États de pointeur

Un pointeur peut être dans les états suivants:

- `NULL`: valeur spéciale qui ne pointe sur rien
- fou (dangling): l'objet pointé n'existe plus
- fuite (leak): l'objet pointé n'est plus nécessaire
- normal

Les pointeurs fous et les fuites sont les deux problèmes fondamentaux liés aux pointeurs.

Gestion mémoire

Allocation (facile) désallocation (aha!)

Comme les boîtes à vitesse:

- Automatique: la désallocation est prise en charge par le langage
- Manuelle: la désallocation est à la charge du programmeur
- Semi-automatique: le langage vérifie les désallocations

Granularité:

- Par objet: chaque objet est alloué/désalloué individuellement
- Par région: chaque objet est alloué dans une région, la désallocation opère sur tous les objets d'une région

Gestion mémoire manuelle en C

Deux fonctions de la *bibliothèque* standard:

```
void *malloc (int n);
void free (void *ptr);
```

Fonctions typiquement implantées en C; rien de spécial!

Utilisent des primitives du SE pour obtenir de la mémoire, e.g. `mmap`

Ces fonctions partagent des informations administratives internes

E.g. une *free list* de zones mémoire encore disponibles

`free` peut avoir besoin de savoir combien de bytes sont libérés

$\Rightarrow$ la taille `n` peut être stockée par `malloc` juste *avant* les `n` bytes

Gestion mémoire par région

Au lieu de 2 opération *allouer N bytes* et *libérer ces bytes*:

```
region *region_new (void);
void   *region_alloc (int n, region *r);
void    region_free (region *r)
```

La création de région parfois reçoit une information de taille

Un objet encore nécessaire empêche toute sa région d'être libérée

`region_alloc` plus efficace que `malloc`

`region_free` plus efficace que plusieurs `free`

Plus facile de savoir quand libérer une région qu'un objet

Gestion mémoire automatique

- Comptage de références: à chaque objet est associé un compteur qui indique combien de pointeurs existent. Lorsque le compteur passe à 0, on peut désallouer l'objet.
- GC (Glanage de Cellules ou plutôt Garbage Collection): à partir des *racines* (i.e. les variables globales et la pile), traverser tous les objets atteignables en passant par tous les pointeurs: les objets non-visités peuvent être désalloués.
- Régions: une analyse sophistiquée du code détermine dans quelle région allouer chaque objet, et à quel moment désallouer chaque région.

Compter les références

Chaque objet contient un champ *refcnt*

refcnt compte les références “entrantes” (qui pointent sur cet objet)

Copie resp. destruction de pointeur incrémente resp. décrémente *refcnt*

Quand $\mathit{refcnt} = 0$:

- Décrémenter les *refcnt* des objets pointés
- Libérer l’objet

Simple à implémenter, récupération prompte, mais coûteux

Difficulté de compter les références

Attention à décrémenter *après* incrémenter

Coût de tous ces incréments et décréments

Synchronizer les incréments et décréments en cas de concurrence

Incapable de récuperer les cycles

Mark&Sweep

Chaque objet contient un *markbit* qui indique si l'objet est accessible

Commencer par marquer tous les objets comme "inaccessibles"

Mark: Marquer récursivement tous les objets accessibles

- Commencer par les *racines*
- Suivre tous les pointeurs des objets rencontrés

Sweep: Récupérer tous les objets encore marqués "inaccessible"

Mark (&Sweep)

```
mark (ptr) {
    if (!ptr->marked) {
        ptr->marked = True;
        for i = 0 to ptr->size
            mark (ptr[i]);
}   }

mark_all () {
    for varptr in roots
        mark (*varptr);
}
```

(Mark&) Sweep

```
sweep_all () {
    ptr = heap_start;
    do {
        if (ptr->marked)
            ptr->marked = False;
        else
            free_object (ptr);
    } while (ptr = next_object (ptr))
}
```

Stop&Copy

Alloue un nouveau tas *To* aussi grand que le tas actuel *From*

- *alloc_ptr* indique quelle partie de *To* est encore libre
- *scan_ptr* indique quelle partie de *To* est terminée

Copy: copier tous les objets accessibles de *From* dans *To*

- Commencer par les *racines*
- Copier les objets trouvés (cela fait avancer *alloc_ptr*)
- Placer un *forwarding pointer* de l'original vers la copie
- Suivre tous les pointeurs entre *scan_ptr* et *alloc_ptr*

Une fois terminé, on peut libérer *From* d'un seul coup d'un seul

(Stop&)Copy

```
copy (ptr) {
    if (ptr->forward = NULL) {
        for i = 0 to ptr->size
            alloc_ptr[i] = ptr[i];
        ptr->forward = alloc_ptr;
        alloc_ptr += ptr->size;
    }
    return ptr->forward;
}
```

Stop&Copy

```
stop&copy () {
   alloc_ptr = scan_ptr = alloc_new_heap ();
   for varptr in roots
      *varptr = copy (*varptr);
   while (scan_ptr < alloc_ptr) {
      for i = 0 to scan_ptr->size
         scan_ptr[i] = copy (scan_ptr[i]);
      scan_ptr += scan_ptr->size;
   }
   free_old_heap ();
}
```

Besoins du collecteur

Refcount	Mark&Sweep	Stop&Copy
champ *refcnt* (1-32bit)	champ *markbit* (1bit)	champ *forward* (1bit)

Obtenir la taille de n'importe quel objet

Savoir quels champs contiennent des pointeurs

inc/dec copie de ptr

Liste de toutes les *racines*

Accès au tas

Stop the world

Stop the world

mutateur: Le programme principal, qui fait le “travail utile”

collecteur: Le code qui s’occupe de récupérer la mémoire inutilisée

M&S et S&C sont tous deux des algorithmes *stop the world*:

Le *mutateur* doit être stoppé pendant que travaille le *collecteur*

Famille de GC

Un GC peut-être:

- incrémental: chaque phase de GC est découpée en petite tranches
- concurrent: *mutateur* et *collecteur* concurrents
- parallèle: *collecteur* divisés en plusieurs threads
- partitionné: le *tas* est divisé en sous-tas collectés indépendamment
- générationnel: GC partitionné en sous-tas ordonnés par âge
- distribué: GC partitionné sur des machines différentes

Finalization, pointeurs faibles

Les systèmes à base de GC offrent souvent la possibilité de détecter quand un objet est désalloué:

- Finalization: le programme spécifie qu'avant de désallouer l'objet X, il faut exécuter la fonction F
- Pointeur faible: pointeur qui n'empêche pas le GC de désallouer l'objet pointé. A chaque usage du pointeur, il faut vérifier s'il est encore vivant

Utilisés typiquement dans les caches, ou lors d'interaction avec des librairies externes que le GC ne comprend pas

Attention: la désallocation n'a pas forcément lieu

Désallocation manuelle

Nécessite des conventions et de la discipline

E.g. bibliothèque de table de hachage:

- Dans `hash_remove`, faut-il désallouer la valeur enlevée?
- Dans `hash_freetable`, faut-il aussi désallouer les valeurs?
- Dans `hash_copytable`, que faut-il faire des valeurs?
- Comment désallouer les valeurs?

Pas de réponses universellement idéales

Facile de faire des choix incohérents

Désallocation par ownership

1. *Un* des pointeurs de chaque objet est désigné *possesseur* (*owner*)
2. L'objet est désalloué lorsque son *possesseur* disparaît
3. $\Rightarrow$ Un pointeur n'est valide que si le *possesseur* est valide $\Leftarrow$

Si l'invariant ne peut pas être préservé:

- Faire des copies, chaque copie a son propre *possesseur*
- Ajouter un compteurs de références (compte nb de *possesseurs*)
- Ajouter un pointeur dont le seul rôle est d'être le *possesseur*

Le *possesseur* peut changer au cours du temps

Gestion mémoire semi-automatique

Gestion manuelle:

- Source intarissable de bugs graves
- Frein au déploiement de bibliothèques

Gestion automatique:

- Pauses indésirables pour usage temps-réel
- Parfois couteux, parfois inefficace
- Contraintes fortes sur l'ensemble du système

Permettre au programmeur de contrôler explicitement la désallocation

mais vérifier qu'il le fait correctement

Langage Rust

Sorte de mélange de Haskell et de C:

- C: Langage de bas niveau
- C: Gestion mémoire explicite
- C: Langage impératif, avec références explicites
- H: Encourage l'immutabilité
- H: Offre les *types algébriques*
- H: Typage statique fort
- H: Classes de type (appelées *Traits*)

[Note: Exemples tirés du manuel de Rust]

Syntaxe de Rust

Haskell	*Rust*
$f\ x\ y = e$	$\texttt{fn}\ f\ (x : \tau_1, y : \tau_2) \to \tau\ \{e\}$
$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$	$\texttt{let}\ x = e_1; e_2$
$\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$	$\texttt{if}\ e\ \texttt{then}\ \{e_1\}\ \texttt{else}\ \{e_2\}$
	$\texttt{while}\ e_1\ \{e_2\}$

Ownership sur les chaînes

```
fn main () {
    let s1 = String::from("hello");
    let s2 = s1;
    println!("s = {}", s2);
}
```

Après le `let s2 = s1;`, la variable s_1 n'est plus utilisable

La chaîne est désallouée à la fin de la fonction

Transfert d'ownership

```
fn prs (mys : String) { println!("mys = {}", mys); }
fn main () {
    let s = String::from("hello");
    prs(s);
    println!("s = {}", s);
}
```

Erreur! La variable s n'est plus valide après *prs*(s);

La chaîne est désallouée à la fin de *prs*

Valeurs sans ownership

Certains types n'ont pas besoin de gestion mémoire:

```
fn main () {
    let x_1 = 16;
    let x_2 = x_1;
    println!("x = {}", x_1 + x_2);
}
```

x_1 est encore valide après `let` $x_2 = x_1;$

La différence est que les types entiers implémentent le *traits Copy*

Références et prêts

Pour permettre accès sans transférer le *ownership*

```
fn prs (mys : &String) { println!("mys = {}", mys); }
fn main () {
    let s = String::from("hello");
    prs(&s);
    println!("s = {}", s);
}
```

L'opérateur & renvoie une *référence* à l'objet

Morceaux de tableaux

Au lieu de pointeurs au milieu des tableaux, Rust offre les *slices*

E.g. le type *&str* décrit une référence sur une sous-chaîne:

```
fn main () {
    let s : String = String::from("Hello");
    let sub : &str = &s[0..4];
    println!("sub = {}", sub);
}
```

Le type Option

Au lieu de *NULL*, utilise le type prédéfini *Option*

```
enum Option<T> { Some(T), None }
fn plus_one (x : Option<i32>) → Option<i32> {
  match x {
    None ⇒ None,
    Some(i) ⇒ Some(i + 1)
} }
```

Les noms viennent de ML, mais c'est sinon identique à *Maybe*

Contrôle des mutations

Rust impose un contrôle sur les modifications des objets

- Pas de modifications via une référence de type $\&T$
- Une référence de type $\texttt{\&mut}\ T$ permet les modifications
- Une seule référence de type $\texttt{\&mut}\ T$ à la fois
- Pas de $\texttt{\&mut}\ T$ et $\&T$ en même temps

$\Rightarrow$ Pas de problèmes causés par des *alias*

The mot-clé `mut`

Un objet de type *Vec<i32>*, est immuable

Cependant:

```
fn fill_vec (v1: Vec<i32>) -> Vec<i32> {
  let mut v2 = v1;
  v2.push(42);
  v2
}
```

Le `let` a changé le type en `mut` *Vec<i32>*

La méthode push prend un argument *self* de type `&mut` *Vec<i32>*

Éviter les références folles?

```
fn test() -> &String {
  let s = String::from("Hello");
  &s
}
fn main() { let r = test(); }
```

La chaîne s est désallouée à la fin de *test*

Le compilateur rejette le programme car $\&s$ survit son *owner*

On peut renvoyer s à la place (en ajustant le type de *test*)

Lifetimes

```
let s1 = String::from("Hello");
let result
{ let s2 = String::from("Goodbye");
  let l = longest(&s1, &s2);
  result = l;
```

Interdire *result* $= l;$ pour éviter une référence qui disparaît trop tard!

Il faut déterminer la *lifetime* de l:

- Comme celle de s_1 ou comme celle de s_2?

Lifetimes explicites

La fonction *longest* doit explicitement décrire ses *lifetimes*:

$$\texttt{fn}\ \textit{longest<'a>}\ (x : \textit{\&'a String}, y : \textit{\&'a String}) \rightarrow \textit{\&'a String}$$

$$\{\ \texttt{if}\ x.\textit{len}() > y.\textit{len}()\ \texttt{then}\ \{x\}\ \texttt{else}\ \{y\}\ \}$$

Ces annotations de *'a* indiquent:

> Le *lifetime* de la valeur de retour est égale
>
> au plus grand *lifetime* commun à ceux de x et y

conversion automatique de $\&$*'a* T en $\&$*'b* T (si *'b* $<$ *'a*)

Sorte de sous-typage

Élision des lifetimes

En fait, toutes les références ont un type de la forme $\&'a\,T$

C'est la gestion et l'inférence de ces *lifetimes* qui vérifie:

Un pointeur n'est valide que si le *possesseur* est valide

Références copiables vers un type avec une *lifetime* plus courte

Ça prend un peu de pratique, mais:

- La difficulté n'est pas artificielle: le même problème existe en C
- Le compilateur Rust nous aide plus que les core-dump de C

Rust

Premier langage populaire d'une longue ligne de recherche

Mélange inhabituel de fonctionalités de haut-niveau et de bas-niveau

Ownership utilisé pour:

- Gestion mémoire: but original principal
- Contrôler la mutabilité:
  - Éviter les problèmes liés aux alias
  - Plus important, éviter les conditions de courses