Programmation avancée

TD8 : Gestion de la mémoire

Jacques-Henri Jourdan Armaël Guéneau Arnaud Golfouse

Adapté d'un sujet de Jean-Christophe Filliâtre — transcrit par Xavier Denis, 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 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/08-memoire/td08.zip`

L'objectif du TD est de programmer un GC *stop & copy* en Rust.

Les valeurs de notre « language » (type `Value`) seront des blocs, contenant soit des entiers, soit des pointeurs vers d'autres valeurs.

Notre type `Memory` a la forme suivante :

```
#[derive(Debug)]
pub struct Memory {
    mem: Vec<InnerValue>,
    roots: HashMap<String, Addr>,
    active_region: Region,
    next_free: Addr,
}
```

On modélise la mémoire de manière très simple, comme un vecteur de valeurs (le champ `mem`). Dans cette mémoire un *bloc* de taille n situé à l'adresse p est représenté de la manière suivante :

- `mem[p]` contient n (la taille du bloc)
- `mem[p+1]`, …, `mem[p+n]` contiennent les n champs.

On note en particulier que $n + 1$ éléments du tableau `mem` sont donc utilisés par ce bloc.

Cette mémoire est ainsi séparée en deux régions : la première moitié correspond à `Region::Lower`, et la deuxième moitié à `Region::Upper`.

Les racines sont déclarées dans la table `roots`, intuitivement elles correspondent aux variables de nos programmes.

1 Échauffement

Architecture :
- `src/memory.rs` : contient les fonctions liées au GC : c'est le fichier que vous devez remplir.
- `src/memory/tests.rs` : tests pour votre implémentation, ne pas hésiter à lire ce fichier.
- `src/memory/address.rs` : définition du type `Addr`.
- `src/memory/values.rs` : opérations pour lire/écrire des `Value` dans la mémoire.

On va commencer par écrire quelque fonctions utilitaires qui nous seront utiles pour implémenter notre GC.

Exercice 1. *Implémenter la méthode* `region_start` *qui renvoie la première adresse de la région active, ainsi que* `next_region_start` *qui renvoie la première adresse de la région inactive.*

Exercice 2. *Implémenter la méthode* `free_cells` *qui renvoie le nombre de cellules libres dans la région active.*

Exercice 3. *Implémenter la méthode* `in_active_region` *qui détermine si une adresse est dans la région active ou dans la région inactive.*

2 Implémentation du ramasse-miettes

Exercice 4. *Remplir le corps de* `fn move_block(&mut self, src: Addr)` *qui bouge le bloc se trouvant à* `src` *vers les prochaines cases disponibles données par* `next_free` *(fonction* « déplace » *dans le cours).*

Note *Cette fonction modifiera la valeur de* `next_free` *pour qu'elle indique la prochaine case disponible après la copie.*

Exercice 5 (Collection). *Implémenter la fonction* `fn stop_and_copy(&mut self)` *qui prends un emprunt mutable sur la mémoire et déplace les blocs vivants de la zone active, vers la zone inactive. Un bloc est considéré vivant si il est atteignable depuis une racine du GC. Cette fonction changera la zone active.*

Remarque *Attention à gérer correctement les pointeurs invalides (*`Addr::INVALID`*).*

Exercice 6. *Compléter les fonctions* `add_root`, `remove_root` *et* `variable_address` *pour manipuler les racines.*

Exercice 7 (Allocation). *Écrire la fonction* `fn alloc(&mut self, size: usize) -> Option<Addr>` *qui prends une taille de bloc en entrée et renvoie l'adresse du bloc alloué. Si le bloc ne peut être alloué directement, cette fonction doit appeler* `stop_and_copy` *pour libérer de la mémoire. Si le bloc ne peut toujours pas être alloué à l'issue de la collection, on renverra* `None`.

3 Tri d'une liste chaînée

Architecture :
- `src/sort_linked_list.rs` : c'est le fichier que vous devez remplir.
- `src/sort_linked_list/tests.rs` : tests pour votre implémentation, ne pas hésiter à lire ce fichier.

Pour tester notre allocateur, nous allons écrire une fonction de tri par insertion sur les listes chaînées. Une cellule de la liste sera une paire d'une valeur et d'un pointeur vers la queue. Si le pointeur est `Addr::INVALID`, alors on est à la fin de la liste.

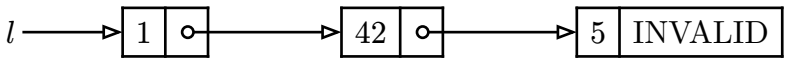

Figure 1. – Représentation mémoire d'une liste chaînée l

Exercice 8. *Écrire les fonctions d'accès* `head` *et* `tail` *qui prennent l'adresse d'une case de liste et qui renvoie la valeur de cette case et l'adresse de la queue respectivement.*

On utilisera la racine `l` comme variable pour stocker le pointeur actuel de notre liste.

Exercice 9. *Écrire la fonction* `cons_l` *qui prends une valeur et rajoute une case à l'avant de la liste chaînée stockée dans* `l`.

Exercice 10. *Écrire la fonction* `make_random` *qui crée une liste de taille i avec des valeurs aléatoires (et la stocke dans* `l`*). Pour créer une valeur aléatoire en Rust on peut utiliser l'expression suivante :* `rand::thread_rng().gen_range(0..200)` *(génère une valeur entre 0 et 199 inclus).*

Exercice 11. *Écrire la fonction* `fn insert_sorted(m: &mut Memory, value: i64)` *qui insert la valeur* `value` *en maintenant que la liste* `l` *soit triée dans l'ordre croissant.*

Exercice 12. *Écrire la fonction* `sort` *qui trie la liste* `l` *dans l'ordre croissant.*