Rust

Outline

1) General Information
2) Language & Features
3) Comparison

General

History

- 2006: personal project of Graydon Hoare
- 2009: sponsoring through Mozilla
- 2010: public announcement
- 2011: self-hosted compiler
- 2015: first stable release
  - before features got dropped frequently
- planned: releases every six weeks

Facts

- OpenSource
- extremely stable nightly builds
  - recommended to use those
- performance comparable to C++ code with similar precautions
- offers ForeignFunctionInterface
- compiler backend: LLVM

Facts

- expression based language
- no garbage-collection
  - variable have lifetimes
  - freeing happens automatically
- strong typesystem
  - type interference in function body
- data is immutable by default

Facts

- high order functions
  - functions can be arguments
- Polymorphism
  - Generics, work like in Java
- object system through traits and structured types
- zero-cost abstractions
  - done at compile-time

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

Goals

- (memory) safety
  - no dangling or null pointers
- control memory layout
  - programmer decides heap or stack
- avoidance of concurrency issues
  - no data-races

The Language

99 Bottles I

```
fn main() {
  let mut beers = 99;
  loop{
    let (first_line, second_line, tmp) = take_one_down(beers);
    println!("{}\n{}\n", first_line, second_line);
    beers = tmp;
    if beers == 0 {
      println!("No more bottles of beer on the wall, no more bottles of beer.");
      println!("Go to the store and buy some more, 99 bottles of beer on the
                                                                    wall.");
      break;
    }
  }
}
```

99 Bottles II

```
fn take_one_down(mut beers:i32) -> (String, String, i32){
  let first_line = if beers == 1{
    format!("{0} bottle of beer on the wall, {0} bottle of beer.", beers)
      }else{
    format!("{0} bottles of beer on the wall, {0} bottles of beer.", beers)
  };
  beers -= 1;

  let second_line = match beers {
    0 => format!("Take one down, pass it around, no more bottles of beer on the wall."),
    n @ 1 => format!("Take one down, pass it around, {} bottle of beer on the wall.", n),
    n @ _ => format!("Take one down, pass it around, {} bottles of beer on the wall.", n),
  };

  (first_line, second_line, beers)
}
```

What do we see?

- looks like a functional language
  - e.g. tuples, “`let`”, “`fn`”, “`->`”
- but basically C/Java- syntax

Syntax

- entry point is main
- variable declaration via ‘`let x = Foo`’
  - this is a statement!
- everything else expression
  - expression statement with ‘;’
  - block has value of last line
    - ➢ if line is a statement then ‘()’ (unit)

Syntax

- allows for instance:
  - `let x= if foo>=bar {foo}else{bar};`
  - `let y = {{{{4+6}}}};`
- tuples
  - returnable from function
  - can be used for destructuring:
    - `let (a,b,c) = foo();`

Features

Disclaimer

- Rust has many interesting features
  - exhaustive pattern matching
  - enum with arbitrary types
  - anonymous functions (Closures)
  - …
- only the most outstanding ones will be explained in details

“Unique” features

- Ownership / Borrowing
- Object system (Traits,...)
- Memory location

Ownership General

- bound data has one owner
  - only one able to access data
- ownership can be moved through rebinding
  - new variable is owner
  - function calls are rebinds!
- end of variable lifetime data is freed
  - can be seen as variable’s scope

Ownership - Example

```
fn main ()  {
    let x = vec![1,2,3];
    let y = x;
    println!("x:{:?}", x);
    // will not compile
}
```

Ownership Copy

- type can implement trait Copy
  - most primitives do
- rebounding copies data to new variable
  - old variable is still usable
- same effect if clone() is used
  - `let x = data.clone();`

Mutability

- variable are by default immutable
  - `let x = 4;`
  - `x = 5; // will not compile`
- “mut” keyword for mutability
  - `let mut x = 5;`
  - `x = 6; // will compile`

Ownership - Example

```
fn main ()  {
    let mut a = 2;
    let b = a;
    a = 5;
    println!("a:{}, b:{}", a, b);
    // will print ‘a:5, b:2’
}
```

Ownership

- calling function with a variable
  - ownership is moved
  - end of function data is freed/lost
- solutions
  - return variable
  - borrowing

Borrowing

- data can be borrowed
- end of scope owner gets data “back”
- more or less like using read/write references

Borrowing Types

- immutable reference -> &T
  - data can be read but not written
- mutable reference -> &mut T
  - data can be written and read
- examples:

```
let x = 5;
let y = &x;
*y +=  5; //error
```

```
let mut x = 5;
let y = &mut x;
*y += 5; //ok
```

Borrowing - Example

```
fn sum(vector:&Vec<i32>) -> i32{
    let mut sum = 0;
    for num in vector{
            sum = sum + num;
    }
    sum
}

fn main(){
    let vector = vec![1,2,3];
    let sum = sum(&vector);
    println!("{} is sum of {:?}", sum, vector);
}
```

Borrowing Rules

- borrow must last for smaller scope
  - binding order & code blocks matter!
- only one kind of borrow possible at a time:
  - 0..N references (`&T`)
  - 1 mutable reference(`&mut T`)

Conditions for data races

A. two or more pointers to same resource
B. one or more writing
C. no synchronization

Data race?

- borrowing rules:
  - 0..N immutable refs
  - 1 mutable ref
- data race condition:
  - 2..N pointers to resource
  - 1+ writing
  - no synchronization

Data race?

- borrowing rules:
  - 0..N immutable refs
  - 1 mutable ref
- data race condition:
  - 2..N pointers to resource
  - 1+ writing
  - no synchronization

Data race?

- borrowing rules:
  - 0..N immutable refs
  - 1 mutable ref
- data race condition:
  - 2..N pointers to resource
  - ~~1+ writing~~
  - no synchronization

Data race?

- borrowing rules:
  - 0..N immutable refs
  - 1 mutable ref
- data race condition:
  - 2..N pointers to resource
  - 1+ writing
  - no synchronization

=> data race impossible

Data race?

- borrowing rules:
  - 0..N immutable refs
  - 1 mutable ref
- data race condition:
  - 2..N pointers to resource
  - 1+ writing
  - no synchronization

Data race?

- borrowing rules:
  - 0..N immutable refs
  - 1 mutable ref
- data race condition:
  - ~~2..N pointers to resource~~
  - 1+ writing
  - no synchronization

Data race?

- borrowing rules:
  - 0..N immutable refs
  - 1 mutable ref
- data race condition:
  - ~~2..N pointers to resource~~
  - 1+ writing
  - no synchronization

=> data race impossible

Structs

- complex data-type
- boxes one or more types together
  - mutability of fields is determined through binding
- instanced through giving all key:value pairs
  - if some should be default values wrapper needed

Structs Example

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

fn main() {
    let mut point = Point { x: 0, y: 0 };
    // mut needed if we want to change fields
    point.x = 5;
}
```

Method Syntax

- function can be “mapped” to structs
  - avoids ‘`baz(bar(foo));`’
- first argument is in general self
  - refers to object
- method call: “`Foo.bar().baz();`”

Traits

- similar to abstract classes
- declare function which must be implemented
- can give default implementation
- can inherit from other traits
- can constrain Generics
- possible to add to primitive types

Example

```
trait Shape{
    fn area(&self)-> i32;
}

struct Square{
    x: i32,
}

impl Shape for Square{
    fn area(&self)-> i32{
        self.x * self.x
    }
}
```

Object system

- possible to follow OOP
  - interfaces/ abstract classes - Traits
  - class - Structs/Traits/Impl
  - inheritance - Traits

OOP

- theoretically:
  - put all together into one file
    - ➢ similar pattern as Java-classes
- however:
  - no need to so
  - some problems easier by sticking to other paradigms

Stack & Heap

- default bindings go on stack
- Box-type enables heap usage
  - memory is automatically allocated
  - deallocated at the end of the lifetime

Stack & Heap

```
fn main() {
    let s = 3;
    // saved on stack
    let h = Box::new(20);
    // saved on heap
}
```

Comparison

Rust & Java/C++

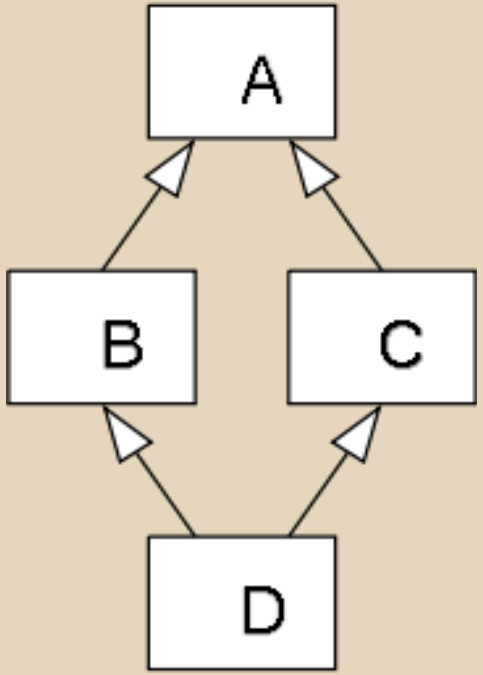
- syntax is similar
- a lot of features are comparable
  - borrowing - pointers/refs
  - structs,traits,implementation - classes
  - traits - interfaces
- still big differences
  - more functional
  - memory handling
  - avoids certain inheritance issues

Memory Management

- **Java** - garbage collector
  - reliable
  - slow
- **C** - manually
  - reliability depends on programmer
  - fast
- **Rust** - automatically at compile-time
  - reliable
  - fast

Diamond Inheritance

- C++/Java “impossible”
- Rust easy
  - A,B,C are traits
  - all traits can have default methods
  - D implements those traits



Rust & functional languages

- Rust has many typical features
  - closures - anonymous functions
  - tuples
  - type inference
  - variable binding
- but Rust is no pure functional language!
  - functions have side-effects
  - data is mutable

Overall

- combines concepts of system & high-level languages
- minimal safety issues
  - compiler checks for invalid pointers, non exhaustive pattern matching, etc.

IDEs or editors available?

- editors (via plugins):
  - syntax-highlighting
  - autocompletion (Racer)
  - e.g. vim, emacs, atom, ...
- IDE: SolidOak
  - neovim as texteditor
  - GUI with common editing features
  - racer autocompletion

Conclusion

- interesting features
- great usability for problems which depend on safety (& performance)
- high learning curve
  - using borrowing gets easier with time
- vivid community
- **worth to check it out!**

Bibliography

- http://rust-lang.org/
- http://rustbyexample.com/
- http://reddit.com/r/rust
- http://steveklabnik.com/fosdem2015/
- https://en.wikipedia.org/wiki/Rust_%28programming_language%29
- http://stackoverflow.com/questions/tagged/rust
- http://smallcultfollowing.com/babysteps/

Thank you.