Functional Rust

An Exploration

Lisa 'lislis' Passing

partial::Conf 2018

$ whoami

$ whoami

Developer at Open Knowledge Foundation DE

$ whoami

Developer at Open Knowledge Foundation DE

Game jammer, wannabe artist

$ whoami

Developer at Open Knowledge Foundation DE

Game jammer, wannabe artist

FP through ClojureScript

$ whoami

Developer at Open Knowledge Foundation DE

Game jammer, wannabe artist

FP through ClojureScript

Rustacean since early 2017

Rust!

Rust is

Rust is

- a systems programming language

Rust is

- a systems programming language
- imperative, multi-paradigm

Rust is

- a systems programming language
- imperative, multi-paradigm
- not garbage collected and does not require manual memory management

Rust is

- a systems programming language
- imperative, multi-paradigm
- not garbage collected and does not require manual memory management

  $\Rightarrow$ ownership and borrowing could fill an entire talk on their own

Rust is

- a systems programming language
- imperative, multi-paradigm
- not garbage collected and does not require manual memory management

  ⇒ ownership and borrowing could fill an entire talk on their own
- young, but popular (1, 2, 3)

Bringing together two worlds

Bringing together two worlds

- systems programmers who are used to bare metal

Bringing together two worlds

- systems programmers who are used to bare metal
- higher level language programmers who are used to abstractions

How does Rust achieve this?

How does Rust achieve this?

Taking good ideas and implementng them well,

How does Rust achieve this?

Taking good ideas and implementng them well,

while staying fast, secure and reliable.

Let's explore some functional features

Functions

Functions in Rust (pt 1)

Definition

Definition

```rust
fn main() {
  println!("Hello World");
}
```

Definition

```
fn head(v:Vec<u32>) -> u32 {
  v[0]
}

fn main() {
  let vector = vec![43, 567, 2, 34];
  println!("{}", head(vector));
}
```

Definition

```rust
fn head(v:Vec<u32>) -> u32 {
  v[0]
}

fn main() {
  let vector = vec![43, 567, 2, 34];
  println!("{}", head(vector));
}
```

- types of variables can be inferred

Definition

```rust
fn head(v:Vec<u32>) -> u32 {
  v[0]
}

fn main() {
  let vector = vec![43, 567, 2, 34];
  println!("{}", head(vector));
}
```

- types of variables can be inferred
- must annotate types of params and return values

Definition

```rust
fn head(v:Vec<u32>) -> u32 {
  v[0]
}

fn main() {
  let vector = vec![43, 567, 2, 34];
  println!("{}", head(vector));
}
```

- types of variables can be inferred
- must annotate types of params and return values
- returns last expression (*no trailing* `;`)

Recursion

Recursion


```
fn fibonacci(nth: i32) -> i32  {
  match nth {
    0 =>  { 0 },
    1 => { 1 },
    n => {
      fibonacci( n - 1 ) + fibonacci( n - 2)
    }
  }
}

fn main() {
  println!("{}", fibonacci(6));
}
```

Recursion

```rust
fn fibonacci(nth: i32) -> i32  {
  match nth {
    0 =>  { 0 },
    1 => { 1 },
    n => {
      fibonacci( n - 1 ) + fibonacci( n - 2)
    }
  }
}

fn main() {
  println!("{}", fibonacci(6));
}
```

```
8
```

Recursion

- Rust does not do tail-call optimization, so the stack is your limit!

Recursion

- Rust does not do tail-call optimization, so the stack is your limit!

- Partial application and currying à la Haskell are not possible

Functions in Rust (pt 4)

Higher order functions

Higher order functions

Take a function as argument and/or return a function

Higher order functions

Take a function as argument and/or return a function

- we'd have to know the types beforehand to annotate them (required by Rust)

Higher order functions

Take a function as argument and/or return a function

- we'd have to know the types beforehand to annotate them (required by Rust)
- the types of two named functions with the same signature are still different

Higher order functions

Take a function as argument and/or return a function

- we'd have to know the types beforehand to annotate them (required by Rust)
- the types of two named functions with the same signature are still different

How do we work around that?

Closures

Closures

- anonymous functions

Closures

- anonymous functions
- can capture their environment

Closures

- anonymous functions
- can capture their environment

```
let square = |num| {
  num * num
}
```

Closures

- anonymous functions
- can capture their environment

```
let square = |num| {
  num * num
}
```

- type annotations are optional

HOF cont.

HOF cont.

Find the sum of all odd squares that are smaller than 10,000. haskell,

rust

HOF cont.

Find the sum of all odd squares that are smaller than 10,000. haskell, rust

```
fn is_odd(n: u32) -> bool {
  n % 2 == 1
}

fn main() {
  let upper = 10000;

  let sum_of_squared_odd_numbers: u32 =
    (0..).map(|n| n * n)
         .take_while(|&n_squared| n_squared < upper)
         .filter(|&n_squared| is_odd(n_squared))
         .fold(0, |acc, n_squared| acc + n_squared);

  println!("Result: {}", sum_of_squared_odd_numbers);
}
```

HOF cont.

Find the sum of all odd squares that are smaller than 10,000. haskell, rust

```
fn is_odd(n: u32) -> bool {
  n % 2 == 1
}

fn main() {
  let upper = 10000;

  let sum_of_squared_odd_numbers: u32 =
    (0..).map(|n| n * n)
         .take_while(|&n_squared| n_squared < upper)
         .filter(|&n_squared| is_odd(n_squared))
         .fold(0, |acc, n_squared| acc + n_squared);

  println!("Result: {}", sum_of_squared_odd_numbers);
}
```

```
Result: 166650
```

Where do map, filter, fold etc come from?

Where do map, filter, fold etc come from?

Iterators!

Can you talk about Iterators more?

Can you talk about Iterators more?

Not yet!

Structs and enums

Structs

Structs

- custom data types

Structs

- custom data types

```
struct Point {
  x: f32,
  y: f32
}
```

Structs

- custom data types

```
struct Point {
  x: f32,
  y: f32
}
```

```
fn main() {
  let p = Point { x: 3.3, y: 4.8 }
}
```

Enums

Enums

- custom data type with *enumerated* possible values

Enums

- custom data type with *enumerated* possible values

```
enum Shape {
  Circle(Point, f32),
  Rectangle(Point, Point)
}
```

Enums

- custom data type with *enumerated* possible values

```
enum Shape {
  Circle(Point, f32),
  Rectangle(Point, Point)
}
```

```
fn main() {
  let circle = Shape::Circle(Point { x: 3.0, y: 4.0 }, 10.0);
}
```

Pattern matching

Recap

- matching on concrete values

Recap

- matching on concrete values

```
fn fibonacci(nth: i32) -> i32  {
  match nth {
    0 =>  { 0 },
    1 => { 1 },
    n => {
      fibonacci( n - 1 ) + fibonacci( n - 2)
    }
  }
}
```

Matching on enums

Matching on enums


```
struct Point {
  x: f32,
  y: f32
}

enum Shape {
  Circle(Point, f32),
  Rectangle(Point, Point)
}
```

Matching on enums


```
fn surface(s: Shape) -> f32 {
  match s {
    Shape::Circle(_, r) => { PI * r.powf(2.0) },
    Shape::Rectangle(Point { x: x1, y: y1 },
                     Point { x: x2, y: y2 }) => {
      (x2 - x1).abs() * (y2 - y1).abs()
    }
  }
}
```

Matching on enums

```
fn surface(s: Shape) -> f32 {
  match s {
    Shape::Circle(_, r) => { PI * r.powf(2.0) },
    Shape::Rectangle(Point { x: x1, y: y1 },
                     Point { x: x2, y: y2 }) => {
      (x2 - x1).abs() * (y2 - y1).abs()
    }
  }
}
```

- destructuring! We can access the inner values and bind them

Matching on enums


```
fn main() {
  let circle = Shape::Circle(Point { x: 3.0, y: 4.0 }, 10.0);
  println!("{}", surface(circle));

let rect = Shape::Rectangle(Point { x: 2.0, y: 4.0 },
  Point { x: 4.0, y: 1.0});
  println!("{}", surface(rect));
}
```

Matching on enums


```
fn main() {
  let circle = Shape::Circle(Point { x: 3.0, y: 4.0 }, 10.0);
  println!("{}", surface(circle));

let rect = Shape::Rectangle(Point { x: 2.0, y: 4.0 },
  Point { x: 4.0, y: 1.0});
  println!("{}", surface(rect));
}
```

```
314.15927
6.0
```

Generics and traits

Generics

Generics

An abstract stand-in for a concrete type

Generics

An abstract stand-in for a concrete type

```
fn head<T>(v: &Vec<T>) -> &T {
  v.first().unwrap()
}
```

Generics

An abstract stand-in for a concrete type

```
fn head<T>(v: &Vec<T>) -> &T {
  v.first().unwrap()
}
fn main() {
  let numbers = vec![43, 567, 2, 34];
  let strings = vec!["hello", "foo", "world"];
  println!("{}, {}", head(&numbers), head(&strings));
}
```

Generics

An abstract stand-in for a concrete type

```
fn head<T>(v: &Vec<T>) -> &T {
  v.first().unwrap()
}
fn main() {
  let numbers = vec![43, 567, 2, 34];
  let strings = vec!["hello", "foo", "world"];
  println!("{}, {}", head(&numbers), head(&strings));
}
```

```
43, hello
```

Generics

The example would break with an empty vector.

Generics

The example would break with an empty vector.

```
let empty: Vec<u32> = vec![];
println!("{}", head(&empty));
```

Generics

The example would break with an empty vector.

```
let empty: Vec<u32> = vec![];
println!("{}", head(&empty));
```

```
thread 'main' panicked at 'called `Option::unwrap()` on a `None` valu
```

Generics

The `Option<T>` type can help us.

Generics

The `Option<T>` type can help us.

```
fn head<T>(v: &Vec<T>) -> Option<&T> {
  v.first()
}
```

Generics

The `Option<T>` type can help us.

```
fn head<T>(v: &Vec<T>) -> Option<&T> {
  v.first()
}
```

```
fn main() {
  let empty: Vec<u32> = vec![];
  match head(&empty) {
    Some(val) => { println!("Head is {:?}", val); },
    None => { println!("No head here!"); }
  }

  let numbers = vec![43, 567, 2, 34];
  let _num_head = head(&numbers).expect("No head!");
}
```

Traits

Traits

Traits define behaviour that types can implement

Traits

Traits define behaviour that types can implement

Example `Display` trait for user facing string output

Traits

Traits define behaviour that types can implement

Example `Display` trait for user facing string output

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

fn main() {
  let p = Point { x: 12, y: 12};
  println!("{}", p);
}
```

Traits

Traits define behaviour that types can implement

Example `Display` trait for user facing string output

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

fn main() {
  let p = Point { x: 12, y: 12};
  println!("{}", p);
}
```

```
println!("{}", p);
|               ^ `Point` cannot be formatted with the default formatt
```

Traits

Implement `Display` on `Point`

Traits

Implement `Display` on `Point`

```rust
use std::fmt;

struct Point {
  x: i32,
  y: i32
}
impl fmt::Display for Point {
  fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
    write!(f, "({}, {})", self.x, self.y)
  }
}
fn main() {
  let p = Point { x: 12, y: 13 };
  println!("{}", p);
}
```

Traits

Implement `Display` on `Point`

```
use std::fmt;

struct Point {
  x: i32,
  y: i32
}
impl fmt::Display for Point {
  fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
    write!(f, "({}, {})", self.x, self.y)
  }
}
fn main() {
  let p = Point { x: 12, y: 13 };
  println!("{}", p);
}
```

```
(12, 13)
```

Trait bounds

Trait bounds

Constrain generic values with traits

Trait bounds

Constrain generic values with traits

```
fn exclamation<T: Display>(s: T) -> String {
  format!("{}!!!!!!", s)
}
```

Trait bounds

Constrain generic values with traits

```
fn exclamation<T: Display>(s: T) -> String {
  format!("{}!!!!!!", s)
}
```

```
fn main() {
  let p = Point { x: 12, y: 13 };
  println!("{}", exclamation(p));

  println!("{}", exclamation(42));
}
```

Trait bounds

Constrain generic values with traits

```
fn exclamation<T: Display>(s: T) -> String {
  format!("{}!!!!!!", s)
}
```

```
fn main() {
  let p = Point { x: 12, y: 13 };
  println!("{}", exclamation(p));

  println!("{}", exclamation(42));
}
```

```
(12, 13)!!!!!!
42!!!!!!
```

Back to Iterators!

The `Iterator` trait

Iterator trait

Iterator trait

handles logic of operating on a sequence

Iterator trait

handles logic of operating on a sequence

Iterators

- are thread safe

Iterator trait

handles logic of operating on a sequence

Iterators

- are thread safe
- are lazy

Iterator trait

handles logic of operating on a sequence

Iterators

- are thread safe
- are lazy
  - adaptors convert the type of Iterator

Iterator trait

handles logic of operating on a sequence

Iterators

- are thread safe
- are lazy
  - adaptors convert the type of Iterator
  - consumers kick off evaluation

Fibonacci cont.

Fibonacci cont.

```
struct Fibonacci {
  current: i32,
  index: i32
}
```

Fibonacci cont.


```
impl Fibonacci {
  fn new() -> Fibonacci {
    Fibonacci {
      current: 0,
      index: 1
    }
  }
}
```

Fibonacci cont.


```
impl Fibonacci {
  fn new() -> Fibonacci {
    Fibonacci {
      current: 0,
      index: 1
    }
  }
  pub fn nth(nth: i32) -> i32 {
    match nth {
      0 =>  { 0 },
      1 => { 1 },
      n => {
        Fibonacci::nth( n - 1 ) + Fibonacci::nth( n - 2)
      }
    }
  }
}
```

Fibonacci cont.


```
impl Iterator for Fibonacci {
  type Item = i32;

  fn next(&mut self) -> Option<Self::Item>{
    self.index += 1;
    let c = Fibonacci::nth(self.index);
    self.current = c;
    Some(self.current)
  }
}
```

Fibonacci cont.

Among the first 10 numbers of the Fibonacci sequence, is there one odd number larger than 100?

Fibonacci cont.

Among the first 10 numbers of the Fibonacci sequence, is there one odd number larger than 100?

```
fn main() {
  let f: bool = Fibonacci::new()
      .take(10)
      .filter(|n| { n % 2 == 1 })
      .any (|n| { n > 100});
  println!("{}", f);
}
```

Fibonacci cont.

Among the first 10 numbers of the Fibonacci sequence, is there one odd number larger than 100?

```
fn main() {
  let f: bool = Fibonacci::new()
      .take(10)
      .filter(|n| { n % 2 == 1 })
      .any (|n| { n > 100});
  println!("{}", f);
}
```

```
false
```

Zero cost abstractions

Zero cost abstractions

Use abstractions *without* additional performance cost!

We learned that Rust

We learned that Rust

- is an imperative language with features inspired by functional languages

We learned that Rust

- is an imperative language with features inspired by functional languages
- has a powerful type system

We learned that Rust

- is an imperative language with features inspired by functional languages
- has a powerful type system
- gets its functional feel from Iterators and closures

We learned that Rust

- is an imperative language with features inspired by functional languages
- has a powerful type system
- gets its functional feel from Iterators and closures
- gives us higher level concepts without having to about performance

Thank you!

📨 mail@lislis.de

🐘 lislis@toot.cat

🐙 https://github.com/lislis

📽 https://lislis.de/talks/partial-conf-2018/

Resources

- The Rust Programming Language
- Rust by Example
- Learn you a Haskell