Concepts of programming languages

Rust

Floris Schild, Mats Veldhuizen, Ruben Schenkuizen, Tobias van Driessel, Tom Freijsen

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Presentation Schedule

- Background
- Design principles
- What problems does Rust solve and how?
- Practical details

Background

- Personal project of Mozilla employee
- Sponsored by Mozilla Research
- First stable release (1.0) in 2015
- Used in Firefox and by Dropbox
- Most loved programming language - SO Developer Survey

Quick Overview

- Statically typed
- Functional **and** Imperative paradigms
- Strict language

Beautiful Quotes

Mozilla Research:

“Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.”

“It fuses the expressive and intuitive syntax of high-level languages with the control and performance of a low-level language.”

Features

Features

- Zero-cost abstractions

Features

- Zero-cost abstractions
- Move semantics

Features

- Zero-cost abstractions
- Move semantics
- Guaranteed memory safety

Features

- Zero-cost abstractions
- Move semantics
- Guaranteed memory safety
- Threads without data races

Features

- Zero-cost abstractions
- Move semantics
- Guaranteed memory safety
- Threads without data races
- Trait-based generics

Features

- Zero-cost abstractions
- Move semantics
- Guaranteed memory safety
- Threads without data races
- Trait-based generics
- Pattern matching

Features

- Zero-cost abstractions
- Move semantics
- Guaranteed memory safety
- Threads without data races
- Trait-based generics
- Pattern matching
- Type inference

Features

- Zero-cost abstractions
- Move semantics
- Guaranteed memory safety
- Threads without data races
- Trait-based generics
- Pattern matching
- Type inference
- Minimal runtime

Features

- Zero-cost abstractions
- Move semantics
- Guaranteed memory safety
- Threads without data races
- Trait-based generics
- Pattern matching
- Type inference
- Minimal runtime
- Efficient C bindings

Variable bindings

- Use *let* keyword to create binding
- Bindings are immutable by default
- Lhs not a name, but a *pattern*
- Type annotations

```
fn main() {
    let y = 3;
}
let (a, b) = (1, 2);
let mut x : u8 = 10; // Make x mutable
x = 255;
```

Functions

- Use *fn* keyword
- Statements vs expressions

```
fn square(x: i32) -> i32 {
 x * x // Expression
}
fn printSomething() {
println!("Something"); // Statement
}
```

Arrays and tuples

Arrays can be created like this:

```
let mut y = [4, 5, 6];
let second = y[1];
let mut z = [1; 10]; // Ten elements initialized to 1
```

Tuples are created as follows:

```
let mut tuple = (1, 2);
let x = tuple.0;
let y = tuple.1;

println!("Value of y is {}", y);

let (x, y, z) = (1, 2, 3); // Destructured tuple
```

Control flow

- *If, else, else if...*
- Loops, like *loop*, *while* and *for*

```
let x = 2;
let y = if x == 3 { 4 } else { 5 };
loop { println!("Infinite loop"); }

for x in 1..10 {}
for (index, value) in (1..10).enumerate() { }

let a = [1, 2, 3, 4, 5];
for elem in a.iter() {
    println!("the value is: {}", elem);
}
```

Vectors

- Dynamic arrays
- Allocated on heap (as opposed to arrays)

```
let mut v = vec![1,2,3]

for i in v {} // For loop takes ownership
for i in &v {} // Reference
for i in &mut v {} // Mutable reference
```

Structs

- Comparable to classes.
- Can also have methods and associated functions

```rust
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn print_xy(&self) {
        println!("x is {}, y is {}", self.x, self.y);
    }
}
let point = Point { x: 3, y: 6 };
point.print_xy(); // Shows x is 3, y is 6.
```

Match

- Comparable to switch
- Allows matching on expressions

```
let x: u32 = 3;

match x {
    1 => println!("one"),
    2 => println!("two"),
    _ => println!("three or more"),
}
```

Enums

- Define type and enumerate on its variants

```
enum Choice {
    Milk(i32),
    Tea(String),
}

let m = Choice::Milk(20);
```

Generics

- Generic structs, enums and functions.

```
fn takes_anything<T, U>(x: T, u: U) {}

struct Value<T> {
value : T
}

enum Choice<T> {
    Milk(T),
}
```

Traits

- Somewhat comparable to an interface
- Use trait bounds on generics

```rust
struct Square {
    side: f64,
}
trait HasArea {
    fn area(&self) -> f64;
}
impl HasArea for Square {
    fn area(&self) -> f64 {
        self.side * self.side
    }
}
fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

Much, much more..

- Smart pointers
- Concurrency
- Closures
- ...

Memory management

Different systems

- Garbage collection

Memory management

Different systems

- Garbage collection
- Smart pointers (reference counting)

Memory management

Different systems

- Garbage collection
- Smart pointers (reference counting)
- Ownership

Memory management

Different systems

- Garbage collection
- Smart pointers (reference counting)
- Ownership

Key difference: When are objects on the heap deallocated?

Ownership

- Every value has one variable that is its owner.

Ownership

- Every value has one variable that is its owner.

Example

```
let x = String::from("hello");
```

The variable *x* is the owner of the string object “hello” on the heap.

The string object will be deallocated when *x* goes out of scope.

Ownership

Another example

```
fn say_hello(name: String) {
    println!("Hello {}", name);
}

let x = String::from("Wouter");
say_hello(x);
say_hello(x); // Error!
```

Ownership

Another example

```
fn say_hello(name: String) {
    println!("Hello {}", name);
}

let x = String::from("Wouter");
say_hello(x);
say_hello(x); // Error!
```

The ownership is passed into *say_hello*, which deallocates our string when its scope ends.

Borrowing

- Pass a reference to the function
- Ownership is not transferred
- Immutable (by default)

```
fn say_hello(name: &String) {
    println!("Hello {}", name);
}

let x = String::from("Wouter");
say_hello(&x);
say_hello(&x); // Works!
```

Mutable references

```rust
fn double(value: &mut isize) {
    *value = *value * 2;
}

let mut x: isize = 3;
double(&mut x);
// x = 6
```

Slices

- Taking a reference to part of a collection

```rust
let some_numbers = [0, 1, 2, 3, 4, 5];
let slice = &some_numbers[0..3]; // [0, 1, 2]
```

Smart pointers

- Sometimes you may want more freedom
- For these situations, reference counting is also possible in Rust

```
let a = Rc::new(String::from("Blue"));
```

Smart pointers

- Box
- Enables recursive data types

Smart pointers

```
enum List {
    Cons(isize, Box<List>),
    Nil
}

let list = List::Cons(1,
                Box::new(List::Cons(2,
                Box::new(List::Cons(3,
                Box::new(List::Nil))))));

// list = (1 : (2 : (3 : [])))
```

Reference counting using smart pointers

```
struct Node {
    value: isize,
    next: Vec<Rc<Node>>
}

let d = Rc::new(Node { value: 8, next: vec![] });
let b = Rc::new(Node { value: 3, next: vec![Rc::clone(&d)] });
let c = Rc::new(Node { value: 5, next: vec![Rc::clone(&d)] });
let a = Rc::new(Node { value: 2, next: vec![Rc::clone(&b),
                                            Rc::clone(&c)] });
```

What problems does Rust (intend to) solve?

- Memory safety
- “Fearless” concurrency
- Performance

Memory safety

Null pointers

- Easy to forget
- The Option enum (similar to Maybe in Haskell)
- The Result enum

Memory safety

Null pointers

```
enum Option<T> {
    Some(T),
    None,
}
enum Result<T, E> {
 Ok(T),
 Err(E),
}
```

Memory safety

Dangling references

- No garbage collector!
- Borrowing rules/Lifetimes

Memory safety

Buffer overruns

- Safety
- Index in array
- Compile/Runtime checks

“Fearless” concurrency

Borrowing rules

- Only one owner
- No aliasing
- Easier debugging

“Fearless” concurrency

Message passing

```
let (tx, rx) = mpsc::channel();

thread::spawn(move || {
    let val = 5;
    tx.send(val).unwrap();
});

let received = rx.recv().unwrap();
```

“Fearless” concurrency

Shared state

```
let m = Mutex::new(0);

thread::spawn(move || {
    let mut num = m.lock().unwrap();
    *num = 5;
});
```

Performance

- No garbage collector
- Fewer run time checks

Practical

- Mature & intuitive package manager (cargo)
- Lots of libraries from a vibrant community
- i.a. IDE priority this year
- Basic linting and debugging
- Get started on https://doc.rust-lang.org
- Used because of safety, performance and being practical

Conclusion

- Fast & safe systems programming language
- Ownership & borrowing paradigms

Questions