CSC2/455 Software Analysis and Improvement
Rust

Sreepathi Pai

URCS

April 17, 2019

Outline

Outline

Problems

- We want to prevent certain classes of errors
- For example, for heap-allocated data, we may want to:
  - ensure there is (one writer and no readers) OR (multiple readers and zero writers)
  - automatically free data *only* when it can no longer be referenced (i.e. no pointer to it exists)
- To prevent:
  - data races (one writer and no readers)
  - use-after-free errors (no pointer to data, so use-after-free is impossible)

Current Solutions

- Preventing data races
  - Use mutexes (i.e. locks)
  - Use atomics (i.e. serialized writes/reads)
- Preventing use-after-free
  - Never Free
  - Garbage Collection (e.g., reference counting, mark-and-sweep, etc.)

These are runtime solutions and carry a runtime cost.

Compiler-based Solutions

Can we have a compiler-based solution that:

- detects data races at the source level?
- automatically frees data when it is are no longer “reachable”?
- and has no runtime cost?
- and still allows everything we can do currently in C/C++?

Nope

Not really, but Rust tries to get very close.

Disclaimer

Warning: I only know Rust from its documentation, not by implementing it.

Outline

Immutability by default

All variables in Rust are immutable by default.

```
let x = 5
x = 6; // ERROR!
```

Variables can be made mutable by changing their type.

```
let mut x = 5;
x = 6;
```

Ownership

Values cannot be copied.

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world", s1); // ERROR: can't use s1
```

Values are moved, and `s1` is invalidated after the assignment to `s2`. I.e. values are "owned" (by a variable).

Moves on “primitive” values are copies

```
let x = 5;
let y = x;      // IS A COPY, NOT A MOVE

println!("x = {}, y = {}", x, y); // WORKS
```

If a type is Copy-able, Rust makes a copy, not a move.

Functions with move-only values

```
fn takes_ownership(s: String) {
     println!("{}", s);
}

fn main() {
   let s = String::from("hello, world!");

   takes_ownership(s);

   println!("{}", s); // ERROR!
}
```

Functions with move-only values: Returning Ownership

```
fn takes_ownership(s: String) {
     println!("{}", s);

     s
}

fn main() {
   let s = String::from("hello, world!");

   let s2 = takes_ownership(s);

   println!("{}", s2);
}
```

Why move semantics?

When a variable goes out of scope in Rust, its value may get dropped (i.e. freed)

- Obviously, can't do this if we don't transfer ownership
- I.e., only the owner can drop a value

References aka Borrowing

We can create a *reference* to temporarily allow shared access – i.e. "borrow" a value.

```
fn takes_reference(s: &String) {
     println!("{}", s);
}

fn main() {
   let s1 = String::from("hello, world!");

   takes_reference(&s1);

   println!("{}", s1);  // should work!
}
```

References create aliases, i.e. multiple names for the same value. Can this *aliasing* cause issues?

Aliases permit read-only sharing

Aliases, like the ones we've created so far with `&`:

- permit sharing, but not writing
- still only one owner (all non-owners are distinguished by `&` in type)
- need to ensure that owner outlives all aliases

Mutability and References

```
let mut s = String::from("mutable string");

let r1 = &mut s;
let r2 = &mut s;      // ERROR: can't have two mutable references

println!("{}, {}", r1, r2);
```

Can't have two mutable references to the same value live simultaneously.

Mutability and References: Part Two

```
let mut s = String::from("mutable string");

let r1 = &s;
let r2 = &s;
let r3 = &mut s;  // ERROR

println!("{}, {}, {}", r1, r2, r3);
```

Can't have mutable and immutable references to the same value live simultaneously.

Mutability and References: Using Scopes

```
let mut s = String::from("mutable string");

{
   let r1 = &mut s;
} // r1 goes out of scope here

let r2 = &mut s;       // WORKS
```

Rules

- One owner
- Can have:
  - zero or more immutable references
  - one mutable reference
  - but not both simultaneously *in scope*

Lifetimes

- All references have "scope of validity" or a *lifetime* associated with them

```
{
    let r;
    {
        let x = 5;
        r = &x;          // ERROR
    } // x goes out of scope here

    println("{}", r);
}
```

- `r` has lifetime `'a`, and `x` has lifetime `'b`
- `'a` outlives `'b`, so `r` can't contain a reference to `x`

Lifetime annotations

- `&i32`, reference to an `i32` variable
- `&'a i32`, reference to a `i32` variable with lifetime `'a`
- `&'a mut i32`, mutable reference to a `i32` variable with lifetime `'a`

Note that lifetimes are inferred. The annotations only give us a way to name them (i.e. annotations can't specify lifetimes).
A special lifetime `'static` indicates whole-program lifetime.

Function lifetime annotations: #1

```
fn example<'a>(arg1: &'a str, arg2: &'a str) -> &'a str {
}
```

- read as `arg1` and `arg2` have the same lifetimes, and the return value also has the same lifetime.
- the `<'a>` syntax is the same as for generics (i.e. polymorphism)

Function lifetime annotations: #2

```
fn example<'a, 'b>(arg1: &'a str, arg2: &'b str) -> &'b str {
}
```

read as?

Preventing mutation

```
fn example(v: &'a mut Vec<i32>) {                    // 'a begins
    v.push(21);                                      // 'c begins and ends

    {
        let mut head: &'b mut i32 = v.index_mut(0);  // 'b begins
        // cannot access v here
        *head = 23;                                  // 'b ends
    }

    v.push(42);
    println!("{:?}", v); // prints [23, ..., 42]
}                                                    //  'a ends
```

What is the lifetime of the return type of the `index_mut` function?

Jung, Jourdan, Krebbers and Dreyer, RustBelt: Securing the Foundations of the Rust Programming Language, POPL 2018

The lifetime 'k

```
fn index_mut<'k>(a1: &'k mut Vec<i32>, a2: usize) -> &'k mut i32
```

- `index_mut` return type must have the lifetime of `head`, which is inferred.
- So `'k` is the same as `'b`
- this means that the borrow of `a1` must be for the same duration (from the type signature)

Story so far

- Rust values
  - are immutable by default
  - are moved when assigned
  - have a lifetime associated with them
- Rust references
  - point to a value, and borrow access to it
  - can't outlive original values
- Lifetimes are inferred

Outline

Warning

- The behaviour of the borrow checker is not described completely anywhere
  - Use the source, Luke!
- Descriptions of the borrow checker have evolved over time
  - on HIR
  - on MIR
  - using alias analysis, etc.

Checks

- Type checks
- Path checks
  - There is only one mutable reference to a value in scope at any time
- Lifetime checks
  - References do not outlive their original values

An Alias Based Formulation of the Borrow Checker

An alias-based formulation of the borrow checker

Outline

References

- The Rust Book
  - Chapter 4: Understanding Ownership
  - Chapter 10: Generic Types, Traits and Lifetimes
- Jung et al., RustBelt: Securing the Foundations of the Rust Programming Language, POPL 2018
  - Section 2, in particular
- The Rust Language Reference
- The Rust C Guide: MIR Borrow Checker (incomplete)
- Rust Borrow Checker (old)
- Rust RFC 2094: Non-lexical Lifetimes