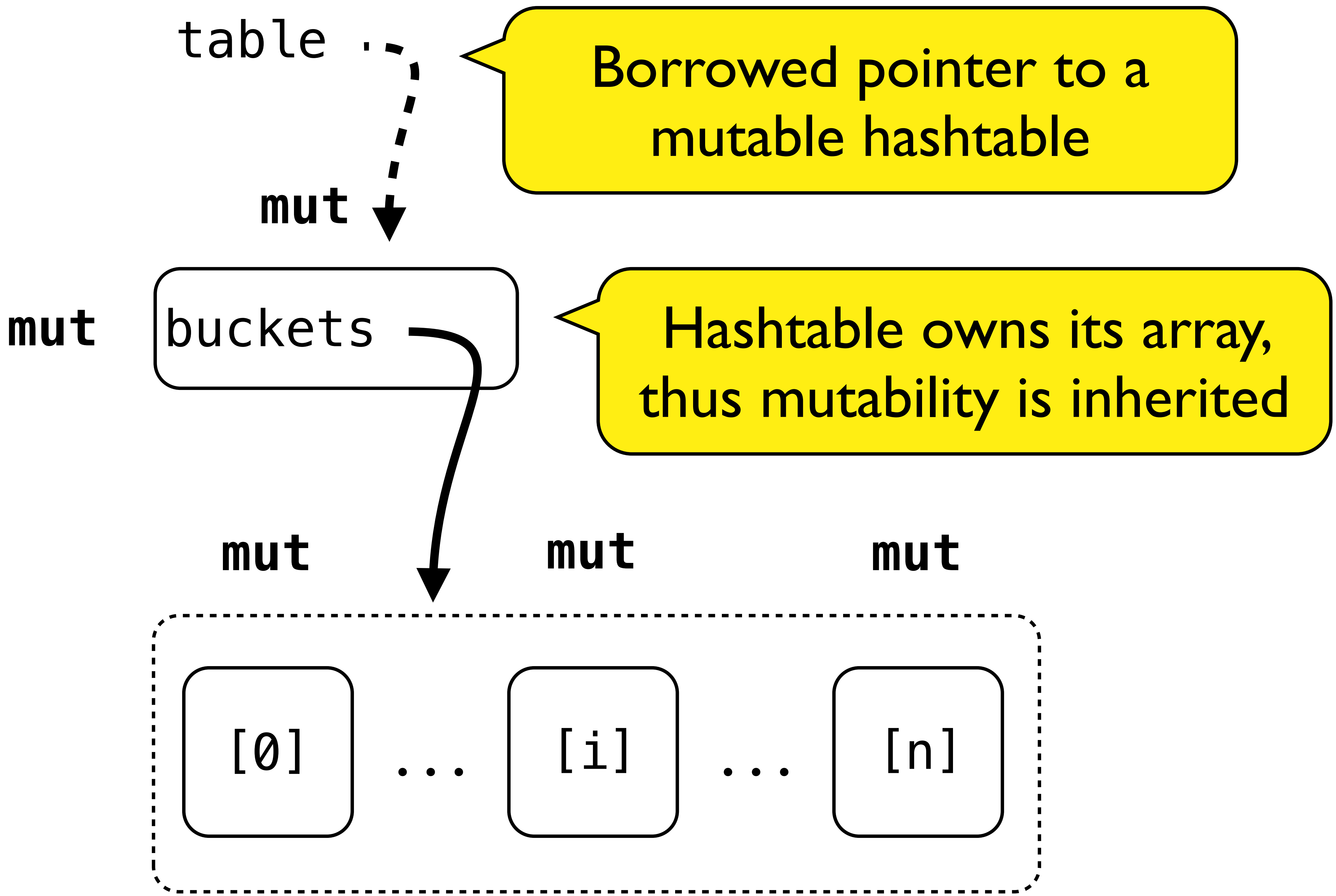


table
mut
Borrowed pointer to a mutable hashtable
mut
buckets
Hashtable owns its array, thus mutability is inherited
mut
mut
mut
[0]
...
[i]
...
[n]

Mutability and uniqueness

`&mut T` Borrowed pointer to mutable data

Type system guarantees uniqueness: no other mutable pointer to the same data.

Linearly tracked.

Mutable borrows

Mutable local variable

x is borrowed for the lifetime of y

```
let mut x = 5;

{
  let y = &mut x;
  x += 1;  // Error: borrowed
  *y += 1; // OK
}

x += 1; // OK
```

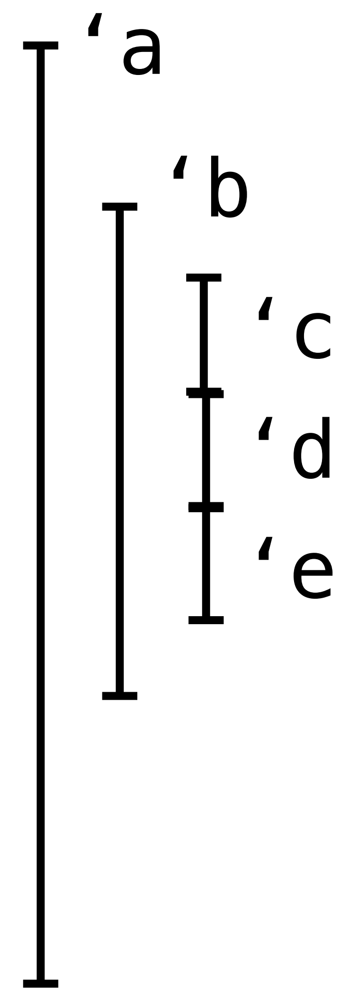
Borrow has expired, y out of scope

Mutable borrows prevent owner from writing.

Lifetimes

```
let mut x = 5;

{
  let y = &mut x;
  x += 1;  // Error: borrowed
  *y += 1; // OK
}

x += 1; // OK
```

y : &’b mut int



Lifetime parameters

```
fn insert<K,V>(
  table: &mut Hashtable<K,V>,
  key: K,
  value: V)
{
  ...
}
```

For any lifetime ‘a specified by caller...

```
fn insert<‘a,K,V>(
  table: &‘a mut Hashtable<K,V>,
  key: K,
  value: V)
{
  ...
}
```

Mutable borrows require

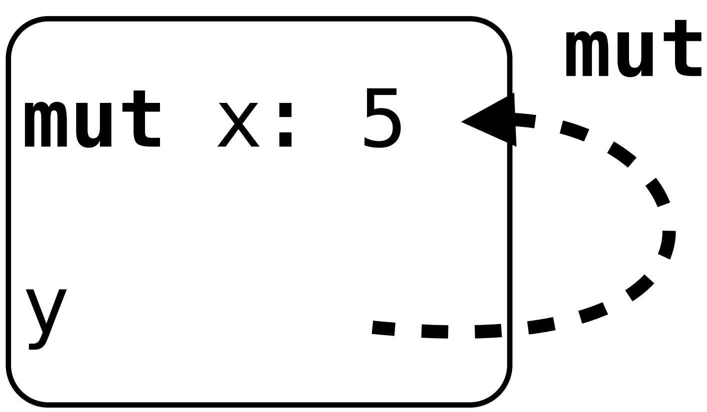
1. Data must be mutable

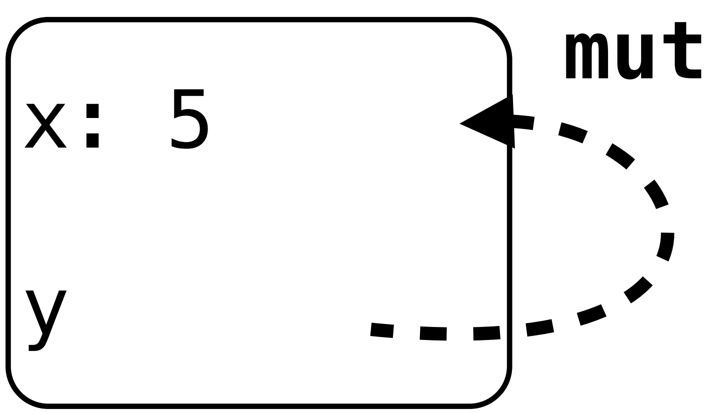
```
let mut x = 5;
let y = &mut x;
```

```
let x = 5;
let y = &mut x;
```

OK

ILLEGAL





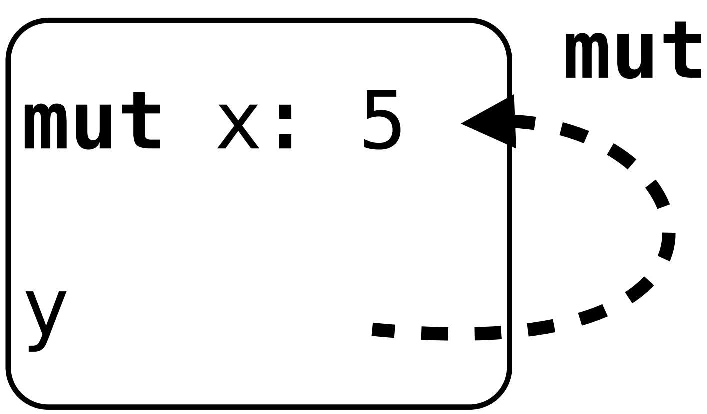
Mutable borrows require

2. Data must live long enough

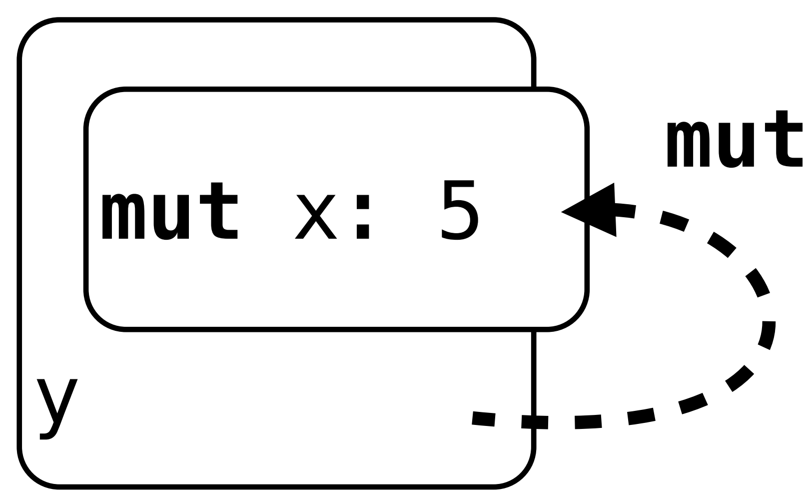
```
{
  let mut x = 5;
  let y = &mut x;
}
```

```
let y;
{
  let mut x = 5;
  y = &mut x;
}
*y += 1;
```

OK



ILLEGAL

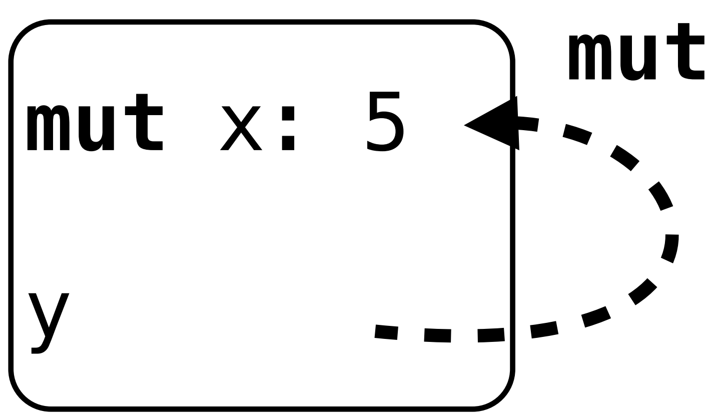


Mutable borrows require

3. Data must be uniquely referenced

```
let mut x = 5;
let y = &mut x;
```

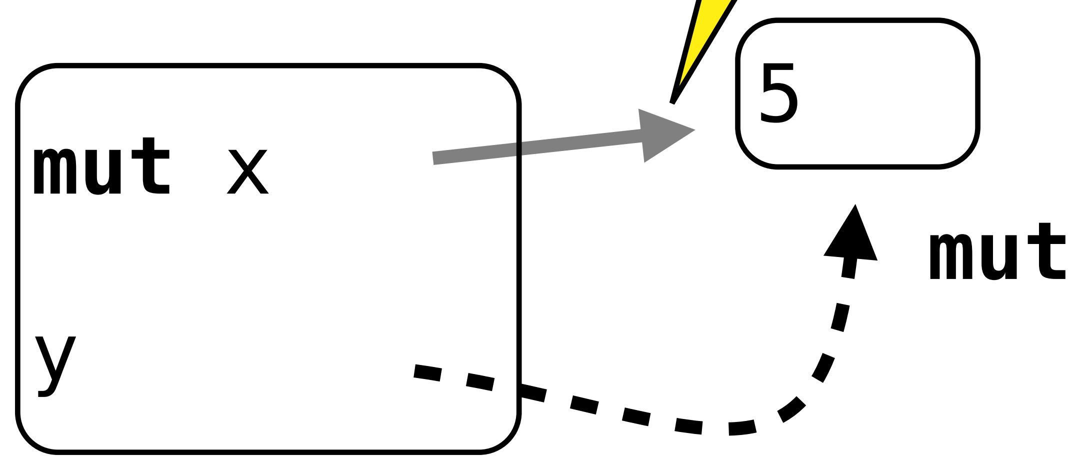
OK




```
let mut x = ~5;
let y = &mut *x;
```

Unusable while y is in scope

OK



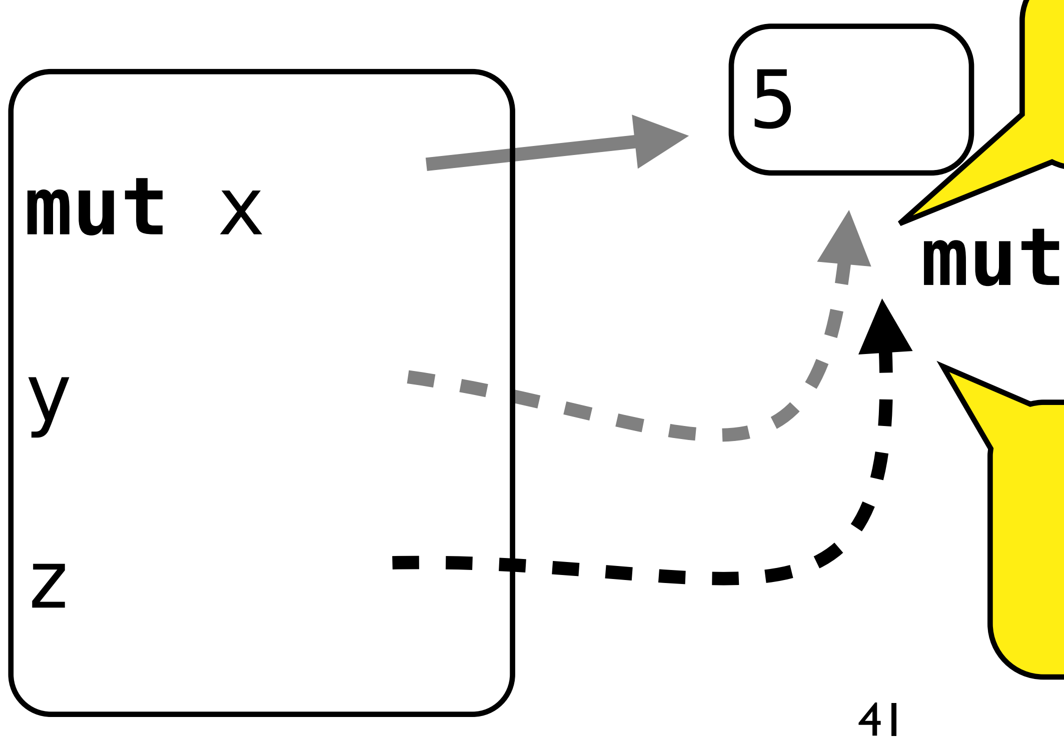
Mutable borrows require

3. Data must be uniquely referenced (cont'd)

```
let mut x = ~5;
let y = &mut *x;
let z = &mut *y;
```

Negative examples to come!

OK



Unusable while z is in scope

Only safe because &mut unique

Immutable borrowed pointers

&T — Borrowed pointer to immutable data

May be aliased

Stronger than C++ const, **nobody** can modify it

Immutable borrows

```
let x = 5;
{
  let y = &x;
  print(x);
}
```

Original can still be read

Immutable borrows do not restrict owner from reading.

But there are other restrictions...

Freezing

```
let mut x = 5;
{
  let y = &x;
  x += 1; // Error
}
x += 1; // OK
```

x declared as mutable

Frozen for lifetime of y

y out of scope, unfrozen

Immutable borrows require

1. Data must live long enough

```
{
  let x = 5;
  let y = &x;
}
```

```
let y;
{
  let x = 5;
  y = &x;
}
print(*y);
```

OK

ILLEGAL

Same as with &mut

Immutable borrows require

2. Mutable data must be uniquely referenced.

```
let mut x = 5;
let y = &x;
```

OK

```
let mut x = ~5;
let y = &*x;
```

OK

```
let mut x = ~5;
let y = &mut *x;
let z = &*y;
```

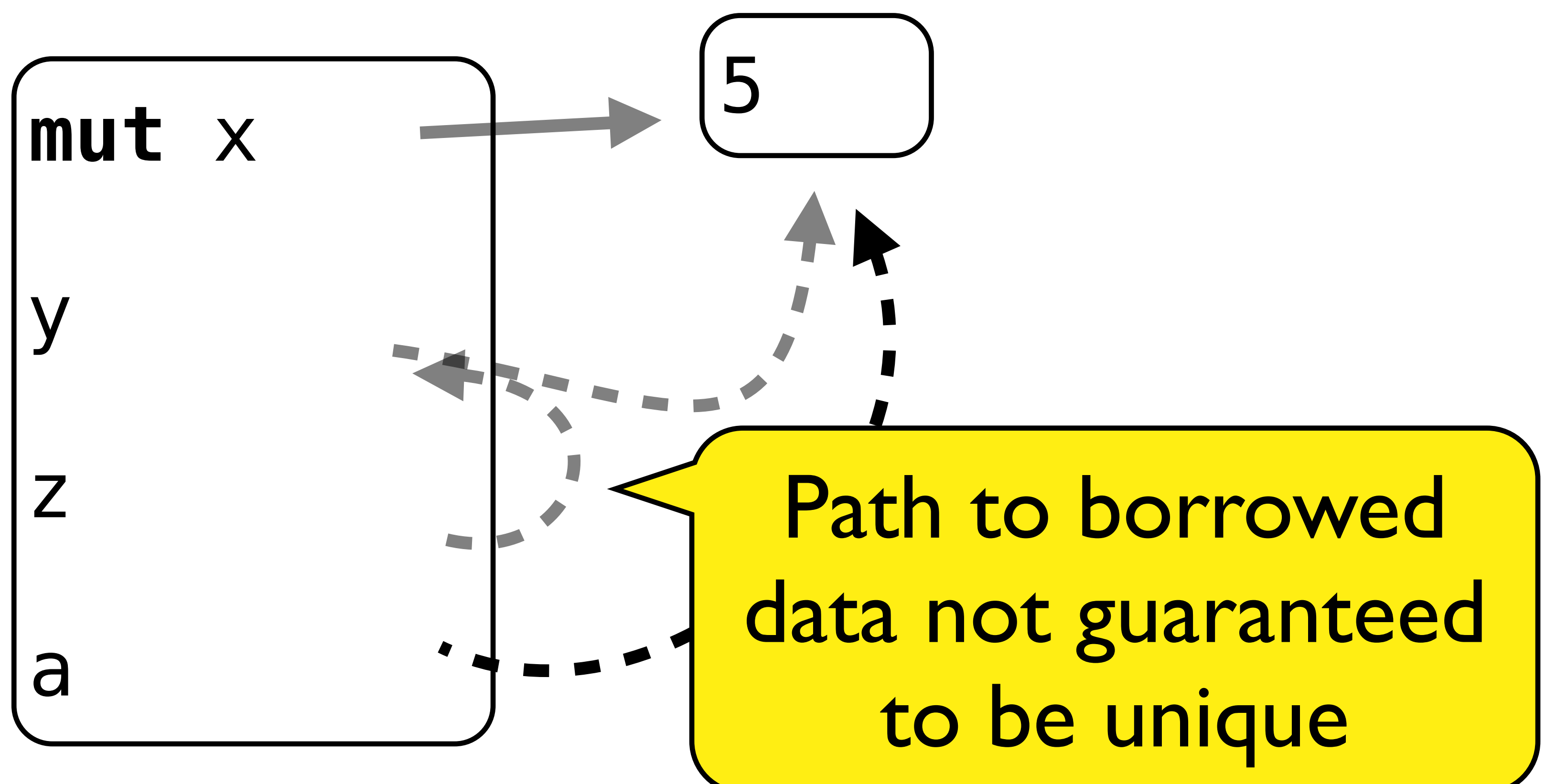
OK

Same as with &mut

An illegal case...

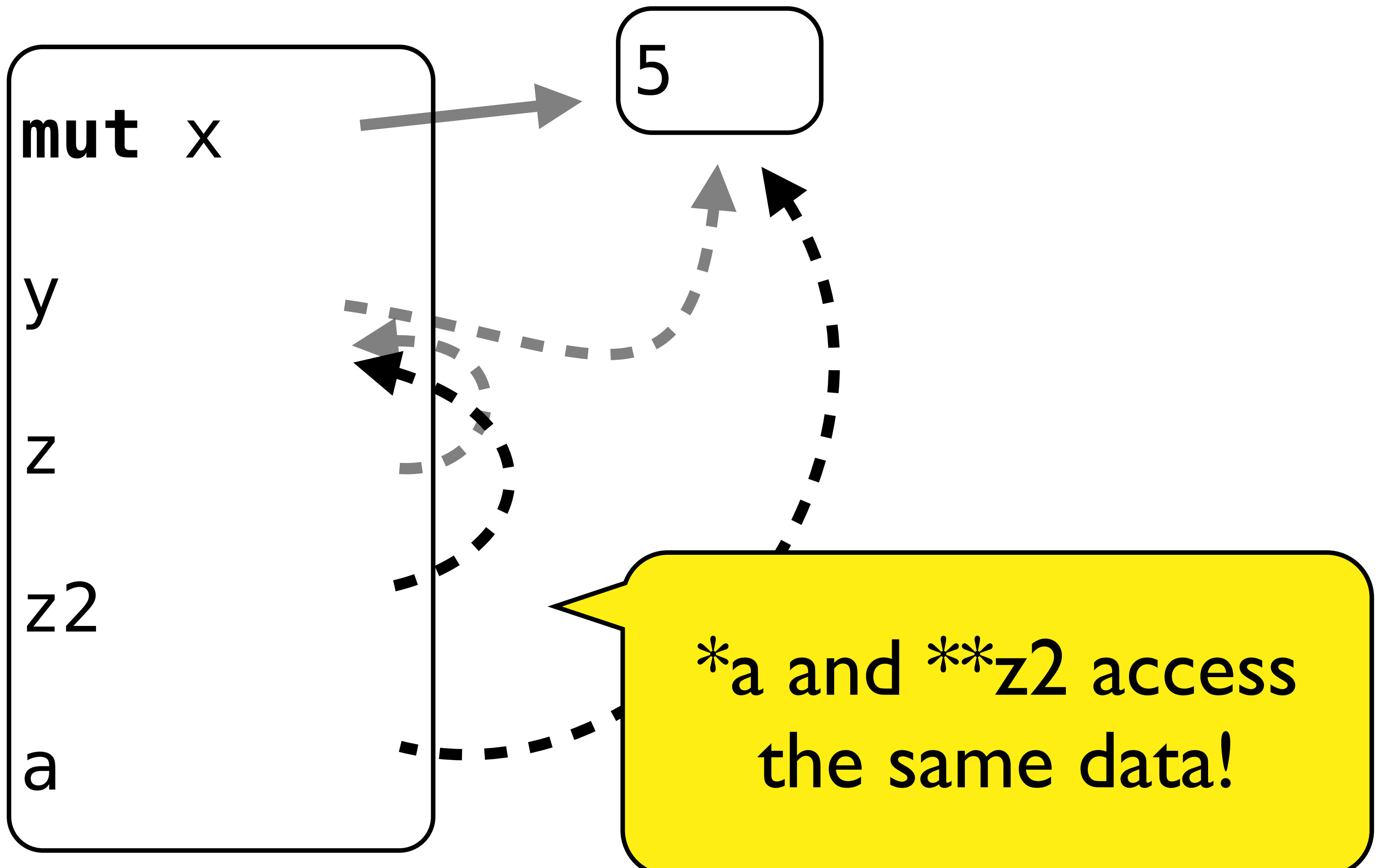
```
let mut x = ~5;
let y = &mut *x;
let z : & &mut int = &y;
let a = &mut **z;
```

ILLEGAL



An illegal case (cont'd)

```
let mut x = ~5;
let y = &mut *x;
let z : & &mut int = &y;
let z2 = z;
let a = &mut **z;
```



Let's put it all together

Remember find()?

Returns a pointer into the hashtable.

Goal: Ensure that hashtable is not modified while this pointer is live.

Lifetime parameters in find

```
fn find<‘a,K,V>(
  table: &‘a Hashtable<K,V>,
  key: &K)
  -> Option<&’a V>
{
  ...
}
```

Lifetime parameters in find

Given: pointer with lifetime ‘a to an immutable hashtable

```
fn find<‘a,K,V>(
  table: &‘a Hashtable<K,V>,
  key: &K)
  -> Option<&’a V>
{
  ...
}
```

Yields: (optional) pointer with lifetime ‘a to an immutable value V

In other words, the value returned is valid as long as:
1. the hashtable is valid
2. the hashtable is not mutated

Find

```
fn find<‘a,K,V>(
  table: &‘a Hashtable<K,V>,
  key: &K)
  -> Option<&’a V>
{
  let index = find_bucket(table, &key);
  match table.buckets[index] {
    None => None,
    Some(ref bucket) => Some(&bucket.value)
  }
}
```

Search for index...

Creates a pointer into value being matched

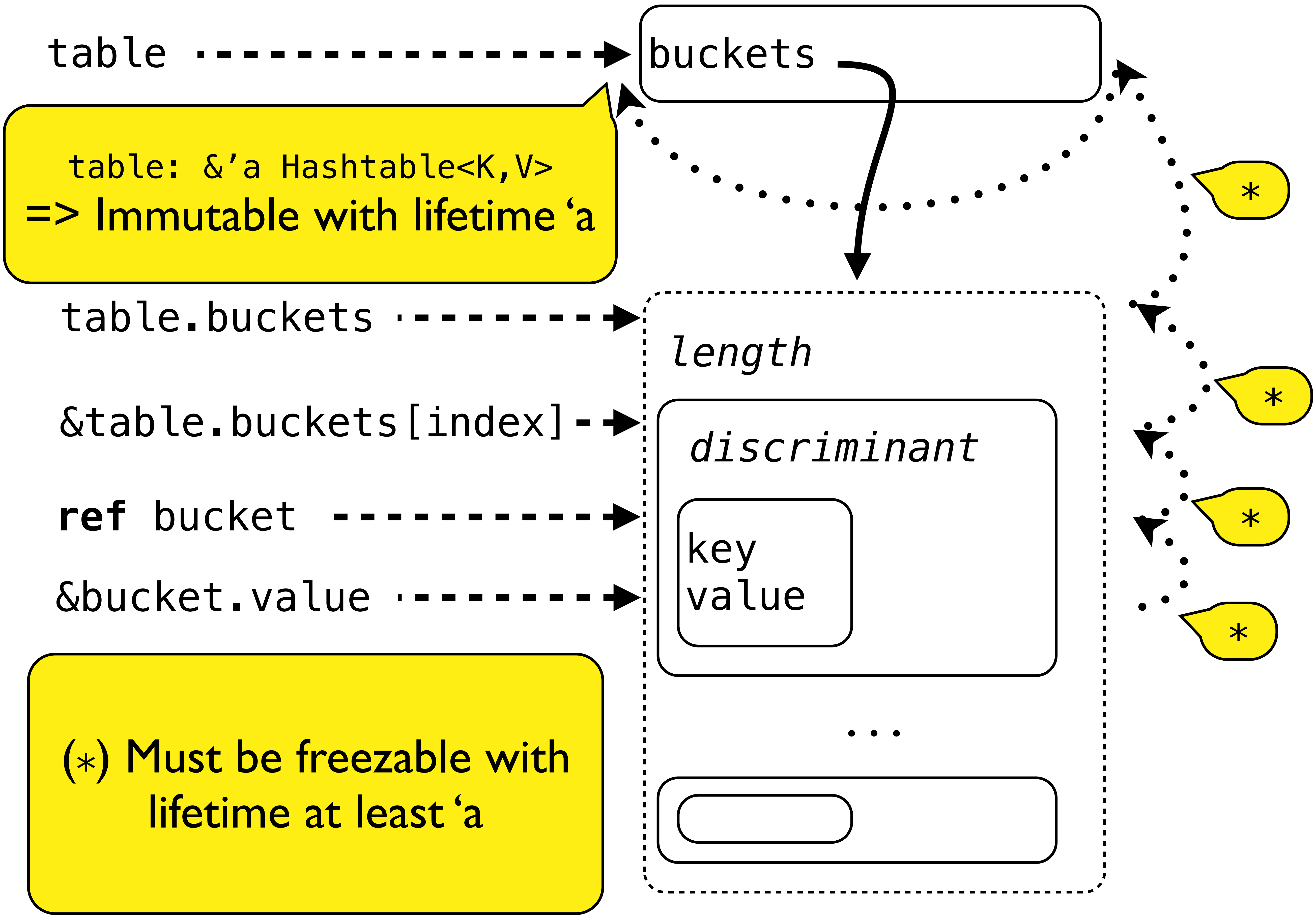


table
buckets
table: &’a Hashtable<K,V>
=> Immutable with lifetime ‘a
table.buckets
length
&table.buckets[index]
discriminant
ref bucket
key
value
&bucket.value
...
*
*
*
*
(*) Must be freezable with lifetime at least ‘a

Voila

```
let mut table: Hashtable<K,V> = ...;

{
  let value = find(&table, ...);

  insert(&mut table, ...);

  match value { ... }
}

insert(&mut table, ...);
```

Frozen for lifetime of value

Error, frozen.

OK, value out of scope.

But wait, there's more...

```
fn compute(input: &[T], output: &mut [T]) {
  if output.len() >= threshold {
    let (left, right) = output.split();
    parallel::do([
      || compute(input, left),
      || compute(input, right)]);
  } else {
    ...
  }
}
```

Immutable, safe to share.

Mutable, but disjoint. Safe.

Divide buffer into two disjoint halves

Recap #1

Owned pointers are
moved, not freely aliased.

⇒

Safe to free.

Safe to send
to another thread.

Safe to resize.

Recap #2

Borrowing limits the owner
for the lifetime of the borrow.

⇒

Borrowed values cannot be
moved or sent between tasks.

Mutable borrowed values can
be temporarily frozen.

Recap #3

Mutable borrowed pointers are unique.

$\Rightarrow$

Allows reborrowing, resizing, and possibly parallelism beyond actors (wip).

More information?

Nicholas Matsakis
nmatsakis@mozilla.com

rust-lang.org

smallcultfollowing.com/babysteps