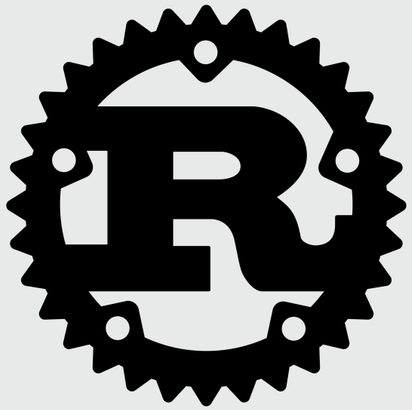

A small dive in Rust

@ Midis de la bidouille

Rust Motto

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

Featuring

- zero-cost abstractions
- move semantics
- guaranteed memory safety
- threads without data races
- trait-based generics
- pattern matching
- type inference
- minimal runtime
- efficient C bindings

History

— Toy langage of Graydon Hoare (2006),
— Endorsed by Mozilla (2009),
— Self-compiling (2011),
— Version 1.0 (2015),
— Production years (Mozilla (Firefox...), Atlassian, Coursera, Dropbox, Samsung, Pollen Robotics...)
— **Today**: version 1.27

Rust community = ❤️

As Rust's community grows, three large camps form:

— Ex C++ users,
— Ex scripting language users,
— Ex functional programmers.

In some ways, Rust is a combination of all three of these things.

Friendly community : code of conduct.

Rust and me

- I love programming languages.
- I contributed to Rust up to version 0.7. Lots of changes have happened since then.
- Nowadays I use Rust as much as possible for:
  - Replacing Python scripts,
  - Scientific computing (even though the ecosystem is quite young). I did a partial rewrite of our numerical framework (`Cadmos`) in Rust (which calls the original `C++` version in some places).

Installing Rust

The best way to install Rust is with rustup, a Rust version manager. To install it type:

```
$ curl https://sh.rustup.rs -sSf | sh
```

To keep your rust up-to-date with the latest release of rust, type

```
$ rustup update
```

Code may also be run on a playground at http://play.rust-lang.org/.

Cargo

Cargo is a tool that helps you develop Rust programs. It does several things:

— Runs tasks: `cargo build` (compile your app), `cargo test` (test your app), `cargo run` (run you app),
— Starts a project: `cargo new`, `cargo init`

Cargo is also the package manager for Rust. This means that you can use Cargo to install and manage bits of other people's code.

— A program or library is called a "crate".
— A package contains one or more crates.
— You can find Crates at http://crates.io.
— You list the Crates you want to use in the `Cargo.toml` file.

Hello world and basic syntax

If we look at the mandatory "hello world":

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

— `fn` declares a function.

— Function bodies go in curly braces: {}.

— The `println!` macro (the ! means it's a macro) prints stuff to the screen.

— Strings go inside double quotes "".

— Lines end with a semicolon ;

Rust variables are not mutable by default:

```
fn main() {
  let x = 5;
  x = 6 ; // -> error[E0384]: cannot assign twice to immutable variable `x`
  let mut y = 6;
  y = 7; // OK
}
```

Rust must know the type of all variables at compile time. Types are often infered by the compiler.

```
let trois: u32 = 3;
```

Scalar Types:

- integer (`i32`, `u64`, `usize`...),
- floats (`f32`, `f64`),
- bool (`true`/`false`).

Tuples

A tuple is a general way of grouping together some number of other values with a variety of types into one compound type.

```
fn main() {
    let tup: (u32, f64, isize) = (500, 6.4, 1);

    let (x, y, _) = tup;

    println!("y is: {}, x is: {}", y, tup.0);
}
```

Another way to have a collection of multiple values is with an array. Unlike a tuple, every element of an array must have the same type.

```
  fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
    let out_of_bounds = a[1515];
  }
/// -> thread 'main' panicked at 'index out of bounds: the len is 5 but the index is 1515', src/main.rs:6:25
```

Arrays in Rust have a fixed length (see `Vec` for dynamic arrays.)

— **String:** UTF-8

```
let st=String::from("hello 🌍");
```

— **HashMaps:**

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from("France"), 1);
scores.insert(String::from("Pérou"), 0);

for (key, value) in &scores {
  println!("{}: {}", key, value);
}
```

Vectors

Vectors allow you to store more than one value in a single data structure that puts all the values next to each other in memory. Vectors can only store values of the same type.

```
let mut v: Vec<i32> = Vec::new();

v.push(5); v.push(6);
v.push(7); v.push(8);
```

Using a macro, initialization code could be much simplified

```
let v = vec![1, 2, 3]; // + type inference
for i in &v { // imutable reference to vec elements
  println!("{}", i);
}
```

Functions

Functions take arguments and return only one value.

```
fn main() {
  let orga="Inria";

  println!("Successfully transformed organization : {}", villanize(orga));
}

fn villanize(o: &str)->String {
  o.replace("ia", "🤖")
}
```

Here `&str` is a slice (reference to a string, same type as `orga`).
Rust does not need headers...

Conditionals are familiar:

```
if x {...} else {...}

while x {
   ...
}
```

and are expressions!

```
let v = if x>2 {1} else {2};
```

⚠️ Ownership

Rust has rules regarding ownership

— Each value in Rust has a variable that’s called its owner.
— There can only be one owner at a time.
— When the owner goes out of scope, the value will be dropped.

```
let x = "Hello world";
{
  let y = "from Inria";
}
println!("{} {}", x, y); // -> error[E0425]: cannot find value `y` in this scope
```

Rust ensures that every resource is bound to a single value. In other words, if a resource is reassigned from one variable to another, the first variable is no longer valid.

```
fn main() {
    let x = String::from("Hello, World!");

    println!("{}", x); // x is valid here

    let y = x; // the resource assigned to x is moved to y

    println!("{}", x); // -> error[E0382]: use of moved value: `x`
}
```

If you need a copy of `x`, it has to be explicitely written:

```
let y = x.clone();
```

Move semantics

Rust is moving by default for non POD types:

```
fn main() {
  let event=String::from("Midis de la bidouille");      // Object is created here
  publicize(event);                                     // And moved inside publicize function
  println!("Huge audience at {}", event);
}
fn publicize(e: String) {
  println!("Please please, don't forget {}!", e);
}
```

does not compile with

```
error[E0382]: use of moved value: `event`
 --> src/main.rs:4:37
  |
3 |     publicize(event);
  |               ----- value moved here
4 |     println!("Huge audience at {}", event);
  |                                     ^^^^^ value used here after move
  |
  = note: move occurs because `event` has type `std::string::String`, which does not implement the `Copy` trait
```

If you want to use `event` after the function call, you have to borrow it!

References and borrowing

— No borrow can outlive the original owner's scope.
— You can have as many immutable borrows, or references, to a resource at any given time.
— You can have a single mutable borrow, or reference, to a resource at any given time.

```
fn main() {
  let event=String::from("Midis de la bidouille");
  publicize(&event);                              // & are explicit!
  println!("Huge audience at {}", event);
}

fn publicize(e: &String) {
  println!("Please please, don't forget {}!", e);
}
```

In your early days in Rust, we will end up fighting the borrow checker that strictly applies these rules. A mutable borrow is allowed just once.

```
publicize(&mut event);
fn publicize(e: &mut String) {
```

References are dereferenced by `*` (not needed after `.`). They cannot be null (no `NULL` or `nullptr`), nor converted from integers.

Slices

Slices let you reference a contiguous sequence of elements in a collection rather than the whole collection.

For strings:

```
let s = String::from("hello world");
let hello = &s[0..5];
```

In fact `"hello world"` is a slice of type `&str` pointing to a static part of memory.

For arrays:

```
let a = [1, 2, 3, 4, 5];
let slice = &a[1..3];
```

Lifetimes and the borrow checker: safely returning references

The main aim of lifetimes is to prevent dangling references, which cause a program to reference data other than the data it's intended to reference. The following incorrect code does not compile.

```
    let r;
    {
        let x = 5;
        r = &x;
    }
    println!("r: {}", r);
```

with

```
error[E0597]: `x` does not live long enough
  --> src/main.rs:7:5
   |
6  |         r = &x;
   |             - borrow occurs here
7  |     }
   |     ^ `x` dropped here while still borrowed
...
10 | }
   | - borrowed value needs to live until here
```

Consider a function that takes two parameters by reference and returns one of them.

```
#[derive(Debug)]                                    // Trait derivation
struct Person(String);                              // Anonymous struct

fn prettiest(hb1: &Person, hb2: &Person)->&Person {
  hb1 // Life is unfair
}

fn main() {
  let f=Person(String::from("Francis"));
  let g=Person(String::from("Gérard"));

  println!("And Miss universe is {}", prettiest(&f, &g));
}
```

This will not compile because what would happen if hb1 or hb2 did not have the same lifetime?

```
Compiling playground v0.0.1 (file:///playground)
error[E0106]: missing lifetime specifier
 --> src/main.rs:4:43
  |
4 | fn prettiest(hb1: &Person, hb2: &Person)->&Person {
  |                                           ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature does not say whether it is borrowed from `hb1` or `hb2`
```

The solution is to tell Rust that the two parameters have to have the same lifetime named a.

```
fn prettiest<'a>(hb1: &'a Person, hb2: &'a Person)->&'a Person {
  hb1 // Life is unfair
}
```

— One can have several lifetimes.

```
fn f<'a, 'b>(r: &'a i32, s: &'b i32) -> &'a i32 { r } »
```

— Struct and method definitions can also hold references.

```
struct BoringStruct<'a> {              // Lifetime of struct is the same as rr
  id: String,
  rr: &'a BoringRef
}
```

— When there is no ambiguity, lifetimes can be elided.

```
fn first_third(point: &[i32; 3]) -> (&i32, &i32) {
  (&point[0], &point[2])
}
// is the elided version of
fn first_third<'a>(point: &'a [i32; 3]) -> (&'a i32, &'a i32)
```

Pointers

In Rust, the different smart pointers defined in the standard library provide extra functionality beyond that provided by references. They are rarely needed in practice.

— `Box<T>` for allocating values on the heap,

— `Rc<T>`, a reference counted type that enables multiple ownership,

— `Ref<T>` and `RefMut<T>`, accessed through `RefCell<T>`, a type that enforces the borrowing rules at runtime instead of compile time.

```
  fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
  }
// -> b = 5
```

The pointers are autodereferenced when there is no ambiguity (as in the previous line).

From an arbitrary address, one could dereference it. Rust prevents it:

```
let address = 0x012345usize;
let r = address as *const i32;
println!("{}",*r); // -> error[E0133]: dereference of raw pointer requires unsafe function or block
```

It can be done but then the code has to be marked unsafe. Functions calling other unsafe languages (e.g. C/C++) have also to be marked unsafe. Self-shooting in the foot is much more difficult in Rust.

Gathering data and functions: Structs

Rust structs resemble `struct` types in `C/C++`or classes in `Python`. They are defined as follows

```
struct Rep {
  name: String,
  email: String,
  genre: Genre,                                                    // BRB
  age: u32
}

// Definition of functions on the Rep struct
impl Rep {
  // Class function
  fn new(name: String, email: String, genre: Genre, age: 32)->Rep {
    Rep {name, email, genre, age}
  }
  // Method (with self, &self, &mut self as first parameter )
  fn is_young(&self)->bool {
    self.age < 35
  }
}
```

and that's it! There is no specific constructor or destructor.

Enum

C-style enums are also available and they can be enriched with data

```
enum Genre {
  Male, Female, Neutral, AI(String)
}
fn main() {
  let gg=Genre::AI(String::from("👽"));
  greets(gg);
}
// Matching clauses have to be exhaustive!
fn greets(g: Genre) {
  match g {                                          // Rust has powerful pattern matching features
    Genre::Male=>println!("Hello 👨"),
    Genre::Female=>println!("Hello 👧"),
    Genre::AI(desc)=>println!("Hello {}!", desc)
  };
}
```

does not compile with

```
error[E0004]: non-exhaustive patterns: `Neutral` not covered
--> src/main.rs:12:11
|
12 |     match g {
|           ^ pattern `Neutral` not covered
```

Option

Option is an enum defined as follows:

```
enum Option<T> {
  None,
  Some(T)
}
```

and indicates an optional value (like an optional parameter).

```
  fn main() {
    let s=Some("thing");

    if let Some(v)=s {
        println!("Some{}",v);
    }
    else {
        println!("Nothing!");
    }
  }
```

Result

Result is defined as

```
enum Result<T,Error> {
  Err(Error),
  Ok(T)
}
```

and is commonly used as a function return value for efficient error handling.

```
fn i_could_fail()->Result<T,Error> {
  ...
  if success {
    return Ok(value);
  }
  else {
    return Err(errror);
  }
}
```

Rust paradigm

— Rust is not a really object oriented language:
  — No inheritance of struct -> use composition or generics
  — We have encapsulation (structs may remain private to their module)
— Rust has many functional languages features.

Iterators

— `map`, `filter`, `enumerate`, `zip`...
— `collect`, `find`...
— `partition`, `collect`...

```
let reps=vec![...];

let reps_ages_as_🐶=reps
  .iter()
  .map(|&r| r.age/7)
  .collect();
```

Closures

Rust has closures, lightweight function-like values.

```
let is_even = |x| x % 2 ==0;

assert_eq!(is_even(14),true);
```

Rust infers the argument and return types.

Generics

Generic data types

In Rust it’s possible to write code that operates on values of many different types, even types that haven’t been invented yet.

We have already met generic in `Enum` and it works for `struct` as well.

```
struct Point<T> {
  x: T,
  y: T,
}

fn main() {
  let integer = Point { x: 5, y: 10 };
  let float = Point { x: 1.0, y: 4.0 };
}
```

Traits are Rust's take on interfaces or abstract base classes.

```
// Generic function where the trait is denoted by T (may be none if list is empty)
fn oldest<T>(persons: &[T]) -> Option<u32> {
    persons
      .iter()
      .map(|&p| p.how_old())
      .max()
    }

fn main() {
  let reps: Vec<Rep>=...;

  println!("The oldest one {} year old", oldest(&reps));
}
```

Previous code won't work as Rust has no guarantee that a passed type has an `age method`. One has to constrain the type.

```
trait HasAge {
  fn how_old(&self)->u32;
}
impl HasAge for Rep {
  fn how_old(&self)->u32 {
    self.age
  }
}
```

Traits can also provide default implementation (but no default member variables -> use accessors methods).

Operator overloading is implemented by defining standard `Add`, `Sub`, `Ord`..., traits for structs.

```
impl Add<Field> for f64 {
    type Output = Field;
    fn add(self, other: Field) -> Field {
      other + self
    }
}
```

Trait derivation

Using derivation, standard traits (and others) are often automagically implemented. Let's say we want to serialize/deserialize our `Rep` struct to `json`.

```
// Serde is a crate devoted to de-serializing to various formats.
extern crate serde;
extern crate serde_json;
#[macro_use]
extern crate serde_derive;
use serde_json::Error;
// Here we automatically derive 3 traits
#[derive(Serialize, Deserialize, Debug)]
struct Rep {
  ...
}
// Serialization occurs there without anything to write on our side
fn rep_to_json(r: &Rep)->Result<(), Error> {
  let f = serde_json::to_string(r)?;
  println!("{}",f);
  Ok(())
}
```

Trait objects

Generics are zero-cost abstraction, there is no penalty at run time. What if we want a collection of types satisfaying a given trait? **Trait objects**.

```
trait HasHairs {}                   // We define a trait
struct Rep;
impl HasHairs for Rep {}            // and its implementations (1)
struct Sed;
impl HasHairs for Sed {}            // (2)

struct HairyPeople {                // We collect a vector of objects satisfying the trait
  members: Vec<Box<HasHairs>>       // A Vec of Trait Object
}
```

This is using dynamic dispatch so it has a cost at runtime.

Interfacing with external libraries (FFI)

Building external sources

Before compiling Rust code, one can build from other sources (e.g. C/C++ codes) and link the Rust binary with it. This is achieved by adding a `build.rs` file inside the crate sources.

```
extern crate cmake;                                                    // CMake support for building
use std::env;                                                          // Environmental variables support

fn main() {
    let eigen_dir = env::var("EIGEN_DIR").unwrap_or(String::from("/usr/local/include/eigen3"));

    let dst = cmake::Config::new("cadmos")
          .define("EIGEN3_INCLUDE_DIR", eigen_dir)
          .define("CMAKE_CXX_FLAGS", "-O -g -Wall -std=c++11")
          .define("BUILD_BINDINGS", "ON")
          .build();

    println!("cargo:rustc-link-search=native={}/lib", dst.display()); // Instructions for Rust linker
    println!("cargo:rustc-link-lib=cadmos_ffi");
}
```

This is some code from my Rust port of our numerical framework.

Declaring alien interface

Interfacing is then a matter of wrapping the C interface.

```
use libc::c_double;                                  // Support for c_double (convert to f64)
use core::MeshDesc;
use field::Field;

extern "C" {
    fn solve_diff_poisson_dirichlet(                 // Routine defined in an extern C library
        desc: *const MeshDesc,
        sol: *mut c_double,
        rhs: *const c_double,
        dir: *const c_double,
    );
}
```

This can be done by a standard tool called bindgen that generates Rust FFI bindings to C (and some C++) libraries automatically.

Calling external routines

The external library is called from Rust by:

```
pub fn solve_poisson_dirichlet(&self, rhs: &Field, dir: &Field) -> Field {
  let mut res = Field::from_value(&self.grid, 0.0);
  unsafe {
      solve_diff_poisson_dirichlet(
          &self.grid,
          res.data.as_mut_ptr(),                    // Returns an unsafe mutable pointer to a slice buffer
          rhs.data.as_ptr(),                        // Returns a raw pointer to a slice buffer
          dir.data.as_ptr(),
      );
  }
  res
}
```

Note the unsafe keyword. The goal is to minimize as much as possible these unsafe calls to benefit from Rust's safety.

Concurrency

Thanks to Rust ownership model and type checking, most concurrency problems (data races, deadlocks) are prevented (at compile time): **fearless concurrency**. There are three ways to use Rust threads:
- Fork-join parallelism
- Channels
- Shared mutable state

Some crates (like crossbeam, or rayon) make writing multithreaded code even easier.

```
extern crate rayon;
use rayon::prelude::*;

// do 2 things in parallel
let (v1, v2) = rayon::join(fn1, fn2);

// do N things in parallel
giant_vector.par_iter().for_each(|value| {
  do_thing_with_value(value);
});
```

There so much more

— Macros
— Modules and visibility
— Advanced closures
— Associated types
— Testing
— Advanced error handling
— Async I/O
— Ecosystem

Are we there yet?

- Steep learning curve.
  - Rigidity (e.g. sorting a vector of floats).
- IDE
- Slow compilation times
  - It gets better.
- Immature ecosystem for scientific computing
  - Numpy equivalent (ndarray) but no standard linear algebra library.
  - Tensorflow bindings.
  - No-const generics.
  - SIMD (as of version 1.27).

References

- Website http://www.rust-lang.org,
- The Rust book (https://doc.rust-lang.org/book/second-edition/index.html)
- User forums (http://users.rust-lang.org),
- Rust community's crate registry (http://crates.io).

Thank you for your attention

Olivier.Saut@inria.fr

http://osaut.monc.fr