Programmation avancée

TD6 : Mutabilité intérieure

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!()`.

Pensez également à décommenter les tests au fur et à mesure, afin de tester votre implémentation avec `cargo test`.

Note Téléchargez le code de démarrage de ce TP à l'adresse suivante : `https://jhjourdan.gitlabpages.inria.fr/prog3-l3-ensps/tds/06-interior-mut/td06.zip`

1 Implémentation de la mutabilité intérieure.

On va implémenter une version (très) simplifiée de `Rc` et `RefCell` par nous-même. Cette implémentation va nécessairement devoir utiliser `unsafe` !

1.1 `Rc`

`Rc` est un raccourci de « Reference Counted » : c'est un pointeur qui peut être dupliqué, et qui garde trace du nombre de pointeur pointant vers le même objet. Lorsque ce compte atteint `0`, l'objet est détruit (Il y a plus de subtilité en vrai, mais c'est l'idée de base).

À chaque fois qu'on ouvrira un block `unsafe`, on essaiera d'ajouter un commentaire `// SAFETY: ...` qui explique pourquoi ce qu'on fait dans le block est justifié (autant que possible, ne pas se faire trop de nœuds au cerveau non plus).

Par exemple:

```
let ptr: *mut i32 = ... ;
// SAFETY: on peut accéder à ptr en écriture, car l'objet sous-jacent est encore vivant, mutable, et aucune référence à celui-ci n'existe actuellement.
unsafe {
    *ptr += 1;
}
```

Exercice 1. *On définit une structure* `RcInner`*, qui contient l'objet à allouer, ainsi qu'un compte.*

Définir la fonction `fn Rc::new(value: T) -> Rc<T>`*. On utilisera la fonction* `Box::leak` *pour obtenir un pointeur.*

Exercice 2. *Implémenter le trait* `Deref` *sur* `Rc`*, de sorte qu'on puisse utiliser un* `&Rc<T>` *comme un* `&T`*.*

Pour le moment, notre `Rc` n'est pas très utile, car on ne peut pas le dupliquer…

Exercice 3. *Implémenter le trait* `Clone` *sur* `Rc`*. Notre implémentation devra augmenter le compte du* `RcInner`*.*

Exercice 4. *Enfin, implémenter le trait* `Drop` *sur* `Rc`*. Cette implémentation doit réduire le compte de* `1`*, et libérer l'objet sous-jacent si le compte atteint* `0`*.*

On utilisera la fonction `Box::from_raw` *pour reconstruire une* `Box` *à partir d'un pointeur.*

1.2 `RefCell`

`RefCell` est une implémentation de la *mutabilité intérieure* : avec seulement un emprunt partageable `&RefCell`, on peut lire ou modifier l'objet sous-jacent. C'est autorisé, car on va vérifier *à l'exécution* qu'on ne créer jamais deux références aliasées.

Le type de base de la mutabilité intérieure est `UnsafeCell`.

Exercice 5. *Pourquoi ne peut-on pas avoir directement les fonctions* `fn borrow(&RefCell<T>) -> &T` *et* `fn borrow_mut(&RefCell<T>) -> &mut T` *?*

On doit à la place définir des types `Ref<'a, T>` et `RefMut<'a, T>`, qui vont *verrouiller* l'accès à la `RefCell` jusqu'à la fin de leur durée de vie.

Si on tente d'appeler `borrow` ou `borrow_mut` lorsque la `RefCell` est verrouillée, cela lance une `panic!`.

Exercice 6. *Ajouter un champ* `locked: Cell<bool>` *à* `RefCell`*, et définir les méthodes* `fn borrow(&self) -> Ref<'a, T>` *et* `fn borrow_mut(&self) -> RefMut<'a, T>`*.*

Exercice 7. *Implémenter les traits* `Deref` *et* `DerefMut` *sur* `Ref` *et* `RefMut`*.*

Enfin, implémenter le trait `Drop` *sur* `Ref` *et* `RefMut`*, afin de déverrouiller la* `RefCell`*.*

2 Tableaux Persistants (Redux)

Nous souhaitons implémenter une bibliothèque de tableaux persistants en Rust. Cette bibliothèque sera capable de représenter des tableaux de manière persistante, c'est-à-dire sans supporter de manière observationnelle des modifications en place. En particulier, la fonction « set » de cette bibliothèque ne mutera pas (observationnellement) le tableau donné en paramètre, mais retournera plutôt un nouveau tableau persistant représentant la version mise à jour.

En interne, un tableau persistant est soit représenté comme un tableau mutable traditionnel, soit comme une valeur spécifique représentant une mise à jour d'un autre tableau persistant. C'est-à-dire qu'un tableau persistant sera représenté comme une séquence de mises à jour d'un tableau mutable stocké traditionnellement en mémoire. Les accès au tableau modifieront la représentation interne des valeurs du tableau persistant pour s'assurer que les valeurs utilisées récemment ont un faible nombre de mises à jour successives du tableau physique. Dans la littérature, ceci est connu sous le nom de « Baker's Trick ».

En Rust, nous utiliserons un module `parray` pour cette bibliothèque, contenant les définitions de types suivantes :

```
// fichier src/parray.rs
use crate::cell::RefCell;
use crate::rc::Rc;

enum PAState<T> {
    Arr(Vec<T>),
    Diff(usize, T, PArray<T>),
    Invalid,
}
use self::PAState::*;

pub struct PArray<T>(Rc<RefCell<PAState<T>>>);

impl<T> PArray<T> {
    fn reroot(&self) {
        panic!()
    }
}
```

(On préférera probablement utiliser `std::rc::Rc` et `std::cell::RefCell`, qui sont plus complètes et dont on est raisonnablement sûr de la correction de l'implémentation)

Le type public des tableaux persistants est `PArray`. Il contient un pointeur à comptage de références (pour permettre le partage facile) vers un `RefCell` (pour permettre le changement de la représentation interne) d'une valeur de type `PAState<T>`. Le type `PAState<T>` est une énumération représentant les deux représentations possibles d'un

tableau persistant : soit `Arr` pour un tableau traditionnel, soit `Diff` pour une mise à jour d'un autre tableau persistant. Le constructeur Invalid peut être utilisé comme valeur par défaut lorsqu'on souhaite prendre possession du contenu d'un `RefCell<PAState<T>>`.

La fonction associée reroot, dont l'implémentation sera remplie plus tard, vise à ré-enraciner la représentation interne d'un tableau persistant : tout en conservant la valeur publiquement visible des tableaux persistants, elle modifie la représentation interne des tableaux persistants de sorte que self pointe directement vers un tableau traditionnel.

Exercice 8. *Écrire la fonction associée `PArray::new` qui crée un nouveau tableau persistant à partir d'un contenu donné.*

Implémenter également le trait `Clone` pour `PArray<T>`, afin que les tableaux persistants puissent être partagés à faible coût.

Exercice 9. *Écrire les deux fonctions `PArray::get` et `PArray::set`, qui effectuent des accès au tableau.*

Ces fonctions n'utilisent pas `reroot`.

Exercice 10 (Bonus). *Implémenter les fonctions `PArray::reroot`, `PArray::get_reroot` et `PArray::set_reroot`.*

`PArray::get_reroot` et `PArray::set_reroot` devraient ré-enraciner les tableaux comme expliqué ci-dessus : `get_reroot` devrait ré-enraciner son paramètre, `set_reroot` devrait renvoyer un tableau ré-enraciné.

3 Union-Find

Union-Find est un algorithme incontournable qui permet de calculer les classes d'équivalence d'un ensemble d'éléments vis-à-vis d'une relation d'équivalence. Il est central dans l'implémentation de *l'unification* dans un type-checker.

Pour faire cela, on utilisera des arbres pour représenter chaque classe d'équivalence. Chaque nœud garde un pointeur vers son *parent* (plutôt que des pointeurs vers ses enfants). À cette structure on donne les deux opérations éponymes: find prend un nœud quelconque et renvoie son représentant (la racine de l'arbre), union prend deux nœuds et combine leurs classes d'équivalence.

Pour implémenter Union-Find de manière efficace on aura besoin de partage. Comme on utilise Rust, on ne peut pas exprimer cela avec de simple emprunts, il faudra qu'on utilise la *mutabilité intérieure*.

On utilisera le type suivant:

```
#[derive(PartialEq, Eq, Debug, Clone)]
struct Node<T>(Rc<RefCell<Inner<T>>>);
```

```rust
#[derive(PartialEq, Eq, Debug)]
enum Inner<T> {
    Root { data: T },
    Link { parent: Node<T> },
}

impl Node<T> {
    fn new(data: T) -> Self {
        Node(Rc::new(RefCell::new(Root { data })))
    }
}
```

Le type `Node` défini ci-dessus est en fait un *pointeur* vers soit un `Root` ou un `Link`.

Exercice 11. *Implémenter les fonctions `Node::naive_find` et `Node::naive_union`.*

Bonus : éviter de faire déborder la pile dans le test `perf_naive`.

L'implémentation naïve des exercices précédents fonctionne, mais peut rapidement dégrader sa performance car l'arbre de Union-Find que l'on construit peut facilement se transformer en liste chaînée ! Pour remédier à ce problème nous allons faire une optimisation simple : on associera à chaque racine d'un arbre un *rang* qui représente la *hauteur* de l'arbre correspondant. Quand on combine deux arbres avec `union`, on pourra placer l'arbre avec la plus petite hauteur sous le plus grand, ce qui a pour effet d'éviter de faire grandir la hauteur.

Exercice 12. *Ajouter à `Root` un champ `rank: usize` et mettre à jour les définitions de `new` et `union` pour implémenter cette optimisation. Utiliser les tests précédents pour vérifier que votre implémentation fonctionne toujours.*

Un autre problème de performance que l'on peut rencontrer avec Union-Find est que sur des grands arbres on peut finir par répéter beaucoup de parcours de l'arbre en appelant `find` répétitivement. Si on peut se rappeler du représentant de chaque classe immédiatement on peut économiser beaucoup de temps de calcul. Pour faire cela on compressera les chemins dans l'arbre: si le représentant du parent d'un nœud n'est pas le même que le parent, on replacera notre parent par ce représentant directement.

Exercice 13. *Ajouter la compression de chemin à votre implémentation de `find`.*

Rustlings pour la semaine prochaine

Faire les exercices de la partie `19_smart_pointers`.