



Rust Moravia

Functional Programming in Rust

Pavel Kučera (*1983)

- C++ developer since ~2006
- C#/.NET developer since ~2015
- Fan of FP since ~2018 (F#)
- Fan of Rust since 2023



Rust and FP

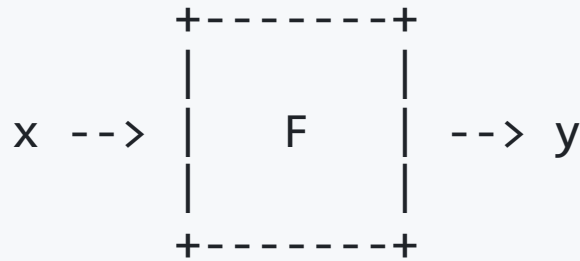
Let's see which FP features we can use in Rust.



What is Functional Programming?

- Function is a basic building block
- Function is a "first class citizen" so a function can be
 - called (obviously)
 - assigned to a variable / expression and call by name
 - passed as an argument
 - returned from another function
 - composed

What is a Function?



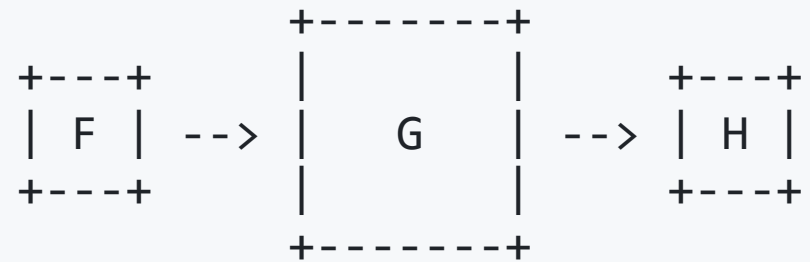
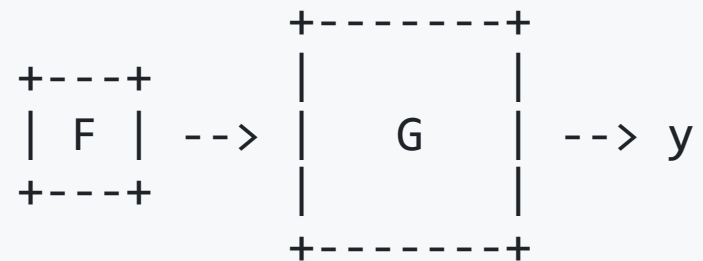
- A "box"
- Single input, Single output
- Every input produces an output (**totality**)
- Same input \Rightarrow same output (**stateless**)
- No side effects (**immutability**)

Quite an limitation! (For good reasons, FPs believe)

$$x \dashrightarrow \begin{array}{c} +---+ \\ | \text{ F } | \\ +---+ \end{array} \dashrightarrow y$$

$$x \dashrightarrow \begin{array}{c} +---+ \\ | \text{ F } | \\ +---+ \end{array} \dashrightarrow \begin{array}{c} +---+ \\ | \text{ G } | \\ +---+ \end{array} \dashrightarrow y$$

$$x \dashrightarrow \begin{array}{c} +---+ \\ | \text{ F } | \\ +---+ \end{array} \dashrightarrow \begin{array}{c} +---+ \\ | \text{ G } | \\ +---+ \end{array} \dashrightarrow y$$



01 - `let`

- **Necht'** (Czech for "let")
- Originates from LISP
- Resembles mathematics
- **immutable** by default (not a "variable")
- Expressions over statements
- keywords in other languages: `var` , `Dim` , `def` , `set` , `my`
 - `let` is a good choice (-:

01 - `let` - shadowing

```
#[test]
fn test_coffee_machine() {
    let coffee_machine = CoffeeMachine::from(CoffeeMachineSettings::default());
    let coffee_machine = coffee_machine.with_small_size();
    let coffee_machine = coffee_machine.with_max_strength();

    let coffee = coffee_machine.brew();
    assert!(coffee.is_ok());

    let coffee = coffee.unwrap();
    assert_eq!(coffee.water_ml, 30);
    assert_eq!(coffee.caffeine_mg, 40);
}
```

01 - `let`

Summary

- `let` keyword
- immutable
- shadowing (expression evaluation)

♥ A++

[demo_01_let.rs](#)

02 - partial (non-total) functions

Recap

- Every input produces an output (**totality**)
- What if not?
- Partial function == undefined behavior

Example

```
// Undefined behavior if bean_weight_mg == 0
pub fn count_beans(portion_weight_mg: i32, bean_weight_mg: i32) -> i32 {
    portion_weight_mg / bean_weight_mg
}
```

02 - partial (non-total) functions

- Rust behavior - panics
- many languages (F# included) have some sort of exceptions

Solution in rust

The only option in Rust is to create a total function instead:

```
pub fn count_beans(portion_weight_mg: i32, bean_weight_mg: i32) -> Option<i32> {  
    portion_weight_mg.checked_div(bean_weight_mg)  
}
```

- Client code is forced to check the returned `Option`
- The `checked_div` is a Rust `std` function. There are others as well for integer overflow, string parse, etc.

02 - partial (non-total) functions

Summary

- Rust is even more FP than e.g. F#
- `PartialEq`, `PartialOrd`

♥ A++

[demo_02_partial_fn.rs](#)

Algebraic data types

- Sum type
- Product type
- Tuple
- unit type
- Newtype
- type alias
- ...

enum - the **sum** type

- So called **Discriminated union**
- The **OR** type
- `enum` is not a great name but who cares
- Different kinds of enum variants
- Exhaustive pattern matching

example

```
pub enum CoffeeOrder {  
    Instant3In1,                // the unit variant  
    Espresso { size: Size, strength: Strength }, // the struct variant  
    PourOver(Temperature, Time), // the tuple variant  
    Other(BrewingMethod),       // another enum inside  
}
```


enum - why sum type?

- The cardinality (number of possible values) are the sum of the cardinalities of all the variants

Quiz I

What is the cardinality of `Bean` type?

```
pub enum Bean {  
    Arabica,  
    Robusta,  
    Blend(BlendType),  
}  
  
pub enum BlendType { Arabica50Robusta50, Arabica40Robusta60, Arabica60Robusta40 }
```

enum - pattern matching

```
impl Display for CoffeeOrder {
    fn fmt(&self, f: &mut Formatter) -> fmt::Result {
        match self {
            CoffeeOrder::Instant3In1 => write!(f, "Instant 3-in-1 coffee"),
            CoffeeOrder::Espresso { bean, strength } => write!(
                f,
                "Espresso with {:?} beans and {:?} strength",
                bean, strength
            ),
            CoffeeOrder::PourOver(temperature, time) => {
                write!(f, "Pour-over at {}°C for {} seconds", temperature, time)
            }
            CoffeeOrder::Other(method) => write!(f, "Other method: {:?}", method),
        }
    }
}
```

struct - the product type

- Also called record
- The AND type
- destructuring
- exhaustive pattern matching

example

```
pub struct Espresso {  
    pub size: Size,  
    pub strength: Strength,  
    pub milk: Option<Milk>,  
}
```

struct - why product type?

- The cardinality (number of possible values) is the product of cardinalities of all the fields

Quiz II

What is the cardinality of Espresso ?

```
pub struct Espresso { pub size: Size, pub strength: Strength }  
  
pub enum Size      { Small, Medium, Large, ExtraLarge }  
pub enum Strength  { Light, Medium, Strong }
```

struct - destructuring

```
pub fn make_espresso(espresso: Espresso) -> Coffee {  
    let Espresso {  
        size,  
        strength,  
        milk,  
    } = espresso;  
  
    todo!("Let's make it!")  
}
```

tuples

- anonymous structs
- mainly for intermediate results
- exhaustive pattern matching
- destructuring

```
pub fn choose_cup_color(esspresso: Espresso) -> String {  
    let color_args = (esspresso.size, espresso.strength);  
    let cup_color = match color_args {  
        (Size::Small, Strength::Strong) => "black",  
        (Size::Medium, Strength::Strong) => "red",  
        (Size::Large, Strength::Strong) => "brown",  
        _ => "white",  
    };  
    cup_color.to_string()  
}
```

03 - Algebraic data types

Summary

- **AND** and **OR** types
- The **New type** idiom
- Destructuring
- Pattern matching
- A struct in an enum



A++

[demo_03_algebraic_data_types.rs](#)

04 - Currying, partial application

Currying and partial application

Motivation

- Function has single input
- "I want more!" (-:
- A function can return a function...

Currying

- Named after [Haskell Brooks Curry](#)
- $F(x,y,z) \Rightarrow F(x)(y)(z)$
- What is it good for? **Partial application.**
- in FP languages, the currying is automatic
- in Rust, currying is not automatic
- We can make it manually
- We can use macros

Partial application

- Create a function G out of function F by fixing some of F's arguments
- Can utilize **Currying** if available
- May be done manually as well
- Examples
 - Building web requests (fixing base url)
 - Logging (fixing severity)
 - Making a coffee

Partial application - no currying

```
// standard function, no currying
pub fn make_espresso(strength: Strength, size: Size) -> Espresso {
    Espresso { strength, size }
}
```

```
let espresso = make_espresso(Strength::Strong, Size::Small);
```

```
let make_strong = |size| make_espresso(Strength::Strong, size);
```

```
let strong_small = make_strong(Size::Small);
let strong_medium = make_strong(Size::Medium);
let strong_large = make_strong(Size::Large);
```

Partial application - manual currying

```
// curried function
pub fn make_espresso(strength: Strength) -> impl Fn(Size) -> Espresso {
    move |size| Espresso { strength, size }
}
```

```
let espresso = make_espresso(Strength::Strong)(Size::Small);
```

```
let make_strong = make_espresso(Strength::Strong);
let espresso_small = make_strong(Size::Small);
let espresso_medium = make_strong(Size::Medium);
let espresso_large = make_strong(Size::Large);
```

Does not work for more than 2 arguments - [#99697](#)

Partial application - `curried` crate

```
use curried::curry;

#[curry]
pub fn make_espresso(beans: Beans, strength: Strength, size: Size) -> Espresso {
    Espresso { beans, strength, size, }
}

let make_robusta = make_espresso(Beans::Robusta);
let make_strong_robusta = make_robusta(Strength::Strong);

let strong_small = make_strong_robusta(Size::Small);
let strong_medium = make_strong_robusta(Size::Medium);
let strong_large = make_strong_robusta(Size::Large);
```

Utilizes `Box<dyn FnOnce>`

Does not work either - because of `FnOnce` .

04 - Currying, partial application

Summary

- Currying not supported natively
 - Workarounds exists but are limited
- Partial application possible, though



B-

[demo_04_partial_application.rs](#)

[demo_04_curried.rs](#)

Recap

Pure function, Total function, immutability, `let`, shadowing, partial functions, sum type, product type, tuple, pattern matching, destructuring, cardinality, currying, partial application,

05 - Todo

```
todo!("Recursive types")
todo!("new type")
todo!("unit type")
todo!("Higher order functions")
todo!("Function composition")
todo!("Function passing")
todo!("Recursion")
```


06 - Q/A

