Programmation avancée

TD4 : Traits

Jacques-Henri Jourdan Armaël Guéneau Arnaud Golfouse

Inspiré de la librairie `ndarray`.

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/04-traits/td04.zip`

1 Échauffement

La macro `println!` permet d'afficher sur la sortie standard. Pour afficher une variable, on peut utiliser des crochets :

```
let x: i32 = 5;
println!("x vaut {}", x);
```

Exercice 1. *Pourquoi la fonction suivante ne compile-t'elle pas ?*

```
fn my_print<T>(x: T) {
    println!("x vaut {}", x);
}
```

Changez la signature de cette fonction, de manière à ce qu'elle compile et affiche la valeur de la variable `x`.

Exercice 2. *Définir une structure `Color`, avec trois champs `r`, `g` et `b` de type `u8`.*

Pourquoi ne peut-on pas appeler `my_print` avec une variable de type `Color` ?

Implémentez le trait `Display` (`https://doc.rust-lang.org/std/fmt/trait.Display.html`) sur `Color` pour résoudre ce problème.

Exercice 3. *Définir `Display` sur la structure `struct Display2<T, U>(T, U);`, de manière à ce que `Display2` affiche ses 2 arguments l'un à la suite de l'autre.*

2 Itérateurs

En Rust les boucles `for` sont définis en utilisant le trait `std::iter::Iterator` de la librairie standard. Ce trait peut être résumé sous la forme suivante:

```
trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
}
```

Quand on appelle `next` sur l'itérateur, il produit la valeur suivante si il en existe une, et sinon il s'arrête en produisant `None`.

Exercice 4. *Donner une fonction `is_evens` qui prend un itérateur produisant des valeurs de type `u32` et détermine si elles sont paires.*

Exercice 5. *Écrire l'itérateur `Circle` qui compte en boucle de `0` jusqu'à une valeur `n` (exclue, de type `u32`) donnée.*

Exercice 6. *Pourquoi le code suivant, qui fait la somme des nombres de `0` à `n - 1`, n'est-t'il pas correct ?*

```
fn somme_n(n: u32) -> u32 {
    let it = circle(n);
    let mut total = 0;
    for x in it {
        total += x;
    }
    total
}
```

Utilisez un adaptateur d'itérateur (`https://doc.rust-lang.org/std/iter/#adapters`) pour corriger ce code.

Exercice 7. *En utilisant l'adaptateur `map` (`https://doc.rust-lang.org/std/iter/trait.Iterator.html#method.map`), écrire une fonction qui fait la somme des entiers* pairs *entre `0` et `n - 1`.*

Exercice 8. *Donner une fonction `is_sorted` qui prend un itérateur produisant des valeurs d'un type `T` implémentant le trait `Ord` et détermine si elles sont triées.*

Considérons l'itérateur `Bomb`:

```
struct Bomb { tick: u32, boom: false }
```

```rust
impl Iterator for Bomb {
    type Item = ();

    fn next(&mut self) -> Option<()> {
        if self.boom { return None };
        if self.tick == 0 {
            self.boom = true;
            return Some(())
        } else {
            self.tick -= 1;
            return None
        }
    }
}
```

Exercice 9. *Cet itérateur est-il* valide *selon la documentation du trait ? Que se passe-t-il si on utilise cet itérateur dans une boucle* `for` *?*

3 Tenseurs

Nous allons développer une version simplifié d'une librairie de tenseurs en Rust. Nous allons aussi explorer comment nous pouvons utiliser les traits de Rust pour généraliser nos opérations de tenseurs.

Un *tenseur* est une généralisation de la notion de vecteur et de matrice (au sens mathématique). En effet, alors qu'un vecteur est une séquence de nombres indexés par un entier et une matrice un tableau de nombres indicés par des paires d'entiers (ligne, colonne), la notion de tenseur permet d'aller au delà, et de définir des objets regroupant des nombres indicés par des n-uplet d'entiers, où n est appelé *l'ordre* du tenseur. Ainsi:

- un simple nombre (scalaire) est un tenseur d'ordre 0,
- un vecteur est un tenseur d'ordre 1,
- une matrice est un tenseur d'ordre 2,
- un tableau n-dimensionnel de nombres est un tenseur d'ordre n.

Tout comme on peut parler de la dimension d'un vecteur (un nombre entier) ou de la dimension d'une matrice (une paire d'entiers: son nombre de lignes et de colonnes), on peut parler de la dimension d'un tenseur d'ordre n: il s'agit d'un n-uplet d'entiers.

Pour commencer, nous allons donc définir le type générique `Dim`, représentant la dimension d'un tenseur, ainsi que des spécialisations pour les ordres de 1 a 4.

```rust
struct Dim<I>(I);

type Dim0 = Dim<[usize; 0]>;
type Dim1 = Dim<[usize; 1]>;
type Dim2 = Dim<[usize; 2]>;
type Dim3 = Dim<[usize; 3]>;
```

Le type `[T; N]` est le type des tableaux d'éléments de type `T` et de longueur `N`. Ainsi, si `N` est l'ordre d'un tenseur, on peut représenter sa dimension (un `N`-uplet) par une valeur de type `[usize; N]`. Pour expliciter le fait qu'un tel tableau représente la dimension d'un tenseur, on le met dans un type dédié, `Dim`.

Nous pouvons ensuite définir le type des tenseurs, ainsi que des alias pour les cas communs:

```
struct Tensor<T, D>(Vec<T>, D);

type Vector<T> = Tensor<T, Dim1>;
type Matrix<T> = Tensor<T, Dim2>;
```

Ainsi, pour stocker un tenseur, on utilise une paire: la première composante contient les données brutes sous la forme d'un vecteur, et la deuxième composante contient sa dimension. Les données d'un tenseur sont stockées « en ligne » dans un vecteur: comme nous le verrons, pour accéder à un élément, il faut convertir un `N`-uplet d'indices en un offset représentant la position dans ce vecteur.

En Rust, le trait `std::ops::Add` permet de surcharger l'opérateur d'addition (potentiellement hétérogène).

Exercice 10. *Donner une implémentation de l'addition des vecteurs (de type `Vector<u64>`), sous la forme d'une instance `&Vector<u64>: Add<&Vector<u64>>`. Elle pourra échouer si les vecteurs passés en paramètre ne sont pas de la même dimension.*

Exercice 11. *Il existe plusieurs types d'entiers en Rust, comment pourrait-on généraliser l'instance d'addition de la question précédente ? Proposer une solution et l'implémenter.*

Exercice 12. *Définir l'addition entre deux tenseurs de même ordre (donc dont la dimension est représenté par le même type `D`). Quelle condition doit-on imposer aux éléments du tenseur ?*

Le trait `std::ops::Index` permet de surcharger l'opérateur `[...]` permettant par exemple d'accéder en lecture un tableau ou un vecteur (par exemple dans l'opération `tab[i]`). Nous voulons donner une instance générique de `std::ops::Index` pour tous les tenseurs, afin de pouvoir écrire, par exemple, `vec[i]`, ou `mat[(i, j)]`.

Pour cela, nous allons définir un trait auxiliaire `NdIndex`, qui permettra de transformer un `N`-indice (i.e., l'objet que nous allons utiliser pour indexer un tenseur) en un offset dans notre représentation en mémoire.

```
trait NdIndex<D> {
    fn offset(&self, dim: D) -> usize;
}
```

Exercice 13. *Définir les instances attendues pour indexer les éléments de nos tenseurs. Par exemple, nous voulons pouvoir utiliser un entier de type* `usize` *pour indicer un* `Vector`*, ou une paire* `(usize, usize)` *pour indicer une matrice, etc…*

Une opération essentielle sur les tenseurs est l'extraction de sous-tenseur. Par exemple, dans une matrice on peut sélectionner une colonne ou même une sous-matrice.

Pour faire cela nous définissons un type `View<'arr, T, D>` qui représente une *vue* d'une sous-partie d'une tenseur de dimension `D1` comme un tenseur de dimensions `D`.

```
struct View<'arr, T, D> {
    data: &'arr [T],
    dim: D,
    stride: D,
}
```

Le champ `stride` indique le nombre d'éléments à sauter chaque fois qu'on fait un pas le long d'un axe. Par exemple, pour une matrice 3×3 représentée directement, les strides seront $(3, 1)$.

Exercice 14. *Implémenter le trait* `Index` *pour* `View`*. Pour cela il faudra étendre le trait* `NdIndex` *pour que* `offset` *prenne aussi l'ensemble de* strides *d'une view.*

Rustlings pour la semaine prochaine

Faire les exercices des parties `14_generics` et `15_traits`.