RUST in peace : à la découverte d'un nouveau langage.

François-Xavier Dupé

(LIF/Aix-Marseille Université, France)

Journée LIFTech ~~mai~~ juin 2016

Tremble you weakings, cower in fear
I am your ruler, land, sea and air
Immense in my girth, erect I stand tall

Tremble you weakings, cower in fear
I am your ruler, land, sea and air
Immense in my girth, erect I stand tall

Rust in peace. . . Polaris, Megadeth

RUST : un nouveau langage

Créé en interne par Graydon Hoare puis sponsorisé par Mozilla (vers 2009). Langage de programmation :

- multi-paradigme ;
- compilé (repose sur LLVM) ;
- concurrent ;
- fonctionnel ;
- impératif ;
- structuré.

RUST : un nouveau langage

Créé en interne par Graydon Hoare puis sponsorisé par Mozilla (vers 2009). Langage de programmation :

- multi-paradigme ;
- compilé (repose sur LLVM) ;
- concurrent ;
- fonctionnel ;
- impératif ;
- structuré.

Maintenant utilisé pour Servo !

Les avantages

Présenté comme permettant :

- des abstractions faciles ;
- une sémantique pour déplacer la propriété des variables ;
- une garantie au niveau de la sécurité de la mémoire ;
- des processus légers (avec gestions des données partagées) ;
- des interfaces génériques (via des *traits*) ;
- l'inférence des types ;
- ...

Plan de rouille

1. Exemple de base
2. Les traits
3. Les « Generics »
4. Gestion de la mémoire

Plan de rouille

Simple ?

```
fn main() {

    let program = "+ + * - /";
    let mut accumulator = 0;

    for token in program.chars() {
        match token {
            '+' => accumulator += 1,
            '-' => accumulator -= 1,
            '*' => accumulator *= 2,
            '/' => accumulator /= 2,
            _ => { /* ignore everything else */ }
        }
    }

    println!("The program \"{}\" calculates the value {}",
             program, accumulator);
}
```

Remarques

- `println!` est une macro !
- `match` proche du `cond` du LISP...
- sans `mut` la variable *accumulator* serait une constante !

Remarques

- `println!` est une macro !
- `match` proche du `cond` du LISP...
- sans `mut` la variable *accumulator* serait une constante !

Évidemment, nous obtenons :

```
The program "+ + * - /" calculates the value 1
```

Plan de rouille

Les structures

Permettent la programmation orientée objet.

```
// Une structure unitaire
struct Nil;

// Un n-uplet
struct Pair(i32, f64);

// Une structure avec deux champs
struct Point {
    x: f64,
    y: f64,
}
```

Les méthodes

```
struct Point {
    x: f64,
    y: f64,
}

impl Point {
    fn origin() -> Point {
        Point { x: 0.0, y: 0.0 }
    }

    fn new(x: f64, y: f64) -> Point {
        Point { x: x, y: y }
    }

    fn print(&self) {
        println!("Point ({},{})",self.x,self.y);
    }
}
```

Les énumérations

Des ensembles d'instances...

```
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}
```

Les énumérations

Des ensembles d'instances...

```
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}
```

Remarque : possibilité de mélanger les types !

Les énumérations (2)

```
let x: Message = Message::Move { x: 3, y: 4 };

enum BoardGameTurn {
    Move { squares: i32 },
    Pass,
}

let y: BoardGameTurn = BoardGameTurn::Move { squares: 1 };
```

Les traits

Les traits proposent :

- des interfaces ;
- un mécanisme proche de l'héritage (voir prochaine section) ;
- des méthodes par défaut ;
- ...

Les traits : exemple (1)

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

Les traits : exemple (2)

```
struct Circle {
    radius: f64,
}

trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

fn main () {
    let c = Circle { radius: 2.01 };
    println!("L'aire est du cercle est {}", c.area());
}
```

Méthodes par défaut

```
trait HasArea {
    fn area(&self) -> f64 {
        println!("Ne fait rien");
        return 0.0;
    }
}
```

Méthodes par défaut

```
trait HasArea {
    fn area(&self) -> f64 {
        println!("Ne fait rien");
        return 0.0;
    }
}
```

Spécialisation possible ensuite via **impl**.

Principes

- indépendant du typage ;
- méthodes génériques ;
- combinaison possible avec les **traits**

Exemple simple

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

let x: Option<i32> = Some(5);
let y: Option<f64> = Some(5.0f64);
```

Combinaisons avec les structures

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

let int_origin = Point { x: 0, y: 0 };
let float_origin = Point { x: 0.0, y: 0.0 };
```

Combinaisons avec les structures

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

let int_origin = Point { x: 0, y: 0 };
let float_origin = Point { x: 0.0, y: 0.0 };
```

Remarque : inférence automatique des types.

Combinaisons avec les structures (2)

```
impl<T> Point<T> {
    fn swap(&mut self) {
        std::mem::swap(&mut self.x, &mut self.y);
    }
}
```

Remarque : nous verrons le &mut dans la section suivante.

Combinaisons avec les structures (3)

```
trait HasArea {
    fn area(&self) -> f64;
}

struct Circle {
    radius: f64,
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

struct Square {
    x: f64,
    y: f64,
    side: f64,
}

impl HasArea for Square {
    fn area(&self) -> f64 {
        self.side * self.side
    }
}

fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

Combinaisons avec les structures (4)

```
struct Rectangle<T> {
    x: T,
    y: T,
    width: T,
    height: T,
}

impl<T: PartialEq> Rectangle<T> {
    fn is_square(&self) -> bool {
        self.width == self.height
    }
}

fn main() {
    let mut r = Rectangle {
        x: 0,
        y: 0,
        width: 47,
        height: 47,
    };

    assert!(r.is_square());

    r.height = 42;
    assert!(!r.is_square());
}
```

Combinaisons avec les structures (4)

```
struct Rectangle<T> {
    x: T,
    y: T,
    width: T,
    height: T,
}

impl<T: PartialEq> Rectangle<T> {
    fn is_square(&self) -> bool {
        self.width == self.height
    }
}

fn main() {
    let mut r = Rectangle {
        x: 0,
        y: 0,
        width: 47,
        height: 47,
    };

    assert!(r.is_square());

    r.height = 42;
    assert!(!r.is_square());
}
```

Remarque : *PartialEq* est un trait de base pour la comparaison.

Suite et fin

- Héritage entre traits,

```
trait Foo {
    fn foo(&self);
}
trait FooBar : Foo {
    fn foobar(&self);
}
```

- Méthode par défaut (sinon le corps est obligatoire lors de l'implémentation).

```
trait Test {
    fn test(&self) {
        println!("Test");
    }
}
```

- Mot-clé *where* pour les « generics » multiple.

```
fn bar<T, K>(x: T, y: K) where T: Clone, K: Clone
```

- Contraintes multiples.

```
fn foobar<T, K>(x: T where T: Clone + Debug
```

Plan de rouille

Principes

Dans RUST les variables sont :

- non-mutable par défaut ;
- change de propriétaire lors d'un passage en paramètre ;
- l'allocation de mémoire est gérée avec un ramasse miette ;
- les variables sont désallouées dès leur fin de portée.

Variable mutable

```
let x = 5;
x = 6; // error !
```

Solution : utiliser le mot-clé *mut*

```
let mut x = 5;
x = 6;
```

Passage en argument

Principe : le passage en argument transmet aussi la propriété

```
fn take(v: Vec<i32>) {
    // plein de belles choses
}

fn main () {
    let v = vec![1, 2, 3];
    take(v);
    println!("v[0] est : {}", v[0]);
}
```

Résultat :

```
test4.rs:8:29: 8:30 error: use of moved value: ‘v‘ [E0382]
test4.rs:8     println!("v[0] est : {}", v[0]);
```

Passage en argument (2)

```
let v = vec![1, 2, 3];

let v2 = v;

println!("v[0] est : {}", v[0]);
```

Résultat :

```
error: use of moved value: ‘v‘
println!("v[0] is : {}", v[0]);
```

Emprunter une variable

⇒ Utilisation de &

```
fn foo(v1: &Vec<i32>, v2: &Vec<i32>) -> i32 {
    // Faire plein de chose

    // Retourne la solution
    41
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let answer = foo(&v1, &v2);

// on continue avec v1 et v2
```

Emprunter une variable mutable

⇒ Utilisation de `&mut`

```
let mut x = 5;
{
    let y = &mut x;
    *y += 1;
} // Fin de l'emprunt
println!("{}", x);
```

Emprunter une variable mutable (2)

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

let mut a = Point { x: 5, y: 6 };

a.x = 10;

let b = Point { x: 5, y: 6};

b.x = 10; // error : cannot assign to immutable field ‘b.x‘
```

Oublis volontaires

- La gestion fine de la mémoire.
- *static* et *const*.
- Les modules avec *crates*.
- Les attributs.
- Allocations sur le tas et la pile.
- La gestion des erreurs.
- ...

Pour en savoir plus

- Le site officiel `https://www.rust-lang.org/`
- Le *Livre* `https://doc.rust-lang.org/book/`
- Par les exemples `http://rustbyexample.com/`
- Pierre de Rosette
  `https://github.com/Hoverbear/rust-rosetta`

The king is gone but he's not forgotten
Is this the story of johnny rotten ?
It's better to burn out 'cause rust never sleeps
The king is gone but he's not forgotten.

Hey hey, my my
Rock and roll can never die
There's more to the picture
Than meets the eye.

The king is gone but he's not forgotten
Is this the story of johnny rotten ?
It's better to burn out 'cause rust never sleeps
The king is gone but he's not forgotten.

Hey hey, my my
Rock and roll can never die
There's more to the picture
Than meets the eye.

Hey Hey, My My (Into the Black), Neil Young

Si les fusils s'inventent des guerres
Et si les feuilles attendent le printemps,
Ne luttons pas, comme eux, contre le temps.
Contre la rouille, il n'y a rien à faire.

Si les fusils s'inventent des guerres
Et si les feuilles attendent le printemps,
Ne luttons pas, comme eux, contre le temps.
Contre la rouille, il n'y a rien à faire.

La rouille, Maxime le Forestier

Merci de votre attention.

Questions ?