# Projet de Compilation Avancée
# Garbage Collector pour la Mini-ZAM

Darius Mercadier

24 février 2020

## Présentation du sujet

**Contexte :** L'année dernière, le projet de Compilation Avancée portait sur la réalisation d'une machine virtuelle (VM) pour le langage OCaml, nommée Mini-ZAM. Il s'agissait d'une version simplifiée de la machine virtuelle utilisée en pratique par le langage OCaml, la ZAM (*ZINC Abstract Machine*). La Mini-ZAM, tout comme la ZAM, est une machine à pile qui peut être vue, dans son noyau fonctionnel, comme une machine de Krivine avec une stratégie d'évaluation par appel par valeur (*call-by-value*). Pour exécuter tout programme OCaml, la ZAM interprète du *bytecode* OCaml représenté par 149 instructions différentes [1], tandis que la Mini-ZAM n'utilise que 27 instructions. Chaque instruction bytecode modifie l'état interne de la machine virtuelle, et l'évolution de cet état représente l'exécution du programme OCaml associé.

Si un langage contenant un garbage collector (GC) est utilisé pour implémenter la Mini-ZAM, il n'est pas nécessaire de doter celle-ci d'un GC, puisqu'elle peut utiliser celui du langage hôte. Cependant, une telle implémentation ne serait probablement pas très performante : il s'agirait d'une VM OCaml tournant au dessus de la VM du langage hôte tournant sur le CPU. Afin de supprimer la couche intermédiaire de la VM du langage hôte, il est nécessaire d'implémenter la Mini-ZAM en C, ou tout du moins dans un langage compilé vers de l'assembleur, et offrant un contrôle de bas niveau sur la machine. Il devient alors nécessaire d'implémenter un garbage collector pour cette machine virtuelle.

**Objectif :** En partant d'une implémentation de la Mini-ZAM en C ne contenant pas de garbage collector, nous allons dans un premier temps y rajouter un GC Stop & Copy. Dans un second temps, nous remplacerons ce GC par un Mark & Sweep, afin de gaspiller moins de mémoire. Dans un troisième temps, nous combinerons ces deux GC pour obtenir un GC générationnel.

**Rendu :** Vous devrez rendre, avant le **22/03/2019 à 23h59 (UTC+1)**, une archive au format `.tar.gz` contenant le code de votre implémentation de l'interprète *Mini-ZAM* (incluant au moins les sources ainsi que les tests), ainsi qu'un rapport (en français ou anglais selon votre préférence). Ce rapport (maximum 10 pages) devra décrire la structure générale du projet, vos choix d'implémentation, ainsi qu'une section détaillant quelles améliorations potentielles pourraient être faites à votre GC (vous pourrez vous inspirer de la section "Pour aller plus loin"). Tout warning durant la compilation devra être solidement justifié dans ce rapport. Le rendu du projet se fera par courrier électronique, aux adresses `darius.mercadier@lip6.fr` et `emmanuel.chailloux@lip6.fr`.

Le sujet du mail aura la forme suivante :

`[4I504] Rendu Projet - <nom1> <nom2>`

(en remplaçant `<nom1> <nom2>` par les noms des membres de votre binôme)
Tout dépassement de la date limite entraînera des pénalités.

1. dans la version actuelle d'OCaml - la 4.07

# 1 Introduction

Cette introduction vise à vous familiariser avec la Mini-ZAM. Commencez par lire la description de la Mini-ZAM est disponible à l'adresse suivante : `https://dariusmercadier.com/assets/documents/projet-minizam.pdf` (il s'agit du sujet du projet de l'année dernière). Toutes les extensions mentionnées sur ce document (appels terminaux, blocs de valeurs et exceptions) sont implémentées dans la Mini-ZAM qui vous est fournie à l'adresse suivante : `https://dariusmercadier.com/assets/documents/minizam-etu.tgz`.

L'implémentation de la Mini-ZAM vise à être au plus proche de l'implémentation de la ZAM complète, tout en gardant le plus de simplicité possible.

**bytecode.** La Mini-ZAM prend en entrée un fichier de bytecode au format spécifié par le projet de l'an dernier. Lors du parsing, les primitives (`+`, `-`, `>=`, *etc.*) sont remplacées par des entiers (pour ne pas avoir à manipuler de chaînes de caractères lors de l'interprétation), les labels sont supprimés et les adresses des sauts sont calculées : `BRANCH L1` sera remplacé par `BRANCH addr` où `addr` est l'adresse absolue correspondant au label `L1`. Le bytecode manipulé par la VM en interne est donc un simple tableau d'entiers 64-bits. Par exemple, le bytecode suivant :

```
    CONST 1
    BRANCHIFNOT L2
    CONST 40
    PUSH
    CONST 2
    PRIM +
    BRANCH L1
L2: CONST 42
L1: STOP
```

Qui correspond au code OCaml `if true then 40 + 2 else 42`, sera convertis en le tableau suivant :

```
[CONST, 1, BRANCHIFNOT, 13, CONST, 40, PUSH, CONST, 2,
 PRIM, PLUS, BRANCH, 15, CONST, 42, STOP]
```

Où `CONST`, `BRANCHIFNOT`, `PLUS`, *etc.* sont définit dans des `enum`s et sont donc des entiers.

**mlvalues.** Toutes les données manipulées par la Mini-ZAM sont de type `mlvalue` (défini dans `mlvalues.h`). Une `mlvalue` est soit un pointeur soit un entier sur 63 bits. Le bit de poids faible différencie les deux cas : si une valeur se finit par un 1, alors il s'agit d'un entier, tandis que si elle se finit par 0, alors il s'agit d'un pointeur. Les macros `Val_long`, `Long_Val`, `Val_ptr` et `Ptr_val` permettent d'obtenir un entier ou un pointeur à partir d'une `mlvalue` et inversement.

**blocks.** Une `mlvalue` qui n'est pas entier est nécessairement un pointeur sur un bloc. Un bloc est un tableau de `mlvalue`s ainsi qu'un header de type `header_t` contenant sa taille, son tag, ainsi que 2 bits réservés pour le garbage collector (inutilisés présentement, mais qui vous serviront plus tard dans le sujet). Les tags possibles sont `ENV_T`, représentant un environnement, `CLOSURE_T` représentant une fermeture, et `BLOCK_T` représentant n'importe quel autre bloc. Un pointeur vers un bloc pointe toujours vers le premier élément du bloc, et non vers le header : le header d'un bloc `b` est à l'adresse `b-1` (cela implique que qu'un header (`header_t`) doit fait la même taille qu'une `mlvalue`). Vous pouvez constater qu'en effet, lorsqu'un bloc est alloué (`make_block` dans `mlvalues.c`), un pointeur vers son premier élément est renvoyé et non vers le header. Les macros `Size` et `Tag` permettent d'accéder à la taille et au tag d'un bloc sans passer explicitement par son header. La macro `Field` permet d'accéder à un élément d'un bloc.

**Allocations.** Dans la version de la Mini-ZAM qui vous est fournie, toutes les allocations mémoires sont faites par les fonctions `make_empty_block`, `make_block` et `make_closure` ; qui appellent toutes `caml_alloc`. La mémoire n'est jamais libérée.

**Variables globales.** Un singleton de type `caml_domain_state` nommé `Caml_state` est déclaré dans le fichier `domain_state.h` et initialisé dans `domain.c` au lancement de la VM. Cette variable contient les données globales du programme. Dans la version qui vous est fourni, il ne contient que la pile. Lorsque vous aurez besoin de variables globales (par exemple, semi-spaces pour le Stop & Copy, freelists et pages pour le Mark & Sweep), vous pourrez les ajouter à cette structure. Similairement, la configuration de votre programme (taille de la pile, taille du tas, etc.) sera définit dans le fichier `config.h`.

**Compilation et testing.** Un makefile vous est fournis pour compiler la Mini-ZAM. Additionnellement, le script `run_tests.pl` (qui est également lançable par la commande `make test`) lance la Mini-ZAM sur un ensemble de fichiers de tests présents dans le dossier `tests`. Pour lancer manuellement la minizam, la syntaxe est `minizam <fichier_de_bytecode> [-res]`. L'argument `-res` est optionnel. Si il est fournis, il doit être après le fichier de bytecode dans la liste des arguments, et il indique à la Mini-ZAM d'afficher la dernière valeur calculée par le programme. Parmis les fichiers de tests fournis, le dossiers `tests/bench` contient des tests manipulant beaucoup de mémoire et qui mettront vos garbage collectors à l'épreuve.

## Bytecode de la Mini-ZAM

Afin de prendre en main le bytecode de la Mini-ZAM, **traduisez manuellement le programme suivant en bytecode** :

```
let rec f n =
  if n > 1000 then 0
  else
   (1 + (f (n + 3)))
let _ = f 3
```

Ce programme calcule la somme de tous les multiples de 3 inférieurs à 1000.

Vous nommerez ce fichier `sum_3.txt`, et vous le placerez dans un dossier nommé `test/perso`. Vous pourrez placer dans ce dossier tous les fichiers de bytecode que vous écrirez durant ce projet.

## Implémentation de la Mini-ZAM

La Mini-ZAM qui vous est fournis souffre de quelques défauts d'implémentation. En particulier, les implémentations des bytecodes `APPLY` et `APPTERM` (dans `interp.c`) ont deux défauts : premièrement, elles font appel à `malloc`, et deuxièmement, elles copient deux fois les arguments de la fonction appelée. Ces deux facteurs impactent fortement les performances des programmes faisant beaucoup d'appels de fonctions ; ce qui représente la quasi-totalité des programmes OCaml. **Réécrivez APPLY et APPTERM sans utiliser de malloc ni de mémoire temporaire**.

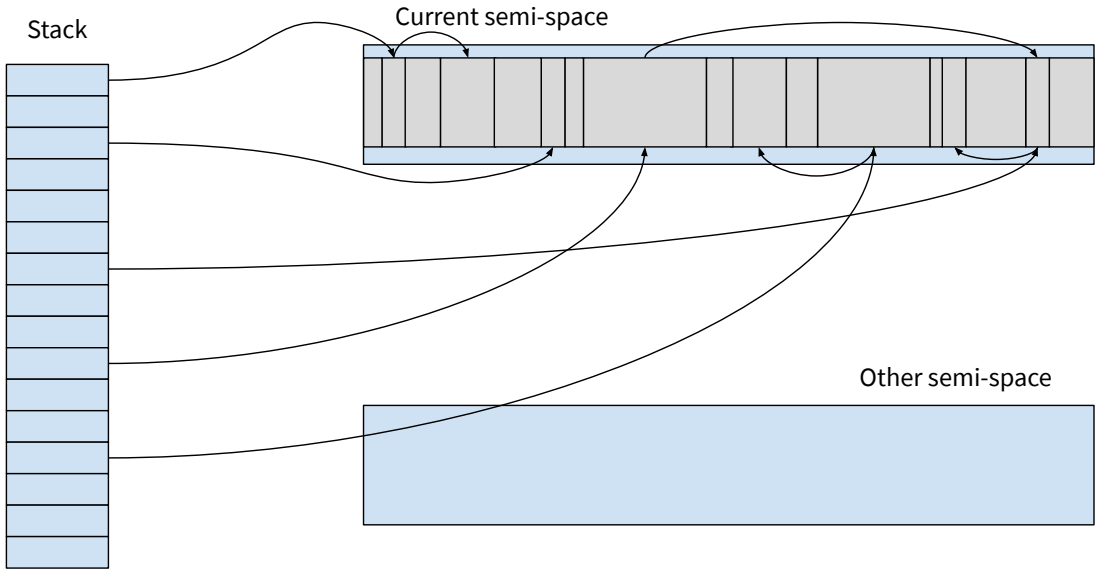
## Fuite mémoires & Valgrind

Lors de ce projet, vous vous retrouverez probablement plusieurs fois confrontés à des "segmentation faults", ou autres problèmes de corruption mémoire, ainsi que des fuites mémoire. Un outil essentiel afin de traquer l'origine de ces erreurs est `valgrind`.

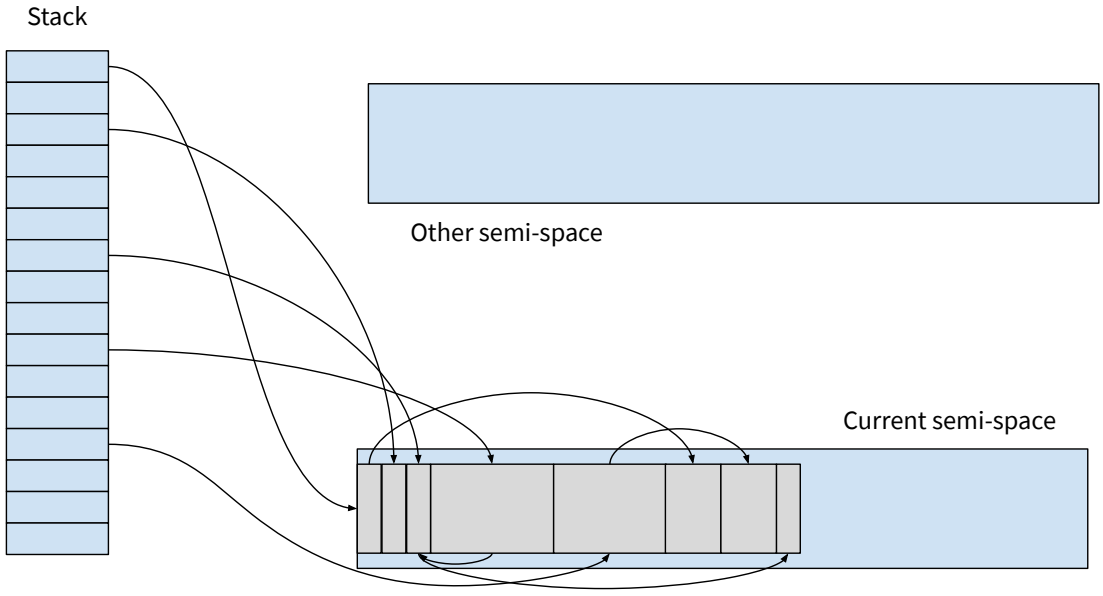
La Mini-ZAM qui vous est fournis contient deux fuites mémoire principales. La première est liée à l'absence de GC, et peut être ignorée pour le moment. La seconde vient du parseur, qui a été codé avec peu d'attention. **Traquez et corrigez cette fuite mémoire**.

## 2 Stop & Copy

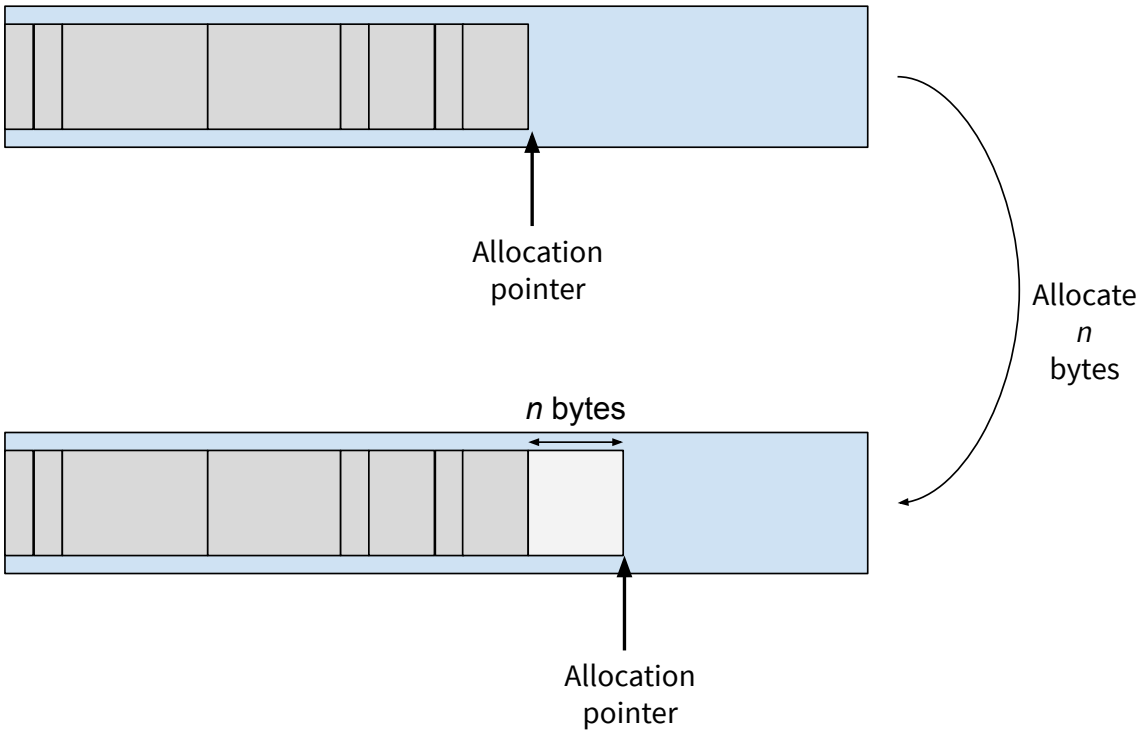
Un garbage collector Stop & Copy utilise deux tas (aussi appelés *semi-spaces*). Toutes les allocations se font dans un seul des deux tas. Une fois que celui-ci est plein, le garbage collector se déclenche : à partir des racines (typiquement la pile, les registres et les variables globales), il trouve tous les objets vivants (*ie.* accessibles depuis le programme en train d'être executé) dans le tas :




Ces objets sont ensuite tous copiés dans le second tas :




Les prochaines allocations seront faites dans ce second tas. Une fois que ce second tas sera rempli, le GC se re-déclenchera et déplacera les objets vivant vers le premier tas. A tout instant du programme (hors GC), uniquement un tas contient des objets, tandis que l'autre est vide. Cet algorithme gâche donc une partie de la mémoire. En contrepartie, les allocations sont très rapides : pour allouer un bloc, il suffit d'incrémenter un pointeur (*bump-pointer* en anglais) dans le tas courant :




Une fois que ce pointeur atteint le bout du tas, le garbage collector est déclenché et déplace les objets vivants vers l'autre tas. Lorsque, à la suite du GC, le tas courant est trop rempli (resp. trop *peu* rempli), il doit être agrandi (resp. réduit).

Un moyen classique d'implémenter un Stop & Copy est l'algorithme de Cheney. Dans cet algorithme, le semi-space que l'on veut collecter est appelé *from-space*, tandis que le semi-space de destination pour les objets vivants est appelé *to-space*. Cet algorithme agit en deux étapes :

1. Objets sur la stack : la stack (et les racines en générales) est parcourue à la recherche de références. Lorsqu'une référence est trouvée, l'une des deux actions suivantes est appliquée :
   (a) Si l'objet n'a pas encore été déplacé dans to-space (son tag n'est pas `FWD_PTR_T`), créer une copie identique dans to-space, changer le tag de sa version dans from-space en `FWD_PTR_T`, et ajouter un pointeur vers ce nouvel objet dans le premier champ de version de from-space (aussi appelé *forwarding pointer*). Modifier ensuite la référence sur la stack pour pointer vers le nouvel objet dans to-space.
   (b) Si l'objet a déjà été déplacé dans to-space, modifier la référence sur la stack avec la valeur du forwarding pointer.
2. Objets dans to-space : le GC parcourt toutes les références contenues dans les objets de to-space, et effectue une des deux actions (a) et (b) ci-dessus. Cette étape peut donc entraîner le déplacement d'objets de from-space vers to-space, sur lesquels elle sera appliquée également.

**Ajouter à la Mini-ZAM un GC Stop & Copy utilisant l'algorithme de Cheney**.

Votre Stop & Copy aura les caractéristiques suivantes :
— Chaque semi-space aura une taille initiale de 32Ko.
— Les semi-space seront agrandis de 50% ($\times 1.5$) lorsqu'ils seront remplis à plus de 50% après la collection du GC.
— Les semi-space seront rapetissés de 50% ($\div 1.5$) lorsqu'ils seront remplis à moins de 25% après la collection du GC.

Nous vous recommandons de suivre les étapes suivantes pour l'implémentation :

1. Au début de l'éxecution de la VM, allouez deux semi-spaces, et stockez leurs pointeurs dans `Caml_state`.

2. Remplacez la fonction `make_block` par une macro qui alloue dans le semi-space courant. Ne vous préoccupez pas du GC pour le moment : la VM crashera une fois que le semi-space courant est rempli.
3. Assurez vous qu'une partie des tests fonctionne, et que les tests non-fonctionnels utilisent en effet trop de mémoire.
4. Identifiez les racines, et trouvez un mécanisme efficace pour que le GC y ait accès.
5. Implémentez le Stop & Copy sans redimensionnement des semi-spaces.
6. Proposez et implémentez un mécanisme de redimensionnement des semi-spaces.

# 3 Mark & Sweep

Le Stop & Copy gaspille beaucoup de mémoire. Afin de pallier à cela, nous allons nous intéresser à une autre approche : le Mark & Sweep.

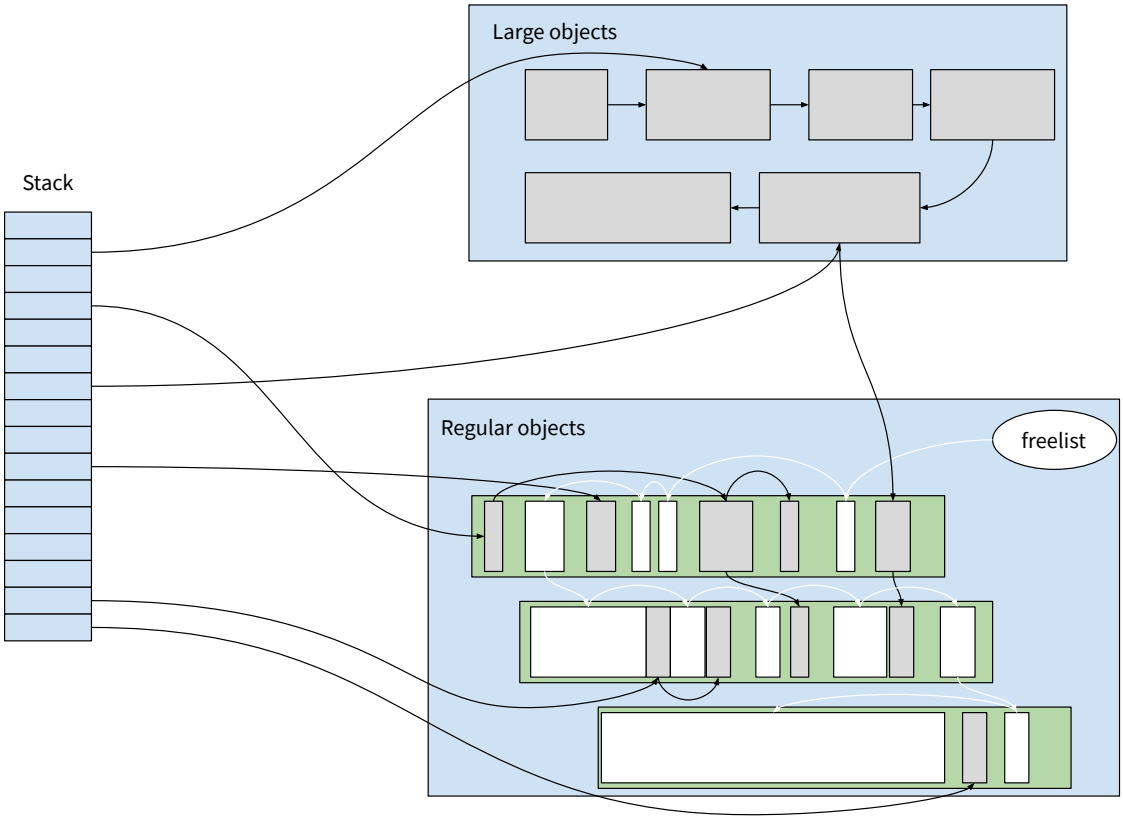
Un Mark & Sweep fonctionne en deux étapes : la première consiste à parcourir et "marquer" toute la mémoire accessible à partir des racines. La seconde consiste à parcourir l'intégralité de la mémoire : chaque bloc qui n'est pas marqué est inaccessible (depuis le programme) et est donc libéré (*swept*). En pratique, les blocs libres sont ajoutés à une liste chaînée appelée *freelist*. Lorsque qu'une allocation est effectuée dans le programme, la freelist est parcourue à la recherche d'un bloc libre, qui sera retiré de la freelist et donné au programme.

La question se pose de comment bootstrapper la freelist : d'où vient la mémoire initiale du programme ? Faire un gros malloc comme le Stop & Copy fait pour son tas ne serait pas idéal : d'une part, cela occuperait plus de mémoire que nécessaire (n'oubliez pas qu'un des avantages du Mark & Sweep est de réduire la quantité de mémoire utilisée), et, d'autre part, les redimensionnements du tas seraient compliqués (puisqu'il faudrait modifier tous les pointeurs du programme). De plus, il suffirait d'un objet vivant au début et un objet vivant à la fin du tas pour empêcher celui-ci d'être réduit, même si tout le milieu est inutilisé.

Une meilleure approche consiste à utiliser un espace paginé : le tas est divisé en pages (qui ne sont pas nécessairement des pages de l'OS). Lorsqu'une page est vide, celle-ci peut être rendue au système (en faisant un `free`). Lorsque la freelist est vide et qu'une allocation est demandée, une page supplémentaire est allouée et ajoutée entière dans la freelist. Cette approche résout également le problème de "un objet vivant au début et un objet vivant à la fin du tas l'empêchent d'être réduit" : un objet peut garder une page vivante au maximum, et non l'ensemble du tas. La phase de Sweep doit désormais parcourir chaque page une par une et ajouter les blocs vides à la freelist. Il est nécessaire de faire du coalescing : deux blocs vides contigus doit être fusionés en un seul bloc. Si une page contient un seul bloc vide de la taille de la page, c'est qu'elle est vide et elle peut donc être libérée (`free`).

Lorsque l'on utilise un espace paginé, les objets dont la taille est supérieure à la taille d'une page doivent être alloués (`malloc`) individuellement. Il est raisonnable de considérer qu'il existe peu de gros objets, et il est donc acceptable d'utiliser une liste chaînée gardant la trace de les gros objets ainsi alloués. Lors de la collection, la phase de Mark reste inchangée. La phase de Sweep quant à elle parcourra additionnellement la liste chaînée de gros objets : les objets non-marqués sont inaccessibles et peuvent donc être libérés (`free`). En pratique, il est courant de considérer que des objets faisant plus de la moitié d'une page sont gros.

Au final, la mémoire de la Mini-ZAM une fois le Mark & Sweep implémenté peut être représentée par le schéma suivant :




**Implémentez un GC Mark & Sweep pour la Mini-ZAM**. Vous utiliserez des `#ifdef` pour choisir quel GC est utilisé par la Mini-ZAM : si `_STOP_n_COPY` est définit, le Stop & Copy sera utilisé, tandis que si `_MARK_n_SWEEP` est définit, le Mark & Sweep sera utilisé.

Votre Mark & Sweep devra avoir les caractéristiques suivantes :
— L'espace sera divisé en pages de 64Ko.
— Les gros objets, de plus plus de 32Ko, seront stockés dans une liste chaînée.
— Une unique freelist sera utilisée pour garder la trace des blocs disponibles.
— Une stratégie de first-fit sera utilisée pour allouer des blocs dans la freelist.
— Un GC sera déclenché lorsque la quantité de mémoire utilisé aura augmenté de 50% ($\times 1.5$) par rapport au GC précédent.

Nous vous recommandons de suivre les étapes suivantes :
— Implémentez la machinerie nécessaire pour stocker les gros objets.
— Considérez que tous les objets sont gros, et modifier votre allocateur en conséquent.
— Implémentez la phase de Mark pour les gros objets.
— Implémentez la phase de Sweep pour les gros objets.
— Assurez vous que tous les tests sont fonctionnels.
— Implémentez un allocateur utilisant une freelist dans un espace paginé.
— Modifiez la phase de Mark si nécessaire pour qu'elle marque tous les objets vivants.
— Implémentez la phase de Sweep pour l'espace paginé.

En bonus, vous pourrez effectuer un ou plusieurs des points ci-dessous :
— Utilisez plusieurs freelists chacune contenant des objets de taille différente.
— Lorsqu'une allocation de taille $n$ est effecutés, utilisez les freelists pour obtenir une zone mémoire plus grosse que demandée, et utilisez du bump-pointer dans cette zone mémoire pour les allocations subséquentes.
— N'utilisez pas de récursion dans la phase de Mark.

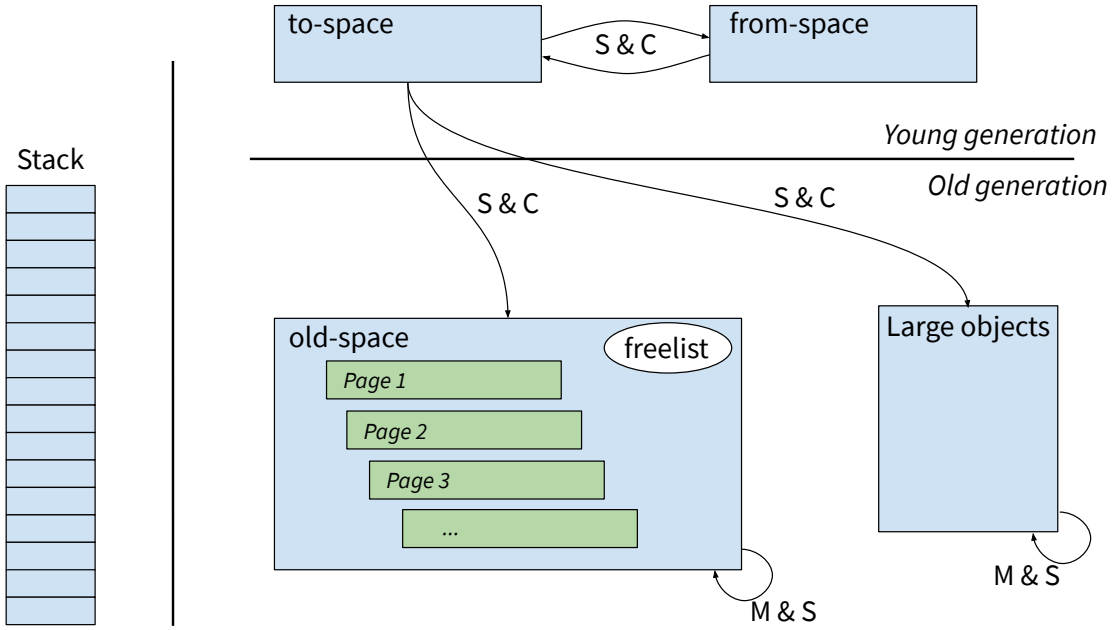
# 4 GC générationnel

En pratique, la plupart des objets alloués sont locaux et/ou temporaires et meurent vite. Ce phénomène est connu sous le nom d'*hypothèse générationnelle*. Chaque cycle de garbage collection a donc tendance à vider la majorité de la mémoire. Corollairement, les objets qui survivent un cycle de collection ont de grandes chances de survivre les quelques cycles suivants.

Cette observation est la base des garbage collectors *générationnels*. Le principe est de séparer le tas en 2 (ou plus) générations : les objets jeunes d'un coté, et les objets vieux d'un autre. Il est alors possible d'effectuer certains GC uniquement sur les objets jeunes et non sur l'ensemble du tas. Les objets qui survivent une ou deux collections sont déplacés vers le tas des vieux objets.

La majorité des GC générationnels utilisent deux algorithmes différents pour leurs générations : un Stop & Copy pour la jeune génération –ce qui offre des allocations très rapides mais gaspille de la mémoire– et un Mark & Sweep [2] pour la vieille génération –minimisant la consommation mémoire au détriment d'un léger overhead lors des allocations–.

Le Stop & Copy déplace les objets du from-space de la jeune génération vers le to-space, et promeut les objets ayant survécus deux collections vers la vieille génération. Si ces objets sont gros, ils seront alloués dans l'espace des gros objets ; sinon, ils seront alloués dans l'espace paginé standard. Le Mark & Sweep gèrera ces zones mémoires en libérant les gros objets inatteignables, et en remplissant les freelists :

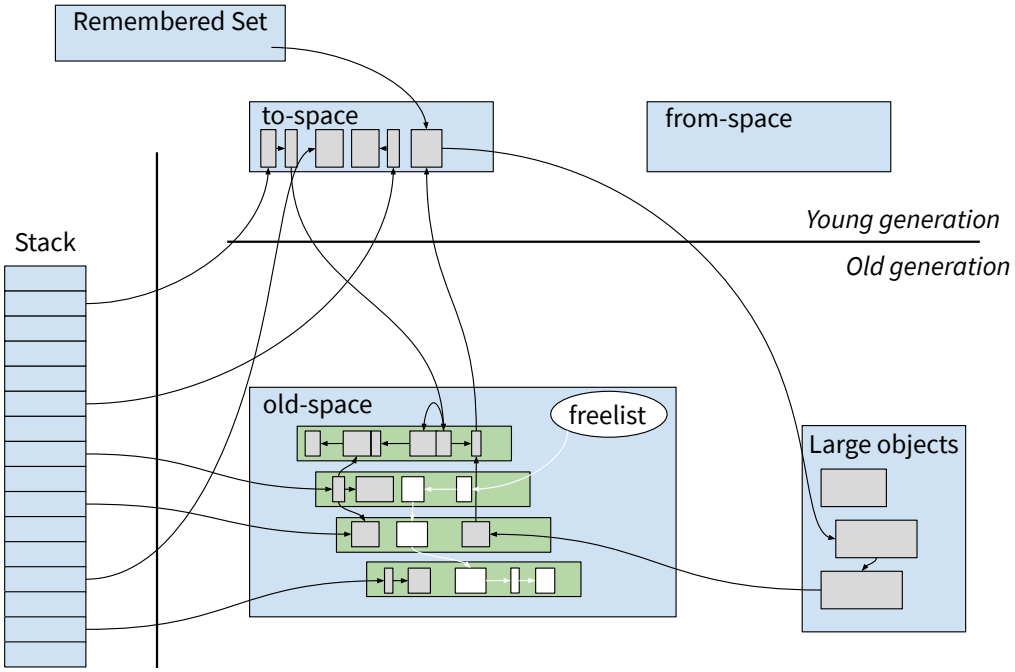



Les pointeurs inter-générations (jeune-vers-vieille, ou vieille-vers-jeune) complexifie la tâche : le Stop & Copy de la jeune génération doit être rapide et ne doit donc pas parcourir l'ensemble de la mémoire pour trouver les blocs vivants dans la jeune génération. Afin de surmonter ce problème, il est nécessaire de stocker les pointeurs de la vieille génération vers la jeune génération (aussi appelés *old-to-new*) dans ce qu'on appelle un *remembered set*. Nous vous proposons d'utiliser un tableau auto-redimensionnable pour stocker le remembered set. Il vous faudra donc modifier les instructions de la VM pouvant créer des pointeurs en rajoutant quelques instructions appelées *write-barrier*, afin de vérifier si une écriture crée un pointeur old-to-new.

De plus, lorsque le Stop & Copy trouve des objets vivants, il est nécessaire de pouvoir différencier entre un objet de la jeune génération (qu'il doit déplacer vers to-space) et un objet de la vieille génération (qu'il doit ignorer). Ceci est relativement facile puisque nous disposons à tout instant du pointeur de début et de fin du semi-space courant de la jeune génération. Une simple comparaison avec ces deux pointeurs permet donc de déterminer si un objet est jeune ou vieux.

2. Ou bien un Mark & Compact, présenté dans la Section "Pour aller plus loin"

Le schéma suivant résume l'organisation de la mémoire (en ne représentant que les objets accessibles dans un soucis de clareté) :




Fortuitement, vous avez déjà implémenté un Stop & Copy et un Mark & Sweep dans la Mini-ZAM. **Connectez ces 2 GC afin d'obtenir un garbage collector générationnel**. Nous vous recommandons de travailler sur une copie des sources précédentes afin de garder une trace de votre Stop & Copy et Mark & Sweep non-générationnels pour le rendu.

Votre GC aura les fonctionnalités suivantes :
— Stop & Copy pour la jeune génération.
— Promotion de la jeune génération vers la vieille après avoir survécu 2 collections.
— Remembered set old-to-new stocké sous forme de tableau auto-redimensionnable.
— Mark & Sweep pour la vieille génération.

# 5 Pour aller plus loin

**Résumé.** Vous avez implémenté au cours de ce projet un garbage collector générationnel utilisant un Stop & Copy pour la jeune génération et un Mark & Sweep pour la vieille génération. La plupart des GC de productions utilisent une configuration similaire ; parfois en remplaçant le Mark & Sweep par un Mark & Compact (pour citer quelques exemples : OCaml, V8 (JavaScript), Java...).

**Mark & Compact.** Un Mark & Compact est basé sur le même principe qu'un Mark & Sweep : les objets vivants sont marqués en parcourant les blocs accessibles. Ensuite, les blocs vivants sont déplacés afin de compacter la mémoire. Cela offre une amélioration en terme de consommation mémoire par rapport au Mark & Sweep, mais augmente le temps de la collection (car il faut déplacer les objets et pas uniquement les parcourir).

**Parallélisation.** Une distinction est faite entre un GC *concurrent*, qui s'exécute en même temps que le programme principal, et un GC *parallel*, qui pause le programme, mais utilise plusieurs threads. La plupart des GC haute-performance sont parallélisés et/ou concurrents.

Il est fréquent que le Stop & Copy soit *stop-the-world*, c'est à dire qu'il mette le programme en pause. En revanche, les Mark & Sweep sont souvent concurrents. Cela implique que la VM doit être prudente lors des écritures : si une écriture est faite lors de la phase de Mark, il faut que le GC en tienne compte afin de ne pas risquer de considérer qu'un objet vivant est mort. Ceci est fait à l'aide d'une write-barrier, typiquement active uniquement lorsque le GC est actif, afin de ne pas trop impacter

les performances. Tandis que les GC non-concurrents stop-the-world peuvent utiliser uniquement 2 couleurs (*eg*, blanc = vivant, noir = mort) pour marquer les objets, les GC concurrents utilisent généralement 3 couleurs, afin de tenir compte des écritures qui peuvent se produire lors du marking.

Les phases d'un GC qui déplacent des objets (la compaction du Mark & Compact par exemple) sont plus rarement effectués de manière concurrente. En effet, si le Mark & Compact était exécuté entièrement de manière concurrente, cela requièrerait de modifier la VM pour, à chaque lecture en mémoire, vérifier si l'objet à lire est bien là où il est supposé être, ou bien si il a été déplacé. Ce mécanisme s'appelle une *read-barrier* et est souvent considéré comme trop coûteux pour justifier de faire la compaction de manière concurrente.

Parce que la compaction est souvent stop-the-world, les GC Mark & Compact sont en réalité souvent (dans mes termes) des GC Mark & Sweep & parfois-Compact ou bien des GC Mark & Sweep & Compact-un-peu : soit la phase de compaction n'est pas faite à toutes les itérations du GC, soit elle ne compacte qu'une partie du tas.

Qu'ils soient concurrents ou non, les GC sont souvent parallélisés : plusieurs threads peuvent effectuer le Marking, le Sweeping et la compaction en parallèle. C'est également pour cette raison qu'il est pratique que le tas soit organisé en pages : on peut réduire les problèmes de concurrence entre les threads en attribuant une page à marquer (ou à sweeper) à chaque thread. Notez d'ailleurs à cette occasion que nous avons utilisé un espace paginé uniquement pour la vieille génération dans ce projet, mais qu'il est possible d'effectuer la même chose pour la jeune génération (le Stop & Copy n'est pas un obstacle).

**Freelists.** Ne pas utiliser de compaction du tout à tendance à causer une fragmentation du tas : il suffit d'un objet vivant sur une page pour empêcher celle-ci d'être libérée. Pour parer à ce cas, il est possible d'utiliser des pages de petites tailles (par exemple, Go n'utilise pas de compaction et utilise des pages de 8Ko). Il faut aussi utiliser plusieurs freelists très précises, afin que lorsqu'un objet est alloué, très peu de mémoire soit gâchée.

Dans un GC disposant de compaction, les allocations dans les freelists peuvent être accélérées en combinant les freelists à du bump-pointing : lorsqu'une requête pour $n$ octets est faite dans les freelists, on cherche un bloc de taille $m$ tel que $m > n$, et on utilise ce bloc pour faire du bump-pointing : les allocations des $m - n$ octets suivants pourront être fait avec du bump-pointing.

**Localité dans les caches.** Les GC Stop & Copy ainsi que les Mark & Compact déplacent des objets. Lors de leur conception, il est indispensable de prendre en compte le cache pour choisir la stratégie à adopter pour déplacer les objets. En effet, il est préférable que des objets manipulés en même temps soient proche dans le cache. Cela se traduit par exemple assez concrétement dans le Stop & Copy que vous avec implémenté : votre version déplace d'abord tous les objets pointés par la pile, puis dans un second temps, tous les objets pointés par ces objets (c'est donc un parcours en largeur de la mémoire accessible). Une autre solution aurait été de faire un parcours en profondeur : déplacer un objet pointé par la pile, puis immédiatement les objets pointés par celui-ci avant de parcourir le reste de la pile.

La localité dans les caches est également un aspect à prendre en compte lors du choix du garbage collector : le Stop & Copy fait toutes ses allocations en bump-pointer (de manière contiguë donc, ce qui est bien pour le cache), tandis que les allocations dans les freelists du Mark & Sweep sont typiquement dispersées dans l'intégralité de la mémoire (ce qui n'est pas bien pour le cache).

**GC précis *vs* GC conservatifs.** La plupart des GC sont *précis* : ils disposent d'un moyen de savoir si un objet est un pointeur ou une autre donnée. Cependant, certains GC n'ont pas cette information. C'est le cas par exemple du GC de Boehm pour le langage C : il n'est pas toujours possible de différencier un entier d'un pointeur sur la pile. Ces GC doivent donc être *conservatifs* : tout ce qui pourrait être un pointeur est considéré comme un pointeur. Ainsi, un entier qui, une fois converti en pointeur, pointe vers une zone mémoire allouée, sera considéré comme un pointeur, et le GC considérera que les données pointées sont vivantes. Cela implique qu'un GC conservatif ne peut pas déplacer d'objets : si il considère qu'une donnée est un pointeur alors qu'il s'agit d'un entier, et qu'il déplace les données pointées, il doit alors modifier le pointeur, et aura alors modifié un entier manipulé par le programme.