Rust

Introduction to next generation programming language

Bartłomiej Małecki

Plan

1. Installation
2. Introduction
3. Rust and other language
4. Scoping rules
5. Concurrency
6. Smart Pointers

Getting stared

```
curl https://sh.rustup.rs -sSf | sh

git clone https://github.com/bmalecki/rust-getting-started
```

Why Rust?

- Fast - uses LLVM to generate byte code
- Safety – compiler prevents from commons programmer’s errors (not permit null pointers, dangling pointers, or data races)
- Multi-paradigm – gains from the best parts of various paradigms
- Modern – created in the 21st century

Interesting facts

- First Rust compiler was written in OCaml
- Developed by the Mozilla Foundation
- NPM (JavaScript repository) uses Rust
- The most loved language according to StackOverflow survey result (2019)

Hello world

```
fn main() {
    println!("Hello, world!");
}
```

Compilation

- rustc main.rs – rust compiler command
- cargo run – package manager command

Output

```
$ rustc main.rs
$ ls -al
-rwxrwxr-x 1 bartek bartek    2,4M Apr 16 19:30 main
-rw-rw-r-- 1 bartek bartek      45 Apr 15 23:06 main.rs
```

By default Rust uses static linking to compile program.

Output

```
$ rustc main.rs -C prefer-dynamic

$ export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/.rustup/toolchains/stable-
x86_64-unknown-linux-gnu/lib

$ ls -alh
-rwxrwxr-x 1 bartek bartek  14K Apr 16 19:37 main
-rw-rw-r-- 1 bartek bartek   45 Apr 15 23:06 main.rs
```

Rust can use dynamic linking but we have to add its libraries to list of directories where the system searches for runtime libraries.

Cargo

Cargo is Rust package manager.

```
cargo new hello-cargo
cd hello-cargo
cargo run

cargo run --bin binary_name # for multiple executable
```

Influences

- SML, OCaml: algebraic data types, pattern matching, type inference, semicolon statement separation
- C++: references, RAII, smart pointers, move semantics, monomorphization, memory model
- ML Kit, Cyclone: region based memory management
- Haskell (GHC): typeclasses, type families
- Newsqueak, Alef, Limbo: channels, concurrency
- Erlang: message passing, thread failure
- Swift: optional bindings
- Scheme: hygienic macros
- C#: attributes
- Unicode Annex #31: identifier and pattern syntax

https://doc.rust-lang.org/reference/influences.html

Comparison Rust to OCaml

Primitive types

Description	OCaml	Rust	Description	OCaml	Rust
Unit type	unit	()	16 bit wide unsigned integer	-	u16, u32, u64
Boolean type	bool	bool	32-bit IEEE 754 binary floating-point	-	f32
Signed integer, machine-dependent width	int	int	64-bit IEEE 754 binary floating-point	float	f64
Unsigned integer, machine-dependent width	-	uint	Unicode scalar value (non-surrogate code points)	-	char
X bit wide signed integer, two's complement	int32, in64	i8, i16, 132, i64	UTF-8 encoded character string.	-	str
8 bit wide unsigned integer	char	u8			

Operators

```
==  !=  <  >  <=  >=  && ||   // Rust
=   <>  <  >  <=  >=  && ||   (* OCaml *)


+  +  +  +  -  -  *  *  /  /  %     !    // Rust
+  +. @  ^  -  -. *  *. /  /. mod   not  (* OCaml *)


&    |   ^    <<     >>     !          // Rust
land lor lxor [la]sl [la]sr lnot       (* OCaml *)
```

Compound expressions

Name	OCaml	Rust	Name	OCaml	Rust
Record expression	{a=10; b=20}	R{ a:10, b:20}	Array expression, fixed repeats of first value		[x, ..10]
Record with functional update	{z with a=30}	R{a:30, .. z}	Index expression (vectors/arrays)	x.(10)	x[10]
Tuple expression	(x,y,z)	(x,y,z)	Index expression (strings)	x.[10]	x[10]
Field expression	x.f	x.f	Block expression	begin x;y;z end	{x;y;z}
Array expression, fixed size	[|x;y;z|]	[x,y,z]	Block expression (ends with unit)	begin x;y;() end	{x;y;}

Functions types and definitions

```
// Rust
// f : |int,int| -> int
fn f (x:int, y:int) -> int { x + y };

// fact : |int| -> int
fn fact (n:int) -> int {
   if n == 1 { 1 }
   else { n * fact(n-1) }
}
```

```
(* OCaml *)
(* val f : int * int -> int *)
let f (x, y) = x + y

(* val fact : int -> int *)
let rec fact n =
    if n = 1 then 1
    else n * fact (n-1)
```

Pattern match and guards

```
// Rust
match e {
  0             => 1,
  t @ 2         => t + 1,
  n if n < 10 => 3,
  _             => 4
}
```

```
(* OCaml *)
match e with
 | 0              -> 1
 | 2 as t         -> t + 1
 | n when n < 10 -> 3
 | _              -> 4
```

Recursion with side effects

```
// Rust
fn collatz(n:uint) {
  let v = match n % 2 {
     0 => n / 2,
     _ => 3 * n + 1
  }
  println!("{}", v);
  if v != 1 { collatz(v) }
}
fn main() { collatz(25) }
```

```
(* OCaml *)
let rec collatz n =
    let v = match n % 2 with
       | 0 -> n / 2
       | _ -> 3 * n + 1
    in
    Printf.printf "%d\n" v;
    if v <> 1 then collatz v

let _ = collatz 25
```

Record types, expressions and field access

```
// Rust
struct Point {
  x : int,
  y : int
}

let v = Point {x:1, y:2};
let s = v.x + v.y
```

```
(* OCaml *)
type Point = {
  x : int;
  y : int
}

let v = { x = 1; y = 2 };
let s = v.x + v.y
```

Algebraic data types

```rust
// Rust
enum Option<T> {
   None,
   Some(T)
}
// x : Option<t>
match x {
  None    => false,
  Some(_) => true
}
```

```ocaml
(* OCaml *)
type 't option =
    None
  | Some of 't

(* x : t option *)
match x with
  | None -> false
  | Some _ -> true
```

Lambda expression

```
// Rust
// ||int,int| -> int, int| -> int
fn ff(f:|int,int|->int, x:int) -> int
    { f (x, x) }

// m2 : |int| -> int
fn m2(n : int) -> int
    { ff ((|x,y| { x + y }), n) }
```

```
(* OCaml *)
(* (int*int->b)*int -> int *)
let ff (f, x) =
    f (x, x)

(* m2 : int -> int *)
let m2 n =
    ff ((fun(x,y) -> x + y), n)
```

OOP in Rust

- Encapsulation (struct)
- Polymorphism (traits) without inheritance

Struct

```
pub struct Counter {
    count: u32,
    id: u32
}
```

Methods

```
impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}
```

Traits

```
pub trait Draw {
    fn draw(&self);
}
```

Implementing traits

```
impl Draw for Counter {
    fn draw(&self) {
        println!("counter id: {} has {}", self.id,
self.count)
    }
}
```

Scoping rules

- Ownership
- Borrowing
- Lifetimes

Ownership

- Each value in Rust has a variable that's called its owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

Borrowing

- one or more references (&T) to a resource,
- exactly one mutable reference (&mut T).

Lifetimes

- The scope for which that reference is valid
- Most of the time, lifetimes are implicit and inferred
- We must annotate lifetimes when the lifetimes of references could be related in a few different ways

The Borrow Checker

```
{
    // Not compile
    let r;                    // ---------+-- 'a
                              //          |
    {                         //          |
        let x = 5;            // -+-- 'b  |
        r = &x;               //  |       |
    }                         // -+       |
                              //          |
    println!("r: {}", r);     //          |
}
```

The Borrow Checker

```
{
    let x = 5;                   // ----------+-- 'b
                                 //           |
    let r = &x;                  // --+-- 'a  |
                                 //   |       |
    println!("r: {}", r);        //   |       |
                                 // --+       |
}                                // ----------+
```

Lifetime Annotations in Function Signatures

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime elision rules

1. Each parameter that is a reference gets its own lifetime parameter.

```
fn foo<'a, 'b>(x: &'a i32, y: &'b i32)

fn foo(x: & i32, y: &i32)
```

Lifetime elision rules

2. If there is exactly one input lifetime parameter, that lifetime is assigned to all output lifetime parameters

```
fn foo<'a>(x: &'a i32) -> &'a i32)

fn foo(x: &i32) -> &i32
```

Lifetime elision rules

3. If there are multiple input lifetime parameters, but one of them is &self or &mut self because this is a method, the lifetime of self is assigned to all output lifetime parameters

```
fn foo<'a>(&’a self , x: &'a i32) -> &'a i32)

fn foo(&self, x: &i32) -> &i32
```

Concurrency

- Shared memory
- Message passing

Message passing

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

Shared-State Concurrency

- Using Mutexes to Allow Access to Data from One Thread at a Time

Smart pointers

- Box<T> for allocating values on the heap
- Rc<T>, a reference counting type that enables multiple ownership
- Ref<T> and RefMut<T>, accessed through RefCell<T>, a type that enforces the borrowing rules at runtime instead of compile time

Smart pointers

- Box<T> for allocating values on the heap
- Rc<T>, a reference counting type that enables multiple ownership
- Ref<T> and RefMut<T>, accessed through RefCell<T>, a type that enforces the borrowing rules at runtime instead of compile time

Smart pointers for concurrency

- Mutex<T>
- Arc<T>

Risk of:

- Deadlock

- RefCell<T>
- Rc<T>

Risk of:

- Creating reference cycle

Bibliography

1. https://doc.rust-lang.org/book
2. https://doc.rust-lang.org/rust-by-example
3. https://science.raphael.poss.name/rust-for-functional-programmers.html
4. https://github.com/rust-lang/rust
5. https://lifthrasiir.github.io/rustlog/why-is-a-rust-executable-large.html