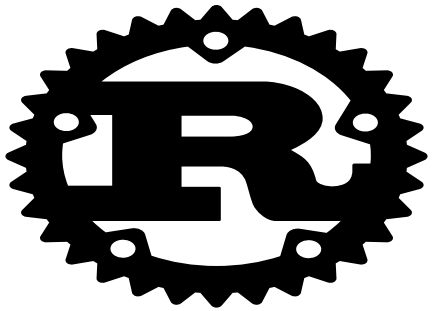
Rustc : meilleur ami, meilleur ennemi

Julien Rolland, Hugo Pompougnac, Vincent Bonnevalle

Université Paris-Didierot



- zero-cost abstraction
- move semantics
- guaranted memory safety
- program without data races
- trait-based generics
- pattern matching
- type inference
- minimal runtime
- efficient C bindings

Plan

- Typage fort
- Inférence de type
- Transtypage et conversion

- Typage fort :

```
let x:i64 = 5;
let y:u64 = x;

> rustc src/main.rs
# ...
--> src/main.rs:3:17
  |
3 |     let y:u64 = x;
  |                 ^ expected u64, found i64

error: aborting due to previous error
# ...
```

- Inférence de type :

```
struct Foo {
    bar: i64,
}
fn f(a: Vec<Foo>) -> Foo {

}
fn g(a: Vec<Foo>) {

}
fn main() {
    let v = Vec::new();
    g(v);
}
```

- Transtypage et conversions :
  - Conversions d'entier (sans perte) :

```
let x:u32 = 1 << 25;
let y:u64 = x as u64
```

  - `From` et `Into` :

```
struct Number {value: i32,}
impl From<i32> for Number {
    fn from(item: i32) -> Self {
        Number { value: item }
    }
}
// ...
let x:i32 = 64;
let n = Number::from(x);
let y:i32 = n.into();
```

trait ≈ interfaces de java

```
pub trait From<T> {
    fn from(T) -> Self;
}
#[derive(Copy, Clone)]
struct Number {value: i32,}
impl From<i32> for Number {
    fn from(item: i32) -> Self {
        Number { value: item }
    }
}
```

Paramètre générique de `Rust` ≈ `templates` de `C++` (mais en mieux).
Nom du type générique en `CamelCase`.

```
fn foo<T>(arg: T) {
    ...
}
struct Bar<T>(T) {
    foo: T,
}
impl<T> Bar<T> {
    fn swap(&mut self) {
        // ...
    }
}
```

Fusion des paramètres génériques et des traits

```
fn foo<T: From<i32>>(arg: T) { ... }
```

ou

```
fn foo<T>(arg: T)
  where T: From<i32> { ... }
// ...
struct Char { c:i32, }
let n = Number::from(64);
foo(n);                    // OK
foo(Char { c:64 }); // KO
```

```
f(c);
```

```
f(c);
```

move :

```
fn f(c: T) { ... }
```

```
f(c);
```

move :

```
let t = T {};
f(t);
println!("t: {:?}", t);
```

```
error[E0382]: borrow of moved value: `t`
  --> src/main.rs:11:25
   |
9  |     let t = T {};
   |         - move occurs because `t` has type `T`,
   |             which does not implement the `Copy` trait
10 |     f(t);
   |       - value moved here
11 |     println!("t: {:?}", t);
   |                         ^ value borrowed here after move
```

```
f(c);
```

copie :

```
#[derive(Copy)]
struct T { ... }
fn f(c: T) { ... }
```

```
f(c);
```

borrow (ou référence) :

```
fn f(c: &T) { ... }
```

```
class T {
  vector<int> vec;
  void do_something();
  void f() {
    for (auto& e : vec) {
      do_something();
    }
  }
};
```

```
class T {
  vector<int> vec;
  void do_something() { vec.push_back(3); }
  void f() {
    for (auto& e : vec) {
      do_something();
    }
  }
};

> g++ main.cpp
```

```
struct T {
    vec: Vec<i64>,
}

impl T {
    fn do_something(&mut self) { self.vec.push(3) }
    fn f(&mut self) {
        for e in &self.vec {
            self.do_something();
        }
    }
}
```

```
error[E0502]: cannot borrow `*self` as mutable
because it is also borrowed as immutable
--> src/main.rs:11:13
10 |  for e in &self.vec {
   |           ----------
   |           |
   |           immutable borrow occurs here
   |           immutable borrow used here,
   |           in later iteration of loop
11 |      self.do_something();
   |      ^^^^^^^^^^^^^^^^^^^ mutable borrow
   |                           occurs here
error: aborting due to previous error
```

Règles sur les références (ou *borrowing*) :

- une référence ne peut pas vivre plus longtemps que l'objet original (pas de *dangling reference*)
- on peut avoir :
  - soit une ou plusieurs références non mutable sur un objet (`&T`)
  - soit une (et seulement une) référence mutable sur un objet (`&mut T`)
  - mais pas les deux en même temps

$\Rightarrow$ pas de `dangling reference` et pas de modification concurrent d'un objet

Quel est le problème avec le parallélisme ?

Dans la plupart des langages impératifs (et singulièrement en C) :

- Il est très simple d'engendrer un bug avec l'écriture concurrente d'un emplacement mémoire insuffisamment protégé.
- Il est très compliqué de retrouver le morceau de code qui cause le bug, surtout si le programme ne crashe pas mais corrompt simplement les données.
- C'est d'autant plus vrai que la survenue ou non du bug dépend de chaque exécution.

Rust à la rescousse !

Rust propose donc de vérifier à la compilation qu'un de ces bugs ne peut pas être introduit :

- Pour que le code compile, il faut qu'il soit thread-safe (et le compilateur indique à quelle ligne il ne l'est pas).
- Une série de mécanismes, dont la plupart reposent sur le borrow-checker de Rust, sont fournis pour apporter ces garanties statiques.
- Ils peuvent être désactivés en insérant le code problématique dans un bloc unsafe, mais dans ce cas-là c'est le programmeur lui-même qui choisit de bugger son code.

Dans le monde du C

Considérons le code C suivant, consistant en la pire manière possible de faire du parallélisme :

```
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 100

int share_var = 0 ;

void * thread_affect(void* arg) {
  share_var = 0 ;
}

void * thread_div(void* arg) {
  if (share_var != 0) {
    fprintf(stderr, "%s", "Suis-je en train de diviser par zero ?\n");
    share_var = 800/share_var + 12 ;
  }
  else {
    share_var = 8 ;
  }
}
```

```
int main() {
  pthread_t threads[NUM_THREADS] ;

  for (int i = 0 ; i < NUM_THREADS ; i++) {
    if (i % 2 == 0) {
    pthread_create(&threads[i], NULL, thread_affect, NULL) ;
  }
  else {
    pthread_create(&threads[i], NULL, thread_div, NULL) ;
  }
}

for (int i = 0 ; i < NUM_THREADS ; i++) {
  pthread_join(threads[i], NULL) ;
}

printf("%d\n",share_var) ;
}
```

La compilation et l'exécution répétée de ce code produit les sorties suivantes :

```
> gcc unsafe_concurrency.c -lpthread
> ./a.out
> ./a.out
Suis-je en train de diviser par zero ?
8
> ./a.out
Suis-je en train de diviser par zero ?
Suis-je en train de diviser par zero ?
Exception en point flottant
> ./a.out
Suis-je en train de diviser par zero ?
Exception en point flottant
```

La raison est simple : une fois qu'on a vérifié que `share_var` est différent de 0, on n'a aucune garantie que c'est toujours le cas deux lignes plus loin.

Dans le monde de Rust

```
const NTHREADS: i32 = 100;
static mut share_var: i32 = 0 ;
fn main() {
    let mut threads = vec![];
    for i in 0..NTHREADS {
        if i % 2 == 0 {
            threads.push(thread::spawn(|| share_var = 0));
        }
        else {
            threads.push(thread::spawn(|| {
                if share_var != 0 {
                    eprintln!("Suis-je en train de diviser par zero ?\n");
                    share_var = 800/share_var + 12 ;
                }
                else {
                 share_var = 8 ;
                }
            }));
        }
    }
    for child in threads { child.join(); }
    println!("{}",share_var);
}
```

Il ne passe même pas la compilation :

```
> rustc unsafe_concurrency.rs
error[E0133]: use of mutable static is unsafe
  and requires unsafe function or block
--> unsafe_concurrency.rs:11:48
error[E0133]: use of mutable static is unsafe
  and requires unsafe function or block
...
```

Nouvel essai

Il semble y avoir un problème avec les variables globales (`static`). Mais que se passe-t-il si nous essayons de "piéger" le compilateur en déclarant `share_var` localement et en cherchant à la modifier via la clôture d'un thread (avec un code moins volumineux) ?

```
use std::thread;

const NTHREADS: i32 = 100;

fn main() {
    let mut share_var: i32 = 0 ;
    thread::spawn(|| share_var = 0);
}
```

Voyons ce que nous dit le compilateur :

```
> rustc unsafe_concurrency2.rs
error[E0373]: closure may outlive
the current function,
but it borrows share_var,
which is owned by the current function
```

Pour passer la compilation, on peut forcer la propriété de `share_var` par le mot-clé `move` :

```
thread::spawn(move || share_var = 0);
```

Mais ce mot clé ne fait qu'un passage par valeur : on ne modifie plus qu'une variable locale au *thread*, et non la variable partagée entre les *threads* ...

Le premier commandement de Rust : "thread isolation"

Jusqu'ici, on constate donc que Rust interdit l'écriture concurrente de données de deux manières :

- En interdisant l'écriture de variables globales.
- En s'assurant, via le *borrow checker*, qu'une variable accessible en écriture ne soit jamais possédée par plus d'une fonction (et donc d'un *thread*).

Les *threads* étant statiquement isolés les uns des autres, il est impossible que l'un d'entre eux corrompe les données utilisées par un autre. Mais à ce compte-là, comment peuvent-ils partager des données ? En les protégeant !

Passer les fourches caudines du compilateur

Spontanément, pour protéger des données partagées, on les verrouille avec un Mutex garantissant l'absence d'accès concurrents :

```
use std::sync::{Arc, Mutex};
use std::thread;
const nthreads: i32 = 100;
fn main() {
    let share_var = Arc::new(Mutex::new(0)) ;
    let mut threads = vec![];
    // Traitement des threads
    for child in threads { child.join(); }
    println!("{}",*share_var.lock().unwrap());
}
```

```
for i in 0..nthreads {
    let share_var = share_var.clone() ;
    if i % 2 == 0 {
        threads.push(thread::spawn(move || {
            let mut data = share_var.lock().unwrap() ;
            *data = 0 ;
        })) ;
    }
    else {
        threads.push(thread::spawn(move || {
            let mut data = share_var.lock().unwrap() ;
            if *data != 0 {
                eprint!("suis-je en train de diviser par zero ?\n");
                *data = 800/(*data) + 12 ;
            }
            else {
             *data = 8 ;
            }
        }));
    }
}
```

"Lock data, no code"

Que se passe-t-il ici ?

- Pour prendre la propriété sur la variable partagée, il faut demander un *lock*.
- Le *lock* est relâché quand on sort de la portée où il est demandé.
- On ne peut donc pas accéder aux données en même temps qu'un autre *thread*.
- Le *mutex* est transmis par valeur aux *threads*, mais peu importe, puisque le verrou sur lequel il pointe est unique (c'est un pointeur de type *Arc*, nous y reviendrons).

Il est donc absolument impossible d'accéder à la variable dans une section du code qui n'est pas protégée. En C, au contraire, on peut écrire :

```
void * thread(void* arg) {
    pthread_mutex_lock(mut);
    share_var = 0 ;
    pthread_mutex_unlock(mut);
    /* ... */
    share_var = 5 ;
    /* ... */
}
```

Le C protège ("dynamiquement") la portion de code encadrée d'un mutex : Rust protège (statiquement) la variable partagée quelle que soit sa position dans le code.

Robustesse et typage en Rust

Jusqu'ici, nous transférons des données d'un thread à l'autre sans les corrompre. Comme dans tous les langages, ce n'est pas possible en toute généralité. Certains types de pointeurs, par exemple, ne garantissent pas l'atomicité de leur déréférencement. C'est le cas des pointeurs `Rc` (contrairement aux pointeurs `Arc` que nous utilisons dans le cas des `Mutex`) :

```
const NTHREADS: i32 = 100;
fn main() {
    let share_var = Rc::new(Mutex::new(0)) ;
    thread::spawn(move || {
        let mut data = share_var.lock().unwrap() ;
        *data = 0 ;
    }) ;
}
```

Donne l'erreur de compilation suivante :

```
> rustc unatomic.rs
error[E0277]: `std::rc::Rc<std::sync::Mutex<i32>>`
    cannot be sent between threads safely
--> unatomic.rs:9:5
```

Thread safety isn't just documentation ; it's law

Ainsi :

- `Rust` permet d'utiliser librement les différents types de données sans avoir besoin de se documenter sur chaque type pour identifier un accès concurrent (en l'occurence à un pointeur).
- On dit des types *thread-safe* qu'il sont `Send`, et des autres qu'ils sont `!Send`.
- À l'inverse, en `C++` par exemple, l'usage d'un *smart-pointer* (spécifié pour être *thread-safe*) ou d'un pointeur "nu" (dont la spécification ne garantit rien) est indifférent du point de vue des erreurs de compilation.

Pourquoi la communication par message ?

Il est courant de synchroniser les données de plusieurs *threads* en passant par des *mutex* : mais n'est-il pas plus simple et plus intuitif de leur permettre de s'envoyer des messages ?
En effet, on se représente l'interdiction des écritures concurrentes de manière moins artificielle :

- S'il faut avoir reçu une donnée pour la manipuler.
- S'il faut envoyer une donnée pour qu'un autre puisse la manipuler.
- Si on ne peut plus modifier une donnée qu'on a envoyée.

Avec un modèle de ce type, notre variable fonctionne comme une feuille de papier : seul celui qui l'a dans les mains peut agir dessus...
Et à nouveau, on devine que le borrow-checker va permettre de rendre ce mécanisme très robuste. C'est de cette manière que fonctionnent les *channels*.

Utiliser les channels : thread isolation++

Un *channel* consiste en une paire (un récepteur et un émetteur) que l'on construit en faisant appel à la fonction : Ce `Sender` et ce `Receiver`, partagés, permettent à plusieurs *threads* de communiquer entre eux.

```
pub fn send(&self, t:T) -> Result<(), SendError<T>>
```

Et on reçoit une variable en utilisant la méthode suivante du `Receiver`, qui lui en reprend la propriété :

```
pub fn recv(&self) -> Result<T, RecvError>
```

La réception est évidemment bloquante.
Ainsi, la question de la protection de la mémoire partagée ne se pose plus (ou très différemment) : en effet, une telle variable n'existe tout simplement pas du point de vue d'un *thread* tant qu'elle ne lui a pas été transférée via un *channel*.

Un exemple : la factorielle

```
use std::thread;
use std::sync::mpsc::channel;

const N: i32 = 10;

fn main() {
    let (tx, rx) = channel();
    for i in 0..N {
        let tx = tx.clone();
        thread::spawn(move|| {
            let mut n = i;
            n = n + 1;
            tx.send(n).unwrap();
            n = n * 100;
        });
    }

    let mut res = 1;
    for _ in 0..N {
        res = res * rx.recv().unwrap();
    }
    println!("{}",res);
}
```

Quelques remarques :

- Cet exemple est artificiel : le calcul pour obtenir n est trivial. Imaginons qu'on l'obtient au prix d'opérations coûteuses, comme la lecture dans un fichier.
- La modification de n après l'envoi n'a aucun impact sur la valeur envoyée (partagée).
- Le thread principal ne finit pas tant qu'il n'a pas reçu ses 10 entiers, et il ne les reçoit pas tant que les threads fils ne les lui ont pas envoyé.
- Les channels standard de Rust permettent plusieurs écrivains, mais un seul lecteur... Notre exemple du début n'est donc pas si facile à écrire de cette manière.

Conclusion

- Ce tour d'horizon est évidemment minimal, et d'autres mécanismes existent : par exemple, la possibilité pour un *thread* de partager sa pile avec ses enfants sans courir de risque.
- Néanmoins, il permet de toucher du doigt l'un des cœurs de la conception de `Rust` : le compilateur, en tirant profit du *borrow-checking*, interdit statiquement tout accès concurrent impliquant une écriture.
- Les principes présentés ici sont issus de https ://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html ; le détail des différents types de données que nous manipulons sont tirés de la documentation officielle ; et les exemples sont écrits par nos soins.