Introduction à Rust

Programmation avancée

Jacques-Henri Jourdan

16 février 2024

Programmation bas niveau

Certains logiciels requièrent un haut niveau de contrôle :

- sur la représentation en mémoire des valeurs ;
- sur l'emplacement des allocations et désallocations ;
- sur les calculs effectués, et à quel moment...

La réponse standard à ces besoins : C/C++.
Ces langages ont leur défauts :

- sémantique très complexe (à cause des pointeurs, notamment) ;
- aucune garantie de sûreté :
  - certains programmes ont un comportement non défini (*undefined behavior*)
  - (Parfois : *segmentation fault*, parfois pire : rien n'est visible.) ;
- mauvais support pour l'abstraction.

Rust

Un langage développé initialement par Mozilla pour réécrire des parties de Firefox.

Hautes performances, haut degré de contrôle avec sûreté et abstraction

Rust

Un langage développé initialement par Mozilla pour réécrire des parties de Firefox.

Hautes performances, haut degré de contrôle avec sûreté et abstraction

Abstraction gratuite (*Zero-cost abstraction*)

Mécanisme d'abstraction puissants sans impact sur la performance :

- Système de type sûr.
- Pointeurs : possession + emprunts avec durée de vie.
- Polymorphisme avec les *traits* ($\simeq$ type classes de Haskell).

Quand le système de types n'est pas assez expressif, on peut utiliser des fonctionnalités unsafe, en les cachant derrière des abstraction sûres.

Pourquoi est-ce intéressant d'étudier Rust

Langage moderne, empruntant beaucoup de concepts à d'autres :

- Polymorphisme, traits, clôtures, multithreading, RAII, abstractions gratuites...

Étudier Rust, c'est étudier ces concepts, et apprendre ces autres langages aussi !

Aujourd'hui : quelques idées-clefs du langage.

Ressources

Références pour apprendre Rust :

- Beaucoup de références ici : https://www.rust-lang.org/learn
- Les exercices que nous vous demandons de faire au fur et à mesure pour apprendre le langage par la pratique : Rustlings
- D'autres tutoriaux que j'aime bien :
  - Rust 101 par Ralf Jung : tutoriel accessible pour les bases du langage ;
    - Certains exemples de ce cours viennent de ce tutoriel.
  - Rust by example : beaucoup d'exemples de Rust idiomatique ;
  - The Rust Book : tutoriel "officiel" pour Rust et une partie de son écosystème. Très complet, mais long et un peu aride.

Le minimum d'une séquence d'entiers

```
enum NumberOrNothing {
    Number(i32), Nothing
}


fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  let mut res = NumberOrNothing::Nothing;
  for el in vec {
    match res {
      NumberOrNothing::Nothing => {
        res = NumberOrNothing::Number(el);
      },
      NumberOrNothing::Number(n) => {
        let m = if n < el { n } else { el };
        res = NumberOrNothing::Number(m);
      }
    }
  }

  return res
}
```

Un fonction pour calculer le maximum d'une séquence d'entiers de 32 bits (signés).

- Les types des paramètres et de retour doivent être annotés (comme en C).
- Les types des variables locales sont inférés (le plus souvent).

Le minimum d'une séquence d'entiers

```
enum NumberOrNothing {
    Number(i32), Nothing
}
```

```
fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  let mut res = NumberOrNothing::Nothing;
  for el in vec {
    match res {
      NumberOrNothing::Nothing => {
        res = NumberOrNothing::Number(el);
      },
      NumberOrNothing::Number(n) => {
        let m = if n < el { n } else { el };
        res = NumberOrNothing::Number(m);
      }
    }
  }

  return res
}
```

Séquence vide
⇒ le minimum n'est pas défini

On utilise un type algébrique :

- entier de 32 bits, ou rien,
- filtrage de motif,
- constructeurs,

Le minimum d'une séquence d'entiers

```
enum NumberOrNothing {
    Number(i32), Nothing
}
use NumberOrNothing::*;

fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  let mut res =                    Nothing;
  for el in vec {
    match res {
                        Nothing => {
        res =                      Number(el);
      },
                        Number(n) => {
        let m = if n < el { n } else { el };
        res =                      Number(m);
      }
    }
  }

  return res
}
```

Séquence vide
⇒ le minimum n'est pas défini

On utilise un type algébrique :

- entier de 32 bits, ou rien,
- filtrage de motif,
- constructeurs,
  - (espace de noms spécifique, peut être importé).

Représentation mémoire

En OCaml (mais aussi Java, Python, Haskell, ...), les valeurs complexes sont boxées.

- La valeur est alors un pointeur vers le tas, qui contient l'information.
- Cette indirection (et son coût) est obligatoire.

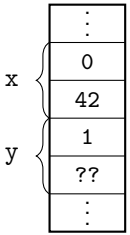
En Rust (comme en C/C++), les types structurés sont représentés sans indirection.

- Les variables locales sont stockés dans la pile.
- Les types ont une taille, correspondant à l'espace utilisé par une telle variable.
- Les affectations et le passage de paramètre nécessitent une copie de plusieurs octets.
- Les indirections sont contrôlées par l'utilisateur.

Représentation mémoire, exemple

```
enum NumberOrNothing {
    Number(i32), Nothing
}
use NumberOrNothing::*;

fn main() {
   let x = Number(42);
   let y = Nothing;
}
```



Directement sur la pile !

Fonctions associées

```
impl NumberOrNothing {
  fn from_i16(x: i16) -> Self {
    Number(x as i32)
  }

  fn print(self) {
    match self {
      Nothing =>
        println!("The number is: <nothing>"),
      Number(n) =>
        println!("The number is: {}", n),
    };
  }
}

fn main() {
  let fortytwo = NumberOrNothing::from_i16(42);
  NumberOrNothing::print(fortytwo);

  let min = vec_min(vec![18,5,7,2,9,27]);
  min.print()
}
```

On peut définir des fonctions associées aux types.

- Accessibles à travers l'espace de noms du type.
- `Self` : nom court pour le type en question.

Fonctions associées

```
impl NumberOrNothing {
  fn from_i16(x: i16) -> Self {
    Number(x as i32)
  }

  fn print(self) {
    match self {
      Nothing =>
        println!("The number is: <nothing>"),
      Number(n) =>
        println!("The number is: {}", n),
    };
  }
}

fn main() {
  let fortytwo = NumberOrNothing::from_i16(42);
  NumberOrNothing::print(fortytwo);

  let min = vec_min(vec![18,5,7,2,9,27]);
  min.print()
}
```

Certaines sont des méthodes.

- Le premier paramètre, `self` a implicitement pour type `Self`.
- Deux syntaxes pour l'appel :
  - Via le nom du type.
  - Avec un point "." : premier paramètre avant le point.
- (Comme dans les langages OO.)

Constantes

On a aussi des constantes globales :

```
const FORTYTWO : i32 = 42;
```

Et des constantes associées :

```
impl NumberOrNothing {
  const FORTYTWO : NumberOrNothing = 42;
}
```

On peut y accéder en lecture, mais pas en écriture :

```
fn f() {
  let x : i32 = FORTYTWO;
  let y : NumberOrNothing = NumberOrNothing::FORTYTWO;
}
```

Enregistrements, tuples, tableaux

On a (bien sûr !) d'autres manières de construire des types :

- Tuples `(T1, T2, ...)` :
  - construction : `(e1, e2, ...)`
  - projections : `t.0`, `t.1`, ...
- Enregistrements déclarés par `struct R { f1: T1, f2: T2, ... }`
  - construction : `R { f1: e1, f2: e2, ... }`
  - projections : `r.f1`, `r.f2`, ...
- Tuples nommés : déclarés par `struct NT(T1, T2, ...)`
  - construction : `NT(e1, e2, ...)`
  - projection : `nt.0`, `nt.1`, ...
- Tableaux de longueur fixe : `[T; 42]`
  - construction : `[e; 42]`
  - accès : `a[i]`

Enregistrements, tuples, tableaux

On a (bien sûr !) d'autres manières de construire des types :

- Tuples `(T1, T2, ...)` :
  - construction : `(e1, e2, ...)`
  - projections : `t.0`, `t.1`, ...
- Enregistrements déclarés par `struct R { f1: T1, f2: T2, ... }`
  - construction : `R { f1: e1, f2: e2, ... }`
  - projections : `r.f1`, `r.f2`, ...
- Tuples nommés : déclarés par `struct NT(T1, T2, ...)`
  - construction : `NT(e1, e2, ...)`
  - projection : `nt.0`, `nt.1`, ...
- Tableaux de longueur fixe : `[T; 42]`
  - construction : `[e; 42]`
  - accès : `a[i]`

Tous ces types sont stockés directement, sans indirection implicite.

- Les valeurs des tuples, enregistrement et tableaux sont la concaténation de leurs composantes.

Alias : contre-exemple en C++

```
void foo(std::vector<int> v) {
  int *first = &v[0];
  v.push_back(42);
  *first = 1337;
}
```

Où est le bug ?

Alias : contre-exemple en C++

```cpp
void foo(std::vector<int> v) {
  int *first = &v[0];
  v.push_back(42);
  *first = 1337;
}
```

- `push_back` peut réallouer le vecteur,
  - invalidant les pointeurs internes tels que `first`.
- Bug un peu pervers :
  - la plupart du temps, `push_back` ne réalloue pas le vecteur ;
  - très souvent, `first` pointe sur la bonne valide (mais est invalide).
- Source du bug :
  - Nous avons "muté" la mémoire avec deux pointeurs aliasés, `first` et `v`.

Invalidation d'itérateur

En C++ :

```
void duplicate_vec(std::vector<int> &v) {
  for(int i: v) {
    v.push_back(i);
  }
}
```

Où est le bug ?

Invalidation d'itérateur

En C++ :

```
void duplicate_vec(std::vector<int> &v) {
  for(int i: v) {
    v.push_back(i);
  }
}
```

- `push_back` peut réallouer le vecteur ;
- boucle `for` : itérateur implicite, contient un pointeur dans le vecteur ;
- ce pointeur est invalidé par la réallocation ;
- invalidation d'itérateur.

En C++, invalidation d'itérateur : règles subtiles.

- Parfois (e.g., tables de hachage), muter n'invalide pas les itérateurs...

En java, détection dynamique. Limite la portée du bug, mais ne l'élimine pas.

Encore une fois : mutation à travers un pointeur aliasé.

Double free/Use-after-free

En C :

```c
void print_ints(int *tab, int len) {
  for(int i = 0; i < len; i++)
    printf("%d\n", tab[i]);
  free(tab);
}

void foo() {
     int *t = (int*)malloc(42 * sizeof(int));
     for(int i = 0; i < 42; i++) t[i] = i;
     print_ints(t, 42);
     free(t);
}
```

Où est le bug ?

Double free/Use-after-free

En C :

```c
void print_ints(int *tab, int len) {
  for(int i = 0; i < len; i++)
    printf("%d\n", tab[i]);
  free(tab);
}

void foo() {
     int *t = (int*)malloc(42 * sizeof(int));
     for(int i = 0; i < 42; i++) t[i] = i;
     print_ints(t, 42);
     free(t);
}
```

- Le tableau est désalloué deux fois.
- On met le gestionnaire de mémoire dans un état incohérent.

On a pas respecté de protocole de possession pour la fonction `print_ints`.

Encore une fois : mutation (`free`) à travers un pointeur aliasé.

Data race

On anticipe un peu sur le cours de concurrence...

```
int main() {
    int x = 0;
    std::thread t([&](){x += 1;});
    x += 1;
    t.join();
}
```

Où est le bug ?

Data race

On anticipe un peu sur le cours de concurrence...

```
int main() {
    int x = 0;
    std::thread t([&](){x += 1;});
    x += 1;
    t.join();
}
```

- La variable `x` est modifiée dans deux *threads* différents, simultanément.
- En C++ : toujours un comportement indéfini.

Encore une fois : mutation à travers un pointeur aliasé (dans la clôture, et directement).

La règle générale des alias

En Rust, on maintient l'invariant : mutation XOR alias.

Si on veut muter la mémoire, on doit avoir l'unique alias actif.

- On voit la mémoire comme une ressource qu'on ne peut pas dupliquer.
- On est le propriétaire de l'emplacement mémoire.

Cela s'applique aussi à la libération de la mémoire.

- Il n'a a pas de ramasse-miette (comme en Caml, Java, Python) en Rust.
  - (Meilleur contrôle des ressources, meilleures performances.)

On ne peut donc pas toujours copier librement les valeurs.

- On dit que Rust a un système de types sous-structurel.

Possession dans vec_min

```
impl NumberOrNothing {
  fn print(self) { ... }
}

fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  ...
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(v).print();


}
```

Possession dans vec_min

```
impl NumberOrNothing {
  fn print(self) { ... }
}

fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  ...
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(v).print();
  v.push(42);
  vec_min(v).print();
}
```

Possession dans vec_min

```
impl NumberOrNothing {
  fn print(self) { ... }
}

fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  ...
}

fn main() {
  let mut v = ve
  vec_min(v).pri
  v.push(42);
  vec_min(v).pri
}
```

```
error[E0382]: borrow of moved value: ‘v‘
  --> src/main.rs:37:3
   |
35 |   let mut v = vec![18,5,7,2,9,27];
   |       ...
36 |   vec_min(v).print();
   |              - value moved here
37 |   v.push(42);
   |   ^^^^^^^^^^ value borrowed here after move
```

Possession dans vec_min

```
impl NumberOrNothing {
  fn print(self) { ... }
}

fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  ...
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(v).print();
  v.push(42);
  vec_min(v).print();
}
```

Passer `v` en paramètre de `vec_min` a déclenché un transfert.

- On a perdu la possession du vecteur. On ne peut plus l'utiliser.
- Heureusement, parce que `vec_min` a désalloué le vecteur automatiquement.
  - RAII : le destructeur (i.e., la désallocation) est exécuté dès qu'on sort de la portée d'une variable (sauf si la variable a déjà été vidée).

Représentation en mémoire de `Vec`

```
fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  ...
}
```

Rappel : le passage en paramètre copie la séquence d'octets d'une valeur.
Est-ce à dire que le contenu d'un vecteur est copié quand on appelle `vec_min` ?

Représentation en mémoire de Vec

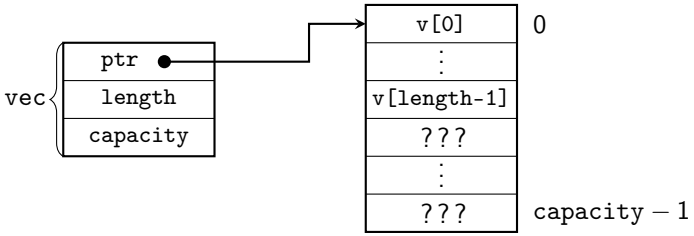
```
fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  ...
}
```

Rappel : le passage en paramètre copie la séquence d'octets d'une valeur.
Est-ce à dire que le contenu d'un vecteur est copié quand on appelle `vec_min` ?

Non !

- Ce serait inefficace si le tableau est grand.
- Ce serait difficile à compiler, car on ne peut pas savoir statiquement le nombre d'éléments à copier.

Représentation en mémoire de Vec



Opérationnellement, le transfert d'un `Vec` ne copie que les champs `ptr`, `length` et `capacity`.

Sémantiquement, la possession de tout le contenu est transférée.
Pas d'alias chez l'appelant.

Pour le programmeur, c'est comme si on avait transféré tout le vecteur, mais avec de bonnes performances.

L'essence de la possession : Box<T>

`Box<T>` est le type des pointeurs possédés pleinement vers `T`.

- Fonction de création :

```
impl<T> Box<T> {
    fn new(x: T) -> Box<T>
}
```

  - Transfert de `x` vers de la mémoire allouée dynamiquement (i.e., `malloc`).
  - Transfert de la possession du contenu de `x` vers la boîte.
  - Syntaxe d'appel : `Box::new(42)`.
- Peut être déréférencé (pour lecture/écriture) : `*p`.
- RAII : mémoire libérée automatiquement quand une variable sort de la portée.

Exemple d'utilisation : liste simplement chaînée :

```
type List = Option<Box<Node>>;
struct Node { elem: i32, next: List, }
```

Possession et forme du tas

Si on n'utilise que des types qui vérifient la possession, quelle est la forme du tas ?

Possession et forme du tas

Si on n'utilise que des types qui vérifient la possession, le tas est une forêt, enracinée aux variables locales (ou globales).

Il n'y a pas de partage, et pas de cycle !

Nous verrons dans un prochain cours comment contourner cette contrainte, lorsque c'est nécessaire.

Exceptions au suivi de la possession

La vérification de possession est-elle bonne pour toutes les variables ?

Exceptions au suivi de la possession

Non ! Certaines valeurs sont des données brutes, et peuvent être copiées à volonté.

- Exemples : `i32`, `(i64, u64)`, `bool`, ...

On dit que ces types implémentent le trait `Copy` (ou "sont `Copy`")

- `Copy` autorise le compilateur à ne pas vérifier la possession pour ces types.

Par défaut, les types déclarés avec `struct` ou `enum` ne sont pas `Copy`.

- Parce qu'ils pourraient être utilisés pour représenter des ressources inconnues du compilateur.
- Pour les rendre `Copy`, facile :

```
#[derive(Copy, Clone)]
enum NumberOrNothing {
    Number(i32), Nothing
}
```

- (Nous expliquerons mieux les traits dans le cours dédié.)

Emprunts partagés

Variante de `vec_min` qui évite le transfert de possession :

```
fn vec_min(vec: &Vec<i32>) -> NumberOrNothing {
  let mut res = Nothing;
  for el in vec {
    match res {
      Nothing => { res = Number(*el); },
      Number(n) => {
        let m = if n < *el { n } else { *el };
        res = Number(m);
      }
    }
  }
  return res
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(&v).print();
  v.push(42);
  vec_min(&v).print();
}
```

Plutôt que de passer la possession du vecteur, on en passe un emprunt partagé.

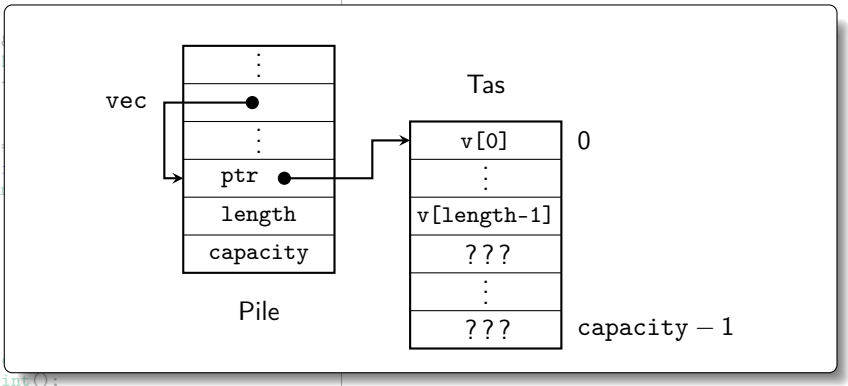
- À l'exécution : un pointeur sur `v`
- Ici, l'emprunt partagé n'est valide que pour la durée de la fonction.
- `&T` implémente `Copy`.
- `&T` n'autorise pas la mutation (en général... voir le cours sur la mutabilité intérieure).
- Synonymes : emprunt immuable, référence immuable.

Emprunts partagés

Variante de `vec_min` qui évite le transfert de possession :

```
fn vec_min(vec: &
  let mut res = 
  for el in vec 
    match res {
      Nothing =>
      Number(n) 
        let m = 
        res = Nu
      }
    }
  }
  return res
}

fn main() {
  let mut v = ve
  vec_min(&v).print();
  v.push(42);
  vec_min(&v).print();
}
```



Emprunts partagés

Variante de `vec_min` qui évite le transfert de possession :

```
fn vec_min(vec: &Vec<i32>) -> NumberOrNothing {
  let mut res = Nothing;
  for el in vec {
    match res {
      Nothing => { res = Number(*el); },
      Number(n) => {
        let m = if n < *el { n } else { *el };
        res = Number(m);
      }
    }
  }
  return res
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(&v).print();
  v.push(42);
  vec_min(&v).print();
}
```

Quel est le type de `el` ? Pourquoi ?

Emprunts partagés

Variante de `vec_min` qui évite le transfert de possession :

```
fn vec_min(vec: &Vec<i32>) -> NumberOrNothing {
  let mut res = Nothing;
  for el in vec {
    match res {
      Nothing => { res = Number(*el); },
      Number(n) => {
        let m = if n < *el { n } else { *el };
        res = Number(m);
      }
    }
  }
  return res
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(&v).print();
  v.push(42);
  vec_min(&v).print();
}
```

Lorsqu'on accède aux éléments d'un vecteur avec un emprunt partagé (e.g., en itérant sur le vecteur avec une boucle `for`), on récupère des emprunts partagés du contenu, parce que, en général, on ne peut pas extraire la possession en dehors du vecteur.

Nous devons donc déréférencer `el` !

Emprunts uniques

```rust
fn vec_inc(vec: &mut Vec<i32>) {
  for el in vec {
    *el += 1
  }
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_inc(&mut v);
  v.push(42); /* ... */
}
```

Les emprunts uniques sont similaires aux emprunts partagés.

Différences :

- Mutation autorisée.
- ...

Emprunts uniques

```rust
fn vec_inc(vec: &mut Vec<i32>) {
  for el in vec {
    *el += 1
  }
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_inc(&mut v);
  v.push(42); /* ... */
}
```

Les emprunts uniques sont similaires aux emprunts partagés.

Différences :

- Mutation autorisée.
- N'implémente pas `Copy`.
- Synonymes : emprunt mutable, référence mutable.

Emprunts et méthodes

```
impl NumberOrNothing {
  fn print( self) {
    match self {
      Nothing =>
        println!("The number is: <nothing>"),
      Number(n) =>
        println!("The number is: {}", n),
    };
  }
}
```

Appel :

- copie of `self`
- transfert complet de possession

Peut-on passer un emprunt pour `self` ?

Emprunts et méthodes

```
impl NumberOrNothing {
  fn print(&self) {
    match self {
      Nothing =>
        println!("The number is: <nothing>"),
      Number(n) =>
        println!("The number is: {}", n),
    };
  }
}
```

On peut demander à ce que `print` prenne un emprunt à `self`.

`&self` signifie que l'argument `self` a comme type `&Self`
(Syntaxe spécifique à `self`).

La syntaxe d'appel ne change pas :

```
let n = Nothing;
n.print();
```

La création de l'emprunt est automatique lors de l'appel.
(Pour l'argument `self` uniquement.)

Bien sûr, cela fonctionne aussi pour `&mut`.

Pointeurs internes aux structures de données

Imaginons que nous avons un vecteur d'objets qui ne sont pas `Copy`.
Disons `Vec<Vec<i32>>`.
Comment modifier l'un des `i32` stockés ?

Une possibilité :

```
fn set(i: i32, v: Vec<i32>, l: &mut Vec<Vec<i32>>)
```

Problème : impossible de modifier un `Vec<i32>` qui est déjà dans le vecteur !
Il faudrait extraire le `Vec<i32>`, puis le remettre en place... Pas très efficace.

Pointeurs internes aux structures de données

Imaginons que nous avons un vecteur d'objets qui ne sont pas `Copy`.
Disons `Vec<Vec<i32>>`.
Comment modifier l'un des `i32` stockés ?

Une autre possibilité :

```
fn get_mut(i: i32, l: &mut Vec<Vec<i32>>) -> &mut Vec<i32>
```

Mais nous avons dit que les emprunts s'arrêtent lorsque la fonction se termine...

- La valeur de retour ne peut pas être utilisée, car elle est un alias du paramètre `l`...

Durées de vie (*lifetimes*)

Les emprunts ont des durées de vie (*lifetimes*), notées `'a`, `'b`, `'c`...

- Dans le cas général, on écrit pour un emprunt : `&'a T` or `&mut 'a T`.
  - Dans certaines situations, cela peut être abrévié en `&T` / `&mut T`.

Les fonctions peuvent être polymorphes sur une durée de vie.
Exemple adapté de la bibliothèque `Vec` :

```
fn last_mut<'a, T>(v: &'a mut Vec<T>) -> Option<&'a mut T>
```

- Renvoie un emprunt avec la même durée de vie que le paramètre.
- Tant que l'emprunt renvoyé est *vivant*, la variable empruntée est inutilisable.
- La fonction appelante peut modifier le contenu du vecteur avec l'emprunt renvoyé, tant qu'elle n'accède pas directement au vecteur.
  - Ceci préserve l'invariant global des alias (mutation XOR alias).
- La durée de vie est inférée automatiquement quand on appelle la fonction.

Lifetimes, example

```
fn last_mut<'a, T>(v: &'a mut Vec<T>)
  -> Option<&'a mut T>
{ ... }

fn f() {
  let mut v: Vec<i32> = vec![1,2,2];

  let opt = last_mut(&mut v);



  match opt {
    Some(last) => *last = 3,
    None => panic!()
  }

}
```

- Quand on crée un emprunt, (avec `&mut v`), Rust crée une variable de durée de vie `'a`.
- `v` est marquée comme empruntée exclusivement pour `'a`.
  - `v` ne peut pas être utilisée tant que `'a` est active.
- Cette variable de durée de vie est unifiée par le vérificateur de types.
  $\Longrightarrow$ la variable `last` a le type `&'a mut i32`
- Contraintes sur `'a` : être vivante lorsque `last` et `opt` peuvent être utilisées.
  $\Longrightarrow$ `'a` est inférée être la zone .

Lifetimes, example

```
fn last_mut<'a, T>(v: &'a mut Vec<T>)
  -> Option<&'a mut T>
{ ... }

fn f() {
  let mut v: Vec<i32> = vec![1,2,2];

  let opt = last_mut(&mut v);



  match opt {
    Some(last) => *last = 3,
    None => panic!()
  }

  v.push(4)
}
```

Ajoutons une mutation de `v` après la fin de `'a`.

Rust vérifie le statut de `v` au moment de l'appel.
$\Longrightarrow$ La durée de vie `'a` est terminée, `v` est disponible.
$\Longrightarrow$ Appel autorisé.

Lifetimes, example

```
fn last_mut<'a, T>(v: &'a mut Vec<T>)
  -> Option<&'a mut T>
{ ... }

fn f() {
  let mut v: Vec<i32> = vec![1,2,2];

  let opt = last_mut(&mut v);

  v.push(42); // Danger!

  match opt {
    Some(last) => *last = 3,
    None => panic!()
  }

}
```

Si on essaie de modifier `v` quand il est encore emprunté.

Rust vérifie le statut de `v` à l'appel.
La durée de vie `'a` n'est PAS terminée, l'appel est impossible.

Lifetimes, example

```
fn last_mut<'a, T>(v: &'a mut Vec<T>)
  -> Option<&'a mut T>
{ ... }

fn f() {
  let mut v: Vec

  let opt = last

  v.push(42); //

  match opt {
    Some(last) =
    None => pani
  }


}
```

Si on essaie de modifier v quand il est encore emprunté.

```
error[E0499]: cannot borrow ‘v‘ as mutable more than once at a time
  --> src/lib.rs:10:3
   |
8  |   let opt = last_mut(&mut v);
   |                      ------ first mutable borrow occurs here
9  |
10 |   v.push(42); // Danger!
   |   ^^^^^^^^^^ second mutable borrow occurs here
11 |
12 |   match opt {
   |         --- first borrow later used here
```

Opérations primitives sur les emprunts

Copier un emprunt partagé (rappel : `&T: Copy`) :

```
let x = &1;
let y = x;
println!("{} {}", *x, *y);
```

Retirer la mutabilité :

```
let x = &mut 1;     // x: &'a mut i32
let y: &i32 = x;  // y: &'a i32
```

Diviser un emprunt (partagé ou unique) :

```
let x = &mut (1, 2);  // x: &'a mut (i32, i32)
let (x1, x2) = x;     // x1: &'a mut i32   x2: &'b mut i32
```

Directement sur une paire/enregistrement/`enum` :

```
let mut p = (42, 12);
let x = &mut p.0;
let y = &p.1; // Autorisé même si p.0 est déjà emprunté exclusivement
let z = &p.1; // Autorisé même si p.1 est déjà emprunté en partage
let t = p.1;  // Idem
*x = 43; // x: &mut i32
println!("{} {} {}", *x, *y, *z); // y, z: &i32
```

Réemprunt

```
fn last_mut<'a, T>(v: &'a mut Vec<T>)
  -> Option<&'a mut T>
{ ... }

let v = &mut vec![1, 2, 3];
match last_mut(&mut(*v)) { ... }
match last_mut(&mut(*v)) { ... }
```

Réemprunter : créer un emprunt plus court du contenu d'un emprunt.
L'ancien emprunt est réactivé lorsque le nouvel emprunt se termine.

- Permet d'utiliser un emprunt unique plusieurs fois.
- Implicite dans la grande majorité des cas.
- La durée de vie de l'emprunt initial doit être plus longue que la durée de vie du réemprunt.

Ordre des durées de vie.

Le mécanisme de réemprunt utilise une notion d'inclusion de durée de vie

- `'a` survit à `'b`, écrit `'a: 'b`.
- (En anglais, `'a` *outlives* `'b`.)

Cette relation d'ordre : sous-typage des durées de vie.

- Polymorphisme de durée de vie contraint par cette relation :

```
fn f<'a, 'b: 'a>(...) -> .. { ... }
fn f<'a, 'b>(...) -> .. where 'b: 'a { ... }
```

Inférence des durées de vie

La plupart des informations de durée de vie est inférée par le compilateur.

- Nous devons seulement annoter les types des fonctions (et les déclarations de types).

Composant clé du compilateur `rustc` : le *borrow checker*.

- Il s'exécute après la vérification traditionnelle des types, et en exploite le résultat.
- Utilise l'information de survie produite par une analyse de flot de donnée.
  - Une variable est vivante si elle peut potentiellement être utilisée plus tard.
  - Différent de la portée : une variable peut être morte mais toujours dans la portée actuelle.
- Suivi de possession (e.g., valeur `Box<T>` consommée une seule fois) et infère les durées de vie.
- Implémenté sur MIR, une représentation intermédiaire sous la forme de graphe de flot de contrôle.

Le *borrow checker*

Inférence des durées de vie

Le *borrow checker* interprètes chaque durée de vie `'a` comme un ensemble $[\![$`'a`$]\!]$ contenant :

- des nœuds du graphe de flot de contrôle,
- des éléments `end('a)` pour les durées de vie `'a` apparaissant dans le type de la fonction.

La vérification des types génère des variables de durée de vie fraîches et des contraintes de survie "`'a: 'b`" :

- Exemples : réemprunt, appels de fonctions, coercions, ...
- Interprété comme des inclusions d'ensembles $[\![$`'b`$]\!] \subseteq [\![$`'a`$]\!]$

Contraintes supplémentaires ajoutées :

- `L` $\in [\![$`'a`$]\!]$ quand `'a` apparaît dans le type d'une variable vivante au nœud `L` ;
- `end('a)` $\in [\![$`'a`$]\!]$ pour toutes les variables de polymorphisme `'a` ;
- `L` $\in [\![$`'a`$]\!]$ pour toutes les variables de polymorphisme `'a` et tout nœud `L`.

Le *borrow checker* cherche la plus petite solution des ces contraintes (algo. point fixe).

Le *borrow checker*

Valider le programme

Quand les ensembles $[\![$`'a`$]\!]$ sont calculés, il reste à vérifier :

- que les accès aux variables sont valides :
  - Pas de lecture si variable non initialisée ou possession extraite.
  - Pas d'accès si la variable est marquée empruntée exclusivement pour une durée de vie active.
  - Pas d'écriture si la variable est marquée empruntée pour une durée de vie active.
  - Question : quand emprunter est-il autorisé ?
- que les contraintes de survie déclarées dans le type de la fonction sont suffisantes :
  - L'algorithme de résolution de contraintes a trouvé la plus petite solution.
  - Si `end('a)` $\in [\![$`'b`$]\!]$, c'est que le corps de la fonction nécessite `'b: 'a`.
  - Il faut donc vérifier que pour n'importe quelles variables de polymorphisme `'a` et `'b`, si `end('a)` $\in [\![$`'b`$]\!]$, alors les contraintes de survies du type de la fonction impliquent `'b: 'a`.

Le *borrow checker*

Valider le programme

Quand les ensembles $[\![\texttt{'a}]\!]$ sont calculés, il reste à vérifier :

- que les accès aux variables sont valides :
  - Pas de
  - Pas d'
    active.
  - Pas d'
  - Questi
- que les con
  - L'algor
  - Si `end('a)` $\in [\![\texttt{'b}]\!]$, c'est que le corps de la fonction nécessite `'b: 'a`.
  - Il faut donc vérifier que pour n'importe quelles variables de polymorphisme `'a` et `'b`, si `end('a)` $\in [\![\texttt{'b}]\!]$, alors les contraintes de survies du type de la fonction impliquent `'b: 'a`.

Complications supplémentaires in `rustc` :

- Plutôt que des variables de programmes : places
  - E.g., `rustc` traite `p.0` et `p.1` indépendamment.
- Fonctionnalités du langage : clôtures, ...
- Algorithme de point fixe efficace.
- ...