A brief view on Rust and rustomata

Freitagsseminar

Christian Lewe

Fakultät Informatik
TU Dresden

26th January, 2018

Overview

1. Introduction to Rust
2. Ownership
3. Borrowing
4. rustomata
5. The end?

What is Rust?

- inspired by C++ and Haskell
- imperative basis
- functional aspects: pattern matching
- type system: statically typed, inference
- fast, close-to-metal, memory-safe, highly parallel
- zero-cost abstractions

https://www.rust-lang.org/logos/rust-logo-blk.svg

Example Rust program

```
fn main() {                   // main method
    let var0: u8 = 0;         // let binding
                              // immutable variable
    let var1 = "1";           // type inference
    let mut var2 = 2;         // mutable variable
    var2 = 3;

    let var3 = my_function(); // function call
    let var4 = var1.len();    // method call
    println!("{}", var0);     // macro! call
}
```

Control structures: `if`, `else`, `loop`, `while`, `for`, . . .

Function syntax

```
fn function_identifier(arg0: type0, ..., argK: typeK)
        -> return_type {
    statement1;        // with semicolon
    // ...
    return statementI; // early return
    // ...
    statementN         // final return
                       // without semicolon
    // ‘return statementN;’ is equivalent
}
```

Revision: scopes

```rust
fn main() {
    {
        let x = 1;
    }

    let y = x; // compile error !!!
}
```

```
error[E0425]: cannot find value ‘x‘ in this scope
6 |     let y = x;
  |             ^ not found in this scope
```

The ‘move’ problem

```
1 fn main() {
2     let x = String::from("Hello");
3     let y = string_len(x);    // x is moved
4     println!("{},␣world", x); // compile error !!!
5 }
6
7 fn string_len(z: String) -> usize {
8     z.len()
9 }
```

```
error[E0382]: use of moved value: ‘x‘
3 |     let y = string_len(x);
  |                        - value moved here
4 |     println!("{}, world", x);
  |                           ^ value used here after move
```

The ownership rules

1. Each value is bound to a variable, which we call its ‘owner’.
2. There can only be one such owner at a time.
3. When the owner goes out of scope, the value is freed.

Explanation of the ‘move’ problem

```
fn main() {
    let x = String::from("Hello"); // x is owner
    let y = string_len(x);         // z becomes owner
    println!("{},␣world", x);      // "Hello" was freed
}                                  // -> compile error

fn string_len(z: String) -> usize {
    z.len()
} // z leaves scope -> "Hello" is freed
```

1. Each value in Rust is bound to a variable, which we call its ‘owner’.
2. There can only be one such owner at a time.
3. When the owner goes out of scope, the value is freed.

Review of the ownership system

Upsides:

- automatic freeing of allocated space
- no garbage collection necessary
- no ‘use after free’ anomalies
- zero-cost abstraction

Downsides:

- different way of writing programs
- high learning curve

How to fix our code with Clone

- types can implement the Clone trait
- value is duplicated (deep copy)
- duplicate is assigned to a new owner (target of .clone())

```
fn main() {
    let x = String::from("Hello");
    let y = string_len(x.clone());
    println!("{},␣world", x);
}
```

Borrowing

- `clone()` takes time and uses memory
  → fast language?!
- faster alternative: reference to data (a.k.a. ‘pointer’)
- taking a reference is called ‘borrowing’
- reference is lifted once it goes out of scope

```rust
fn main() {
    let x = String::from("Hello");
    let y = string_len(&x);
    println!("{},␣world", x);
}

fn string_len(z: &String) -> usize {
    z.len()
}
```

Kinds of references

Immutable reference:

- created using ‘`&`’
- read-only access
- arbitrary number allowed at the same time

Mutable reference:

- created using ‘`&mut`’
- read-and-write access
- only one allowed at the same time

Only one kind of reference is allowed at any time for any value.
$\Longrightarrow$ *either* immutable *or* mutable

What is rustomata?

‘Framework for (weighted) automata with storage’

Features:

- accessible through CLI commands
- construct automata from grammars:
  - LCFRS → tree-stack automaton
  - CFG → push-down automaton
- parse input words, using an automaton:
  - tree-stack automaton $\overset{\text{word}}{\rightarrow}$ parse tree
  - push-down automaton $\overset{\text{word}}{\rightarrow}$ parse tree
- much more

`https://github.com/tud-fop/rustomata`

rustomata live demo

Rust has many more features!

- lifetimes
- traits
- iterators & closures
- error handling
- smart pointers
- parallel programming
- modules & crates
- package management with cargo
- unit and integration tests
- standard library
- unsafe Rust

You want to learn more?

- ‘The Rust Programming Language’
  `https://doc.rust-lang.org/book/`
- ‘Rust by Example’
  `https://rustbyexample.com`
- ‘The Rust Standard Library’
  `https://doc.rust-lang.org/std/`
- ‘rustup’
  `https://www.rustup.rs/`

Bonus slides!

6 More ownership

7 Traits

Ownership and Copy

- simple types can implement the Copy trait
- data on the stack
- size known at compile-time: bools, integers, chars, floats, . . .
- ‘automatic clone’

```
fn main() {
    let x = 1;
    let y = x; // x is copied
    println!("{}␣+␣1␣=␣2", x);
}
```

Generics and missing methods

```rust
struct Rectangle {
    width: u8,
    height: u8,
}

fn print_on_equal<A>(x: &A, y: &A) {
    if x.eq(y) {
        println!("Equal");
    }
}

fn main() {
    let r1 = Rectangle { width: 1, height: 2 };
    let r2 = Rectangle { width: 1, height: 3 };
    print_on_equal(&r1, &r2);
}
```

Generics and missing methods

```
6  fn print_on_equal<A>(x: &A, y: &A) {
7      if x.eq(y) {
8          println!("Equal");
9      }
10 }
```

```
error[E0599]: no method named ‘eq‘ found
        for type ‘&A‘ in the current scope
  |
7 |     if x.eq(y) {
  |          ^^
```

Traits

- restrict acceptable generic types
- types with a trait **must** implement all its methods
- similiar: ‘type classes’ in Haskell, ‘abstract classes’ in C++
- default implementations

```
pub trait PartialEq<A = Self> {
    fn eq(&self, other: &A) -> bool;  // required

    fn ne(&self, other: &A) -> bool { // default
        !self.eq(other)
    }
}
```

How to fix our code with PartialEq

- implement PartialEq for type Rectangle
- other has to be of type Rectangle
- print_on_equal requires PartialEq

```
impl PartialEq            for Rectangle {
    fn eq(&self, other            ) -> bool {
        (self.width == other.width) &
        (self.height == other.height)
    }
}

fn print_on_equal<A           >(x: &A, y: &A) {
    if x.eq(y) {
        println!("Equal");
    }
}
```

How to fix our code with PartialEq

- implement PartialEq for type Rectangle
- other has to be of type Rectangle
- print_on_equal requires PartialEq

```
impl PartialEq<Rectangle> for Rectangle {
    fn eq(&self, other: &Rectangle) -> bool {
        (self.width == other.width) &
        (self.height == other.height)
    }
}

fn print_on_equal<A           >(x: &A, y: &A) {
    if x.eq(y) {
        println!("Equal");
    }
}
```

How to fix our code with PartialEq

- implement PartialEq for type Rectangle
- other has to be of type Rectangle
- print_on_equal requires PartialEq

```
impl PartialEq<Rectangle> for Rectangle {
    fn eq(&self, other: &Rectangle) -> bool {
        (self.width == other.width) &
        (self.height == other.height)
    }
}

fn print_on_equal<A: PartialEq>(x: &A, y: &A) {
    if x.eq(y) {
        println!("Equal");
    }
}
```

Automatic implementation with derive

- many traits can be derived automatically
- #[derive(Trait1, Trait2, ...)]
- Eq, PartialEq, Ord, PartialOrd, Clone, Copy, ...

```rust
#[derive(PartialEq)]
struct Rectangle {
    width: u8,
    height: u8,
}

fn print_on_equal<A: PartialEq>(x: &A, y: &A) {
    if x.eq(y) {
        println!("Equal");
    }
}
```