Programmation avancée

TD3: Introduction à Rust

Jacques-Henri Jourdan Armaël Guéneau Arnaud Golfouse

Pensez bien à vérifier que votre code compile. Si vous n'avez pas terminé, remplacez les parties manquantes par `todo!()`.

Note Téléchargez le code de démarrage de ce TP à l'adresse suivante : `https://jhjourdan.gitlabpages.inria.fr/prog3-l3-ensps/tds/03-rust-intro/td03.zip`

1 Échauffement

1.1 Types algébriques

Exercice 1. *Donnez un type* `Point` *correspondant à un point en 3 dimensions avec des coordonnées (de type flottant) appelées* `x`, `y`, `z` *respectivement.*

Exercice 2. *Définissez le point zéro en utilisant une* constante associée*; on veut pouvoir utiliser* `Point::ZERO` *pour y référer.*

Exercice 3. *Implémentez une fonction* `new_normed` *qui prend en entrée trois coordonnés et renvoie le vecteur* normalisé *indiqué par ces coordonnées.*

Exercice 4. *Définissez le produit scalaire de deux points en utilisant une* méthode*. Il faut qu'on puisse faire* `p1.dot(p2)`*. Ensuite, définissez le produit vectoriel (*`cross`*).*

1.2 Appartenance et Emprunts

Exercice 5. *Implémentez la fonction :* `fn concat(v1 : Vec<i32>, v2: Vec<i32>) -> Vec<i32>` *qui renvoie la concaténation de* `v1` *et* `v2`*.*

Exercice 6. *Écrivez une variante de* `concat` *qui pour prendre ses paramètres par* emprunts *et modifie* `v1` *en place. Vous pouvez appeler cette fonction* `concat2`*.*

Exercice 7. *Pourquoi ne peut-on pas implémenter une fonction avec la signature suivante ?*

```
fn bad<'a, T>(x: T) -> &'a mut T
```

*Que ferait une telle fonction dans votre machine ? (*Indice*: comment fonctionne la pile d'appels ?)*

Exercice 8. *Pourquoi ne peut-on pas implémenter une fonction* `fn dup<T>(x: T) -> (T, T)` *? De quelle caractéristique du système de types de Rust cela vient-il ? Quelles conditions faut-il imposer à* `T` *pour implémenter* `dup` *?*

Exercice 9. *Implémentez la fonction* `fn insertion_sort(&mut Vec<i32>)` *qui modifie le vecteur en place.*

2 Mémoire

On va chercher, comme en Ocaml, à savoir quelle est la place occupée par certaines structures.

Il n'y a pas d'équivalent de `memgraph` en Rust, on se contentera donc d'utiliser certaines fonctions de la bibliothèque standard.

> *Remarque* : Pour juste lancer une fonction (et non des tests), on doit créer un fichier `main.rs` à côté de `lib.rs`, et écrire une fonction `fn main() { /* */ }` dedans.
>
> On peut ensuite utiliser `cargo run` pour exécuter cette fonction.

Exercice 10. *La fonction* `std::mem::size_of::<T>()` *renvoie la taille, sur la pile, d'un objet de type* `T` *(en octets).*

Quelle est la taille en octets de `u8`*,* `u32`*,* `u64` *et* `usize` *?*

Exercice 11. *La fonction* `std::mem::size_of_val(&x)` *renvoie la taille en octets, sur la pile, de l'objet* `x`*.*

Si on définit `let b = Box::new(42);`*, quelle est la taille de* `b` *? Et* `*b` *? Combien de mémoire (en octets, pile + tas) est alors allouée par cet objet ?*

Exercice 12. *Si on définit* `let v = vec![1, 2, 3, 4, 5];`*, quelle est la taille de* `v` *? Et* `v[0]` *? Combien de mémoire (en octets, pile + tas) est alors allouée par cet objet ?*

3 Arbres de Recherche

Nous allons maintenant implémenter un *arbre de recherche binaire*. Pour fixer les idées, on ne traitera que le cas simple ou les clés et valeurs sont des entiers, la généralisation est laissé en bonus.

Cependant, pour s'assurer que l'on respecte bien les règles de possession, on utilisera pour les valeurs un alias du type `u64` des entiers 64 bits non signés, qui n'implémente *pas* `Copy` :

```
// Un type opaque de valeur, avec suivi de la possession
mod private {
    #[derive(PartialEq, Eq, Debug)]
    pub struct Value(u64);
    // Vous pouvez définir une fonction `new` pour créer de nouvelles valeurs
}
use private::Value;
```

On utilisera le type d'arbre binaire suivant :

```
#[derive(PartialEq, Eq, Debug)]
struct Node {
    left: BST,
    key: u64,
    val: Value,
    right: BST,
}

#[derive(PartialEq, Eq, Debug)]
enum BST {
    Leaf,
    Node(Box<Node>),
}
```

On maintiendra sur ce type l'invariant que l'arbre a la possession d'un arbre de recherche et que les clés contenues sont uniques.

Exercice 13. *Implémentez la fonction* `BST::new` *qui renvoie un arbre vide.*

Exercice 14. *Implémentez la* méthode `fn find<'a>(&'a self, k: u64) -> Option<&'a Value>`.

Exercice 15. *Implémentez la* méthode `find_mut` *qui cherche une valeur associée à une clef, et renvoie un* emprunt mutable *sur la* valeur.

Exercice 16. *Implémentez la* méthode `fn insert(&mut self, k: u64, v: Value) -> Option<Value>` *qui insère une nouvelle valeur associée à une clef, et renvoie la valeur précédente si il y en a une. Vous pouvez utiliser la fonction* `std::mem::replace` *dont vous chercherez la documentation.*

4 Itérateurs

Pour parcourir une structure en Rust on utilise fréquemment des *itérateurs*. Chaque itérateur est donné comme un type avec une méthode `next` qui renvoie les valeurs successives de la structure selon l'ordre d'itération demandé. Ce mécanisme est utilisé pour implémenter les boucles `for` de Rust.

Considérons l'exemple suivant d'un itérateur pour un *vecteur*[1].

```
impl<'a, T> Vec<T> {
    fn iter(&'a self) -> Iter<'a, T> {
        Iter { ix: 0, vec: self }
    }
}

struct Iter<'a, T> {
    ix: usize,
    vec: &'a Vec<T>,
}

impl<'a, T> Iter<'a, T> {
    fn next(&mut self) -> Option<&T> {
        let res = self.vec.get(self.ix);
        self.ix += 1;
        res
    }
}
```

1. Ceci est simplement une illustration du mécanisme et pas l'implémentation actuelle pour les vecteurs.

Exercice 17. *Implémentez un* draining iterator*, c'est a dire un itérateur qui* consomme *l'arbre et renvoie des paires de clés et valeurs en* possession complète*. Vous pouvez utiliser le type suivant :*

```
struct IterDrain {
    cur: BST,
    stack: Vec<(u64, Value, BST)>,
}
```

Le champ `cur` *représente le sous-arbre actuellement parcouru pendant que* `stack` *contiens les clés, valeurs et sous-arbres a droite.*

Indice: *Vous pouvez utiliser* `std::mem::replace`*.*

Implémentez également une méthode `BST::into_iter`*, de signature* `fn into_iter(self) -> IterDrain`*, qui initialise l'itérateur.*

Exercice 18. *Écrivez l'itérateur immuable* `IterImmut` *qui prend un* `&'a BST` *et renvoie des paires d'emprunts sur la clé et la valeur d'une entrée de l'arbre.*

Implémentez également une méthode `BST::iter`*, de signature* `fn iter(&self) -> IterImmut`*, qui initialise l'itérateur.*

Note *: vous pouvez vous inspirer de l'exemple sur les vecteurs pour définir le type d'itérateur.*

Exercice 19. *Écrivez l'itérateur mutable* `IterMut` *qui prend un* `&'a mut BST` *et renvoie des paires d'un emprunt immuable sur la clé et un emprunt* mutable *sur la valeur.*

Si nécessaire, vous pourrez changer le type de `cur` *en* `Option<BST>`*.*

Implémentez également une méthode `BST::iter_mut`*, de signature* `fn iter_mut(&mut self) -> IterMut`*, qui initialise l'itérateur.*

Que se passerait-il si on renvoyait un emprunt mutable *sur la clé ?*

Rustlings pour la semaine prochaine

Faire les exercices de la partie `10_modules`.

Remarques :

- Pour passer un exercice, on peut aller dans le fichier `info.toml` et commenter l'exercice correspondant.
- Pour avoir de l'autocomplétion, etc… Il faut lancer la commande `rustlings lsp`, puis redémarrer son éditeur.