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;
}
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;
}
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;
}
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

```cpp
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

```cpp
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)
– Algebraic data types
– Lifetimes (regions)
– 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.
let mut table: Hashtable<K,V> = ...;
{
    let value = table.find(...);
    table.insert(...);
    match value {
        ... } 
} 

Illegal, hashtable frozen

OK, value is out of scope
Hashtable in Rust

```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

⇒

Control over aliasing
Ownership is implicit in C/C++

Some pointers are temporary:

```c
memcpy(void *dest, const void *src, uintptr_t count);
V* find(Hashtable<K,V> *table, K *key);
```

Some pointers are not:

```c
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:

```haskell
{ 
    let x: ~int = ~22;
    ... 
}
```

Allocate owned integer

x freed automatically here
Owned pointers are moved from place to place.

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

    move_to(x);  // Moves x into callee's stack frame

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

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

x no longer accessible
Freed by callee
Moving into a data structure

Owned pointers can also be owned by a data structure.

```ocaml
let b = ~[];

let table = Hashtable {
  buckets: b
};

b[0] = ...; // Error

// Move it into the hashtable

// Only accessible via table

// Create a fresh array
```

Thursday, July 18, 13
Hashtable\(<K, V>\)

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

Option<Bucket<K, V>>

Bucket\(<K, V>\)

Data if Some, like a C union

Unaliased

Bounds checks

None vs Some

length

discriminant

key

value

...
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;
}
Mutability in Rust

```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
If X is mutable, everything owned by X is mutable too.

**Mutability and ownership**

mut table

mut buckets

[0] ... [i] ... [n]
Inherited mutability

```rust
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 \sim T

Managed pointers are designated @T
fn insert<K,V>(
    table: &mut HasTable<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});
}
Borrowing

Borrowed pointer to a mutable hashtable

Hashtable owns its array, thus mutability is inherited
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

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

x += 1; // OK
```

Mutable local variable

- `x` is borrowed for the lifetime of `y`

Borrow has expired, `y` out of scope

- `x` is borrowed for the lifetime of `y`

Mutable borrows prevent owner from writing.
let mut x = 5;
{
    let y = &mut x;
    x += 1; // Error: borrowed
    *y += 1; // OK
}

x += 1; // OK
fn insert<K,V>(
    table: &mut Hashtable<K,V>,
    key: K,
    value: V)
{
    ...
}

fn insert<'a,K,V>(
    table: &'a mut Hashtable<K,V>,
    key: K,
    value: V)
{
    ...
}
Mutable borrows require

1. Data must be mutable

\[
\begin{align*}
\text{let } \text{mut } x &= 5; \\
\text{let } y &= \&\text{mut } x;
\end{align*}
\]

\text{OK}

\[
\begin{align*}
\text{let } x &= 5; \\
\text{let } y &= \&\text{mut } x;
\end{align*}
\]

\text{ILLEGAL}
Mutable borrows require

2. Data must live long enough

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

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

**OK**

```rust```

**ILLEGAL**

```rust```
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;

OK

Unusable while y is in scope
Mutable borrows require

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

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

OK

Unusable while z is in scope

Only safe because &mut unique

Negative examples to come!
Immutable borrowed pointers

\&T

Borrowed pointer to immutable data

May be aliased

Stronger than C++ const, nobody can modify it
Immutable borrows

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

Immutable borrows do not restrict owner from reading.

But there are other restrictions...
Freezing

```rust
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

```rust
{   
    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.

\[
\begin{align*}
\text{let mut } x &= 5; \\
\text{let } y &= &x; \\
\text{OK}
\end{align*}
\]

\[
\begin{align*}
\text{let mut } x &= \sim5; \\
\text{let } y &= &*x; \\
\text{OK}
\end{align*}
\]

\[
\begin{align*}
\text{let mut } x &= \sim5; \\
\text{let } y &= &\text{mut } *x; \\
\text{let } z &= &*y; \\
\text{OK}
\end{align*}
\]

\text{Same as with } &\text{mut}
let mut x = ~5;
let y = &mut *x;
let z : & &mut int = &y;
let a = &mut **z;

ILLEGAL

Path to borrowed data not guaranteed to be unique
An illegal case (cont’d)

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

*a and **z2 access the same data!*
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

```rust
define find<'a,K,V>(
    table: &'a Hashtable<K,V>,
    key: &K)
    -> Option<&'a V>
{
    
    ...
}
```
Lifetime parameters in find

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

Given: pointer with lifetime ‘a to an immutable hashtable

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
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)
    }
}
table: &'a Hashtable<K,V> => Immutable with lifetime `a

table.buckets

&table.buckets[index]

ref bucket

&bucket.value

(*)  Must be freezable with lifetime at least `a
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...

```rust
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 {
        ...
    }
}
```

Divide buffer into two disjoint halves

Immutable, safe to share.

Mutable, but disjoint. Safe.
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.
Mutable borrowed pointers are unique.

⇒

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

Nicholas Matsakis
nmatsakis@mozilla.com

rust-lang.org

smallcultfollowing.com/babysteps