Rust in Peace

Balaji Sivaraman (@balajisivaraman)

ThoughtWorks

March 15, 2018

About Me

- Primarily worked on Java/Spring/ROR stack in ThoughtWorks, writing microservices
- Pure functional programming advocate in languages like Scala/Haskell/Purescript
- Bitten by the Rust bug last year after reading a post on how it enabled Firefox's superior performance
- Currently on the way to transitioning from an applications developer to a systems programmer, thanks primarily to Rust

Agenda

- Introduction
- Ground Rules
- Systems Programming
- What about C or C++?
- The Rust Programming Language
- Ownership and Borrowing
- Lifetimes
- Why Rust - The Good Stuff
- Questions?

Ground Rules

What this talk is about?

- How Rust benefits newcomers to systems programming?
- What modern PL design sensibilities has Rust borrowed?
- What PL design ideas has Rust brought forth?

What this talk is not?

- Fully detailed comparison between Rust & C/C++

Systems Programming

- What is Systems Programming?
- Why is it different from application programming?

What is Systems Programming?

From O'Reilly's Programming Rust [1]:

> *You close your laptop. The OS detect this, suspends all the running programs, turns off the screen, and puts the computer to sleep. Later, you open the laptop: the screen and other components are powered up again, and the program is able to pick up where it left off. We take this for granted. But systems programmers wrote a lot of code to make that happen.*

So, what is Systems Programming?

Again from O'Reilly's Programming Rust [2]:

Systems programming is ***resource-constrained*** *programming. It is programming when every byte and every CPU cycle counts.*

What about C or C++?

- Why have these languages dominated this space for 3 decades?
- Why is it time for a change right now?

Key Language Features

- Functional Language Features I like in Rust
  - Pattern Matching
  - ENums similar to Algebraic Data Types
  - Lazy Iterators
  - Functions as first class values
  - Error Handling Primitives using Result ADT
- Rust Lang Features I Like
  - **Ownership, Borrowing and Lifetimes**
  - Unit Testing primitives as part of the core language
  - Concurrency Primitives - Threads, Channels, Atomic Values etc.

Ownership and Borrowing

A Simple Program

```
pub fn main() {
  let v = vec!(1,2,3);
  println!("{:?}", v);
}
```

What the Rust compiler does?

```
pub fn main() {
  let v = vec!(1,2,3); ⟶ Rust compiler allocates memory in heap for V here
  println!("{:?}", v);
} ⟶ Rust compiler drops the vector V here
```

Another Simple Program

```rust
pub fn main() {
  let v = vec!(1,2,3);
  do_something(v);
  println!("{:?}", v);
}

fn do_something(v: Vec<u64>) {
  // Do something with v
}
```

What happens here?

```rust
pub fn main() {
  let v = vec!(1,2,3);
  do_something(v);
  println!("{:?}", v);
}

fn do_something(v: Vec<u64>) {
  // Do something with v
}
```

Ownership of V transferred to `do_something`

Rust doesn't allow us to use V here

Rust compiler drops the vector V here

Returning Ownership Back

```
pub fn main() {
  let v = vec!(1,2,3);
  let v1 = do_something(v);  ⟶ Ownership of V transferred to do_something
  println!("{:?}", v1);  ⟶ We can safely use V1 here
}  ⟶ Rust compiler drops the vector V1 here

fn do_something(v: Vec<u64>) -> Vec<u64> {
  // Do something with v
  return v;  ⟶ Ownership of V returned to calling fn
}
```

Borrowing

```
pub fn main() {
  let v = vec!(1,2,3);
  do_something(&v);  ⟶  do_something borrows ownership of V
  println!("{:?}", v);  ⟶  Rust doesn't complain about V usage here
}  ⟶  Rust compiler drops the vector V here

fn do_something(v: &Vec<u64>) {
  // Do something with v
}
```

Let's Mutate Things

```
pub fn main() {
  let v = vec!(1,2,3);
  do_something(&v);
  println!("{:?}", v);
}

fn do_something(v: &Vec<u64>) {
  v.push(4);
}
```

Uh Oh!

```
pub fn main() {
  let v = vec!(1,2,3);
  do_something(&v);
  println!("{:?}", v);
}

fn do_something(v: &Vec<u64>) {
  v.push(4); ⟶ Cannot mutate immutably borrowed V here
}
```

Everything is Mutable

```
pub fn main() {
  let mut v = vec!(1,2,3);
  do_something(&mut v);
  println!("{:?}", v);
}
```

→ Rust compiler drops the vector V here

```
fn do_something(v: &mut Vec<u64>) {
  v.push(4);
}
```

One Final Note [3]

```
pub fn main() {
  let mut v = vec!(1,2,3);
  let v1 = &v; //First Immutable Borrow is Fine
  let v2 = &v; //Second Immutable Borrow is Fine
  let v3 = &mut v; //Mutable and Immutable Borrows are Not Fine
  println!("{:?}", v);
}
```

Ownership and Borrowing Summary

- Ownership once transferred, cannot be regained
- There is always one owner for value, which is responsible for dropping it
- Cannot mutate immutably borrowed content
- Cannot borrow both mutably and immutably at the same time
- Can immutably borrow any number of times

Why is all this necessary?

- Eliminates common class of memory errors. For eg: Double Free Error
- Avoid data races by allowing only one mutable borrow

Lifetimes

Brief Generics Recap

```
public <A> A genericFunction(A a1, A a2) {
  return a1;
}
```

Brief Generics Recap

```
public <A, B> A genericFunction(A a1, B a2) {
  return a1;
}
```

What do you expect this to do? [4]

```rust
pub fn main() {
  let v1 = vec!(1,2,3);
  let v2 = vec!(4,5,6);
  let result1 = do_something_1(&v1);
  let result2 = do_something_2(&v1, &v2);
  println!("{:?}", result1);
  println!("{:?}", result2);
}

fn do_something_1(v1: &Vec<u64>) -> &Vec<u64> {
  return v1;
}

fn do_something_2(v1: &Vec<u64>, v2: &Vec<u64>) -> &Vec<u64> {
  return v2;
}
```

Rust Befuddles Us

```rust
pub fn main() {
  let v1 = vec!(1,2,3);
  let v2 = vec!(4,5,6);
  let result1 = do_something_1(&v1);
  let result2 = do_something_2(&v1, &v2);
  println!("{:?}", result1);
  println!("{:?}", result2);
}

fn do_something_1(v1: &Vec<u64>) -> &Vec<u64> {
  return v1;
}
```

Rust knows the lifetime of returned vector should be same as input vector

```rust
fn do_something_2(v1: &Vec<u64>, v2: &Vec<u64>) -> &Vec<u64>  {
  return v2;
}
```

Rust complains it doesn't know about lifetime of returned vector

The ‘Fix‘

```rust
pub fn main() {
  let v1 = vec!(1,2,3);
  let v2 = vec!(4,5,6);
  let result1 = do_something_1(&v1);
  let result2 = do_something_2(&v1, &v2);
  println!("{:?}", result1);
  println!("{:?}", result2);
}

fn do_something_1(v1: &Vec<u64>) -> &Vec<u64> {
  return v1;
}

fn do_something_2<'a>(v1: &'a Vec<u64>, v2: &'a Vec<u64>) -> &'a Vec<u64>  {
  return v2;
}
```

We tell Rust that all vectors have the same lifetime

Why are Lifetimes necessary? [5]

```
pub fn main() {
  let v1 = vec!(1,2,3);
  let result;
  {
    let v2 = vec!(4,5,6);
    result = do_something(&v1, &v2);
  }
  println!("{:?}", result);
}

fn do_something<'a>(v1: &'a Vec<u64>, v2: &'a Vec<u64>) -> &'a Vec<u64> {
  return v2;
}
```

Why are Lifetimes necessary?

```
pub fn main() {
  let v1 = vec!(1,2,3);
  let result;
  {
    let v2 = vec!(4,5,6);
    result = do_something_1(&v1, &v2);
  } ⟶ v2 is dropped here
  println!("{:?}", result);
} ⟶ v1 and result are dropped here

fn do_something<'a>(v1: &'a Vec<u64>, v2: &'a Vec<u64>) -> &'a Vec<u64> {
  return v2;
}
```

The Lifetimes Fix

```rust
pub fn main() {
  let v1 = vec!(1,2,3);
  let result;
  {
    let v2 = vec!(4,5,6);
    result = do_something_2(&v1, &v2);
  }
  println!("{:?}", result);
}

fn do_something<'a, 'b>(v1: &'a Vec<u64>, v2: &'b Vec<u64>) -> &'b Vec<u64> {
  return v2;
}
```

Why Rust - The Good Stuff

- The Rust Lang Book [4]
- Beginner Friendly Ecosystem - Rustup, Cargo, VSCode Plugin (RLS Integration) etc.
- Community that is accomodating of newcomers and is always glad to help
- Lot of scope for contributions (For eg: Rust Lang Nursery)
- CLI Infrastructure powered by Rust (For eg: ripgrep, fd)

Questions?

References

[1] Jim Blandy and Jason Orendorff. "Programming Rust: Fast, Safe Systems Development". In: O'Reilly Media, 2017. Chap. Preface, p. xv. ISBN: 1491927283. URL: http://shop.oreilly.com/product/0636920040385.do.

[2] Jim Blandy and Jason Orendorff. "Programming Rust: Fast, Safe Systems Development". In: O'Reilly Media, 2017. Chap. Preface, p. xvi. ISBN: 1491927283. URL: http://shop.oreilly.com/product/0636920040385.do.

[3] The Rust Lang Community. "The Rust Programming Language - 2nd Edition". In: 2018. Chap. 4. URL: https://doc.rust-lang.org/book/second-edition/ch04-02-references-and-borrowing.html.

[4] The Rust Lang Community. *The Rust Programming Language - 2nd Edition*. 2018. URL: https://doc.rust-lang.org/book/.

[5] The Rust Lang Community. "The Rust Programming Language - 2nd Edition". In: 2018. Chap. 10. URL: https://doc.rust-lang.org/book/second-edition/ch10-03-lifetime-syntax.html.

Thank you!

Slides source available at: https://github.com/balajisivaraman/rust-in-peace