Rust

José Rodrigues - 7991083
Carlos Motta - 7991228

Por que criaram Rust?

Objetivos da linguagem

Rust surgiu com o objetivo de:

> “Projetar e implementar uma linguagem de programação de sistemas que seja prática, segura e concorrente.”

Grande atenção foi dada às falhas das demais linguagens existentes:

1. Muito pouca atenção é dada a **segurança**.
2. Suporte a **concorrência** é precário.
3. Falta de formas **claras** sobre como resolver determinados problemas.
4. Controle **limitado** de recursos.

Objetivos da linguagem

- Foco em programação de sistemas
- Eficiência
- Segurança de memória
- Concorrência e paralelismo (sem medo)

Objetivos da linguagem

Rust é uma alternativa que provém, ao mesmo tempo, **alta performance** e **um nível confortável de abstração**, enquanto melhora os quatro pontos citados como falhas das demais linguagens existentes.

O que é Rust?

A linguagem Rust

- Linguagem de baixo nível
- Não dogmática: trade-offs existem e podem ser melhor avaliados pelo programador (unsafe Rust)
- Fortemente tipada e inferência de tipos
    - Todas as variáveis têm seu tipo bem definido no momento de compilação
- Por padrão as variáveis são **imutáveis**, mas podem ser mutáveis

Conceitos fundamentais

- Ownership
- Borrowing
- Lifetimes

Ownership

Três regras básicas:

1. Todo valor possui uma variável que é seu *owner*.
2. Só existe **um** *owner* a cada momento.
3. Quando o *owner* sai de escopo, a memória é liberada.

Ownership

```
1  #[derive(Debug)]
2  struct BraveStruct {
3      courage: u32,
4      cowardness: u32,
5  }
6
7  fn go_to_die(some_struct : BraveStruct) {
8      println!("Last words: {:?}...", some_struct);
9      // Do nothing else and goes out of scope
10 }
11
12 pub fn main() {
13     let the_bravest;
14     println!("A hero is born!");
15
16     the_bravest = BraveStruct {
17         courage: 9001 as u32,
18         cowardness: 2 as u32,
19     };
20
21     go_to_die(the_bravest);
22     //println!("{:?} didn't die!", the_bravest);
23 }
```

```
rustc 1.17.0 (56124baa9 2017-04-24)

A hero is born!
Last words: BraveStruct { courage: 9001, cowardness: 2 }...

Program ended.
```

Ownership

```
rustc 1.17.0 (56124baa9 2017-04-24)
error[E0382]: use of moved value: `the_bravest`
  --> <anon>:24:34
   |
22 |     go_to_die(the_bravest);
   |               ----------- value moved here
23 |
24 |     println!("{:?} didn't die!", the_bravest);
   |                                   ^^^^^^^^^^^ value used here after move
   |
   = note: move occurs because `the_bravest` has type `BraveStruct`, which does not implement the `Copy` trait

error: aborting due to previous error
```

Ownership

Benefícios deste paradigma:

1. Certeza sobre qual parte do código é responsável por um valor na memória.
2. Liberação de memória automática e com segurança.
    - O compilador impede que um espaço liberado na memória seja acessado.
3. Efeito colateral: o programador se torna muito mais consciente sobre o gerenciamento da memória do computador.

Borrowing

- Como em C, podemos acessar o valor da variável (*owner*) via uma **referência**: “pegamos emprestado” o valor.
- Podem existir diversas referências **imutáveis** ao mesmo tempo.
- Se existe uma **referência mutável** de uma variável mutável ela é a **única referência** existente enquanto viver.

Borrowing

```
1  #[derive(Debug)]
2  struct Tutor {
3      name: &'static str,
4      position: &'static str,
5  }
6
7  fn get_phd(some_tutor : &mut Tutor) -> bool {
8      some_tutor.position = "Professor";
9      // Got his Phd -> return true :)
10     true
11 }
12
13 pub fn main() {
14     let alfredo = Tutor { name: "Goldman", position: "Professor" };
15     let mut pedro = Tutor { name: "Bruel", position: "TA" };
16
17     let our_professor = &alfredo;
18     let his_supervisor = &alfredo;
19     let our_TA = &pedro;
20
21     // Impossible to be a Professor while being a TA :(
22     get_phd(&mut pedro);
23 }
```

Borrowing

```
rustc 1.17.0 (56124baa9 2017-04-24)
error[E0502]: cannot borrow `pedro` as mutable because it is also borrowed as immutable
  --> <anon>:22:18
   |
19 |     let our_TA = &pedro;
   |                   ----- immutable borrow occurs here
...
22 |     get_phd(&mut pedro);
   |                  ^^^^^ mutable borrow occurs here
23 | }
   | - immutable borrow ends here

error: aborting due to previous error
```

Lifetimes

- Toda referência possui um tempo de vida
- Ela vive menos ou tanto quanto a variável (*owner*) do valor referenciado
- Evita-se *dangling references*
- Garante-se segurança de memória em tempo de compilação

Lifetimes

```
1  fn main() {
2      let future_ref;
3
4      {
5          let x = 5;
6          future_ref = &x;
7      }
8
9      println!("referencing: {}", future_ref);
10 }
```

```
rustc 1.17.0 (56124baa9 2017-04-24)
error: `x` does not live long enough
  --> <anon>:7:5
   |
6  |          future_ref = &x;
   |                       - borrow occurs here
7  |      }
   |      ^ `x` dropped here while still borrowed
...
10 | }
   | - borrowed value needs to live until here

error: aborting due to previous error
```

Lifetimes

```
1  fn main() {
2      let string1 = String::from("Paralela e Concorrente");
3
4      {
5          let string2 = String::from("mac5742");
6          let result = longest(string1.as_str(), string2.as_str());
7          println!("Apresentação de {}", result);
8      }
9  }
10
11 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
12     if x.len() > y.len() {
13         x
14     } else {
15         y
16     }
17 }
```

```
rustc 1.17.0 (56124baa9 2017-04-24)

Apresentação de Paralela e Concorrente

Program ended.
```

Paralelismo e Concorrência

Paralelismo e Concorrência

“Do not communicate by sharing memory; instead, share memory by communicating.”

-- Effective Go

Entretanto, Rust permite (e facilita) ambos os métodos de compartilhamento de estado entre threads.

Paralelismo e Concorrência - Canais

```rust
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let vals = vec![
            String::from("One"),
            String::from("message"),
            String::from("at"),
            String::from("a"),
            String::from("time"),
        ];

        for val in vals {
            tx.send(val).unwrap();
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

```
rustc 1.17.0 (56124baa9 2017-04-24)

Got: One
Got: message
Got: at
Got: a
Got: time

Program ended.
```

O transmissor `tx` é movido para o escopo da thread. Além disso, um valor enviado pelo canal tem sua *ownership* movida (enviada) também.

Paralelismo e Concorrência - Mutex

```rust
use std::sync::{Mutex, Arc};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = counter.clone();
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

```
rustc 1.17.0 (56124baa9 2017-04-24)

Result: 10

Program ended.
```

Outros Tópicos de Interesse

Pattern Matching

Macros

Módulos

Enums

Crates.io

Tratamento de Erros

Traits

Documentação Built-in

Reference Counting

Testes Built-in

Smart Pointers (Heap vs. Stack)

Unsafe Rust

Funções Genéricas

Tipos Genéricos

Referências

https://www.rust-lang.org/en-US/

https://doc.rust-lang.org/beta/book/second-edition/

https://play.rust-lang.org/

https://rustbyexample.com