Rapport Lisp

Paul Bastide, Clara Bégué
Alex Coudray et Lauric Desauw*

5 Mars 2019

*DIT, ENS Rennes

Table des matières

Introduction

Lisp est un langage interprété basé sur le traitement de listes. Ici, les opérations sont notées de manière préfixe pour faciliter l'interprétation. Dans le cadre de ce projet, nous avons réalisé un interpréteur dynamico-statique Lisp en C++. Ainsi, les fonctions peuvent être définies dans l'environnement courant (dynamiques) ou dans une clôture (statiques). De plus, nous avons implémenté un *garbage collector* qui limite les fuites mémoires ainsi qu'une méthode permettant de charger un fichier dans l'interpréteur.

1 Concepts de Lisp

1.1 Principe

1.1.1 Objets Lisp

Les objets Lisp sont de cinq types de base : nombre entier, chaîne de caractère, symbole, qui correspond à des noms de variables et de fonctions, booléen, associé aux symboles `#t` et `#f`, et paire, composée de deux objets Lisp. On peut donc définir des listes d'objets Lisp de cette façon.

Il est possible d'appliquer des opérations de base sur ces objets comme des opérations arithmétiques (addition, multiplication et soustraction sur les objets nombres), la concaténation de deux chaînes de caractères et la manipulation de listes (avec les fonctions `car`, `cdr`, `cons` et `rplacd`). De plus, nous avons implémenter des fonctions permettant de tester le type d'un objet. Ces fonctions et les opérations de base sont appelées *subroutines* car leur exécution ne dépend que des paramètres et pas de l'environnement courant. Nous reviendrons plus tard sur leur implémentation et la façon dont elles sont appliquées.

De plus, il est possible de définir ses propres variables et fonctions à l'aide de la commande `define`. Par exemple, on peut définir une variable `a` instanciée à 1 de la façon suivante : `(define a 1)`. L'information `a = 1` est stockée dans une liste Lisp appelée *environnement*.

Lorsque le symbole `a` est trouvé dans une commande Lisp, il faut aller chercher dans l'*environnement* la valeur de `a` la plus récente. Si `a`, initialement initiée à 1, est redéfini par la suite à 2 (avec `(define a 2)`), la commande `(+ a 1)` devra renvoyer $2 + 1 = 3$. Notons que l'information `a = 1` n'est pas écrasée par la nouvelle définition de `a` : `a = 2` est simplement enfilée en tête de l'environnement lors de l'évaluation de l'instruction `(define a 2)`. De plus, nous avons choisi d'interdire la redéfinition des *subroutine* pour des raisons de sécurité et de lisibilité évidentes. Par exemple, l'appel à `(define + 1)` lève une exception car `+` est une *subroutine*.

1.1.2 Eval/Apply

Lorsque l'utilisateur rentre une expression dans l'interprète on aimerait calculer la valeur de cette expression pour la retourner à l'utilisateur. Pour cela il faut passer l'expression dans la boucle *Eval/Apply*.

`Object eval(Object obj, Env env)` commence par tester le type de l'expression qu'on lui donne en argument. Si elle est de type nombre, booléen ou chaîne de caractéres alors *Eval* renvoie l'expression car elle contient sa valeur (il suffira de changer son type). Sinon l'expression est une paire. On regarde si le premier terme est une structure de contrôle, dans ce cas on applique la fonction associée à la queue de la paire. Si le premier terme n'est pas une structure de contrôle c'est un symbole, donc on appelera *Eval* sur la chaque éléments de la queue de la paire et on donnera le tout à *Apply*.

`Object apply (Objec func, Object lvals, Env env)` regarde si `func` est reconnu comme une *subroutine* et dans ce cas on applique la fonction associée à `lvals` et on retourne le résultat. Si `func` n'est pas une *subroutine* on regarde si c'est un symbole (il pourrait donc avoir une définition dans l'environnement) et dans ce cas on évalue la définition associée et on la renvoie dans `apply`. Si func n'est ni une *subroutine* ni un symbole alors c'est une liste et cela doit être une lambda expression, on verifie que son premier terme est bien `"lambda"`, on créer un environnement local contenant les paramétres li à éeur valeurlpuis on applique ppply sur le corps de la lambda expression avec le nouvel environnement.

Eval et *Apply* vont donc s'appeller mutuellement jusqu'à avoir calculer la valeur de l'expression grâce à l'environnement et aux *subroutines*.

1.2 Implémentation machines des objets

1.2.1 Objets Lisp et API

Les objets Lisp sont stockés sous la forme d'éléments d'une classe `Cell`. Celle-ci possède un attribut statique `number_of_cells_parent` qui compte les cellules instanciées sans distinction de type. De cette classe mère, héritent des classes filles spécifique à chacun des types d'objets Lisp. De même, les classes filles possèdent un attribut statique qui compte le nombre de cellules de chaque type. Ainsi, on peut, en activant le mode *stats* avec `(stats #t)`, connaître la répartition des cellules en fonction de leur type.

Afin de gérer les fonctions définies statiquement (*ie* dans leur environnement de définition et pas dans l'environnement courant), un attribut `closure` ainsi que les fonctions `set_closure`, `get_closure` et `is_static` ont été rajoutées à la classe `Cell`.

Les booléens ont été définis dans l'API comme des objets de type `Cell_symbol` avec comme valeur `#t` et `#f`. Ainsi les booléens et la liste vide sont vus comme des constantes de l'API.

Les fonctions permettant de manipuler les cellules sont implémentées dans l'API. Notons que les paramètres des fonctions sont de type `Object = Cell*`, les objets doivent donc être transformés en objets C++. Pour cela, ils sont castés dynamiquement. Avant d'appeler les fonctions de l'API dans le module `Eval`, nous avons vérifié que les objets étaient du bon type à l'aide des fonctions `nullp`, `numberp`, `boolp` ...

1.2.2 Environnement

Pour l'implémentation futur d'un interprète statique (section 2) il est nécessaire qu'un environnement soit un objet Lisp. Ainsi sa construction se fait en deux étapes. Il est dans un

premier temps nécessaire d'implémenter des *bindings* pour ensuite construire un environnement à proprement parlé. Le but d'un *binding* est de lié un symbole à une valeur, soit en Lisp : lier un objet symbole à un objet quelconque. Pour se faire on crée simplement une liste qui contient le symbole et sa valeur. Ce qui se traduit schématiquement en Lisp par :

`pair_object (symbol_object , object)`

Une fois ceci fait, l'environnement se traduit naturellement comme un `pair_object` qui contient l'ensemble des *bindings*.

1.3 Fonctions

Le terme "fonction" regroupe ici deux notions qui peuvent sembler similaire pour l'utilisateur mais dont l'implémentation diffère. D'une part les *subroutines*, de l'autre le λ-calcul. Les *subroutines* sont les fonctions "de base", elles n'apparaissent pas dans l'environnement mais sont définies explicitement dans le code C++ de l'interprète. On peut citer `+`, `*`, `car` par exemple. Elles sont les briques qui permettent de coder des fonctions plus complexe en λ-calcul. Par exemple :

```
(lambda (n m) (+ (* n m) 2) 1 4)
```

Le `lambda` est un mot clé du langage qui indique à l'interprète que ce qui suit doit être vu comme une λ-expression, ensuite `(n m)` représente l'ensemble des variables, ensuite l'expression à évaluer : `(+ (* n m) 2)` et enfin les valeurs des paramètres `1 4`. Cette expression se traduit mathématiquement comme suit :

$$f(1,4) \text{ où } f : (n,m) \mapsto n \times m + 2$$

Informatiquement l'évaluation d'une λ-expression `f` se passe en deux étapes. Dans un premier temps on crée une copie de l'environnement global actuel auquel on ajoute des *bindings* entre chaque variables de la λ-expression. Il suffit ensuite d'appeler `eval` sur l'expression avec l'environnement que l'on vient de créer. Cette environnement est local, en effet il n'est utilisé que pour évaluer l'expression interne à `f`.

2 Interprète statique/dynamique

2.1 Concept

Jusqu'à présent l'interprète était dynamique c'est-à-dire que l'évaluation d'une *lambda*-expression était réalisé par rapport à l'environnement global. Cela peut poser plusieurs problèmes, par exemple si la définition de `f` fait référence à un objet `a`, l'évaluation de `f` dépendra de la valeur **actuelle** de `a`. Pour résoudre ce problème il faudrait évaluer `f` dans son environnement de définition (nous appellerons clôture de `f` cet environnement). Si l'on fait ainsi `f` est dite statique.

2.2 Implémentation

Pour intégrer l'évaluation statique il est nécessaire de changer l'implémentation de l'interprète en profondeur. Une solution envisageable serait de modifier l'environnement pour qu'un *binding* soit la représentation d'un triplé `(nom,valeur,cloture)`, malheureusement cela demande de recoder toutes les fonctions de l'interface environnement.

Une deuxième méthode consiste à légèrement modifier la classe `Cell` en ajoutant un attribut `closure` de type environnement qui contient la clôture de la `Cell` (ou le pointeur nul si la `Cell` est dynamique). On rajoute aussi des méthodes pour modifier et accéder à cette clôture dans la classe `Cell`. Maintenant les objets Lisp peuvent avoir une clôture. Il faut encore définir l'équivalent de `define` pour les objets statiques. Cette méthode `definestat` crée un *binding* statique pour lequel la clôture de sa partie "valeur" est initialiése à l'environnement global actuel. Enfin il faut modifier `apply` plus précisément la partie qui gère le mot clé `lambda`. Dans le cas où la λ-expression est statique il faut faire les *bindings* des variables dans la clôture de celle-ci puis appeler `eval` de l'expression dans l'environnement créé.

3 Extensions pour l'utilisateur

3.1 Gestion d'erreur

On laisse à l'utilisateur la possibilité d'écrire ce qu'il souhaite et en particulier du code non fonctionnel. On doit vérifier à chaque étape que les instructions sont rentrées correctement que l'utilisateur ne fait pas d'erreur et lui signaler lorsqu'il en fait.

Pour cela nous avons définis différentes classes d'erreurs. Premièrement nous avons définis la classe `Custom_exception` qui hérite de `runtime_error`. Les éléments de cette classe ont un attribut `string message` qui contient un message d'erreur et la méthode `string what()` qui permet de récupérer le message d'erreur. Comme les erreurs sont ratrappées à deux niveaux différents nous avons définis deux classes d'erreurs qui héritent de `Custom_exception`, `Toplevel_exception` pour les erreurs que l'on souhaite ratrapper dans la boucle *Toplevel* et `Evaluation_exception` pour les erreurs que l'on souhaite ratrapper dans la boucle *Eval*. Chaque fonction ayant des demandes particulières pour le nombre de valeurs ainsi que leur type. Chaque fonction gère donc individuellement ses erreurs grâce aux fonctions `void eval_error` et `void toplevel_error` qui permettent de *throw* une erreur.

Dans la boucle *Eval* les fonctions n'ont pas accès à leurs noms, on intercepte donc les messages d'erreurs afin de lancer une `Toplevel_exception` avec un message d'erreur de la forme `nom de la fonction : message d'erreur`. Les erreurs dans la boucle Toplevel sont rattrapées afin d'afficher le message d'erreur à l'utilisateur et relancer l'interpréteur avec l'ancien environnement.

3.2 Fonction *load*

La possibilité de charger des fichier et créer des modules permet de créer des programmes de taille importante tout en gardant le code clair. Dans cette optique, l'intégration d'une fonction de chargement de fichier *load* est un avantage pour l'utilisateur. Pour charger un fichier on utilise une fonction du lexer permettant de changer le flux d'entrée. Lorsque le `Toplevel` trouve un load, on lance la suite d'instructions suivantes :

— sauvegarder le flux actuel,
— créer un flux à partir du chemin du fichier,
— changer le flux d'entrée pour le nouveau flux
— lancer une boucle du toplevel (*via* `toplevel.go()`)
— libérer le nouveau flux
— changer le flux d'entrée pour remettre l'ancien

Une subtilité cependant, le flux standard stdin doit être réinitialisé une fois qu'on l'a quitté pour de nouveau fonctionner correctement.

3.3 *Garbage collector*

L'utilisateur dispose à ce niveau d'un interprète Lisp fonctionnel implémenté comme expliqué précédemment. Lors de la création de programmes contenant un grand nombre d'instructions, on peut craindre que l'accumulation d'instances d'objets Lisp dans la mémoire ne cause une perte de performance. Pour éviter cela, il faudrait libérer la mémoire au fur et à mesure du programme quand celle-ci n'est plus utilisée. Se pose alors le problème de la condition de libération de la mémoire : il faut être assuré qu'un objet ne sera plus utilisé avant de pouvoir le supprimer. Pour automatiser cette suppression et vérification on implémente un *garbage collector*.

3.3.1 Principe du *Mark and Sweep*

Pour créer notre *garbage collector*, on utilise l'algorithme du *Mark and Sweep*. On considère qu'on possède un ensemble *Root* d'objets qu'on désire garder. On considère aussi qu'on a un ensemble *All* contenant des pointeurs vers chaque objet instancié. On se donne la possibilité de marquer des objets. L'algorithme se découpe en 3 phases.

— Étape *Clean* : on enlève les marques de tous les objets de l'exécution (*All*). (voir figure 4)
— Étape *Mark* : par un parcours depuis l'ensemble des objets de base (*Root*), on marque tous les objets accessibles par un chemin de pointeurs. (voir figure 5)
— Étape *Sweep* : on supprime tous les objets de l'exécution (*All*) qui ne sont pas marqués en libérant la mémoire associée. (voir figure 6)

On ne garde ainsi que les objets du *Root* et les objets qui sont nécessaire à leur fonctionnement.

3.3.2 Implémentation

Le garbage collector implémenté dans l'interprète Lisp se base sur deux structures. On définit ainsi une classe `Garbage_collector` et une interface `Collectable` sous forme d'une classe virtuelle. Tout objet qu'on souhaite traiter avec le *garbage collector* doit hériter de `Collectable` et redéfinir la fonction `void mark_used()` . La redéfinition de cette méthode permettra de

faire le parcours de l'étape *Mark*. Le constructeur de `Collectable` par un appel à une méthode de `Garbage_collector` ajoute automatiquement au vecteur *All* un pointeur vers l'objet en cours d'instanciation. Ici c'est la classe `Cell` qui hérite de `Collectable`. L'ensemble des objets à traiter et des objets à garder sont enregistrés dans `Garbage_collector` sous la forme de `vector<Collectable*>`. La classe `Garbage_collector` implémente des méthodes pour effectuer les étapes *Clean*, *Mark* et *Sweep*. Un diagramme UML du *garbage collector* est en figure 7. Entre chaque appel, il ne reste plus qu'à appliquer les différentes étapes de l'algorithme *mark and sweep* pour nettoyer la mémoire.

Conclusion

Nous avons réussi à implémenter un interpréteur Lisp dynamico-statique, ce qui permet de faire facilement des fonctions doublement récursives. De plus, notre *garbage collector* permet de limiter les fuites mémoires : avec, nous avons seulement, 0,03 % de fuites contre 97 % sans. Enfin, notre système de chargement des fichiers permet de créer des modules Lisp et de faciliter la programmation.

En extension, nous aurions pu optimiser le *garbage collector*, notamment en libérant l'environnement avant de quitter l'interpréteur. De plus, nous n'avons pas eu le temps d'implémenter un mode débug, qui aurait permis de faciliter l'utilisation de l'interpréteur.

Annexes

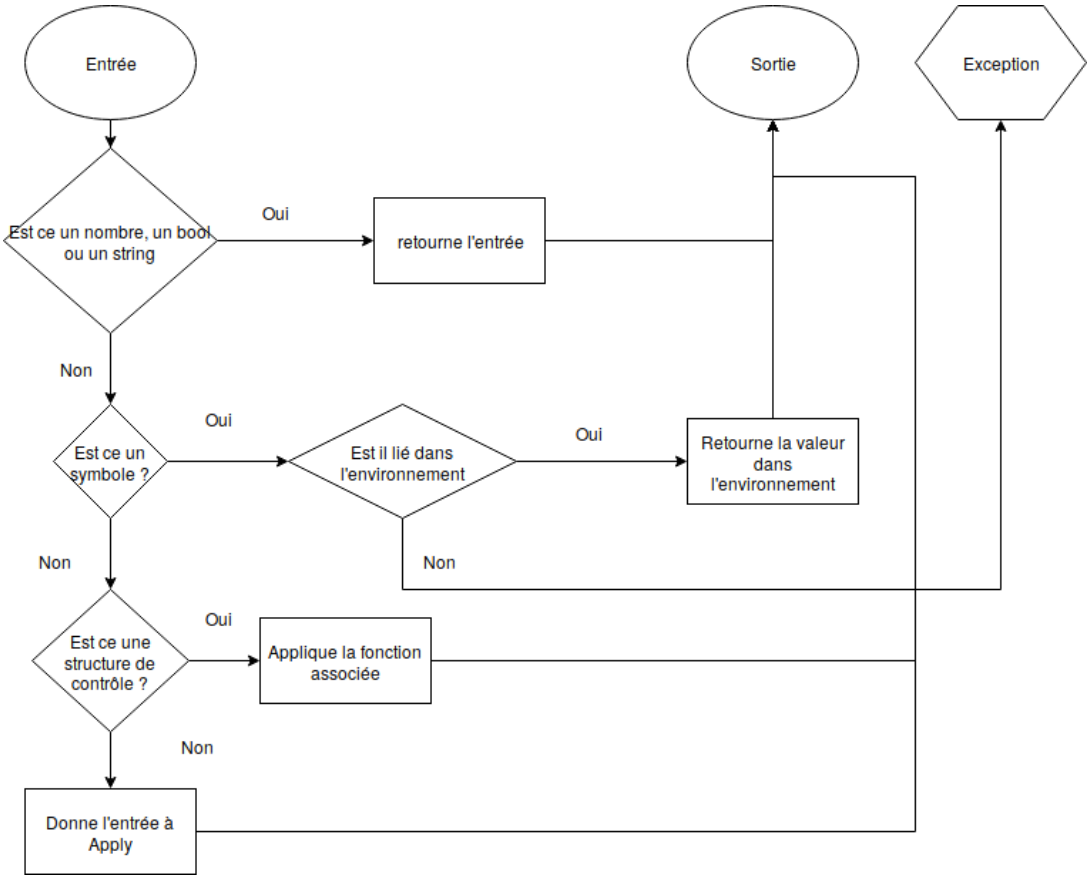

FIGURE 1 – Diagramme de la fonction Eval

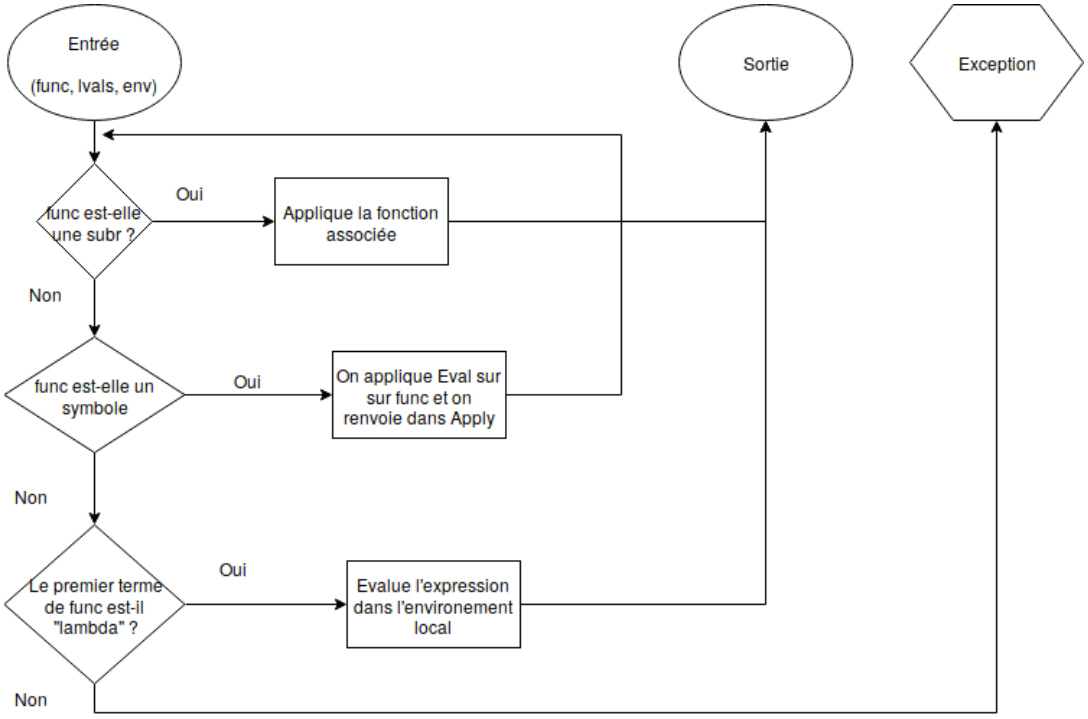

FIGURE 2 – Diagramme de la fonction Apply

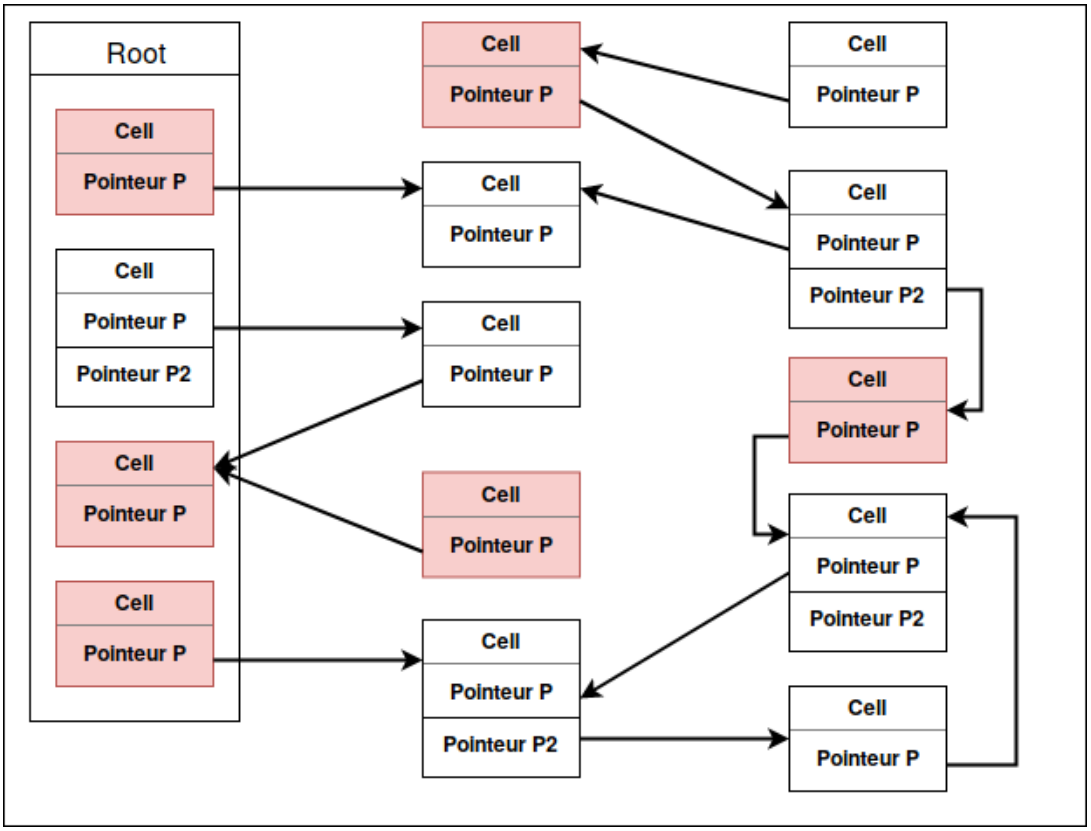

FIGURE 3 – Départ de l'algorithme du *garbage collector*

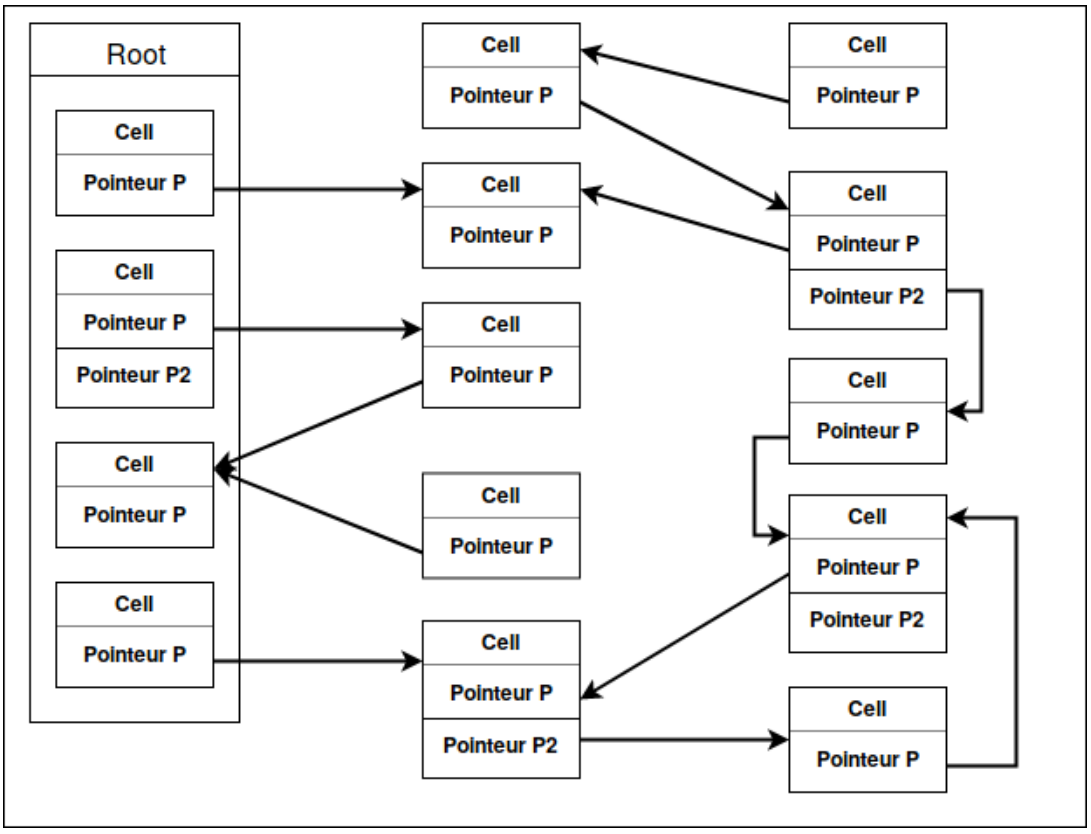

FIGURE 4 – Etape *clean* du *garbage collector*

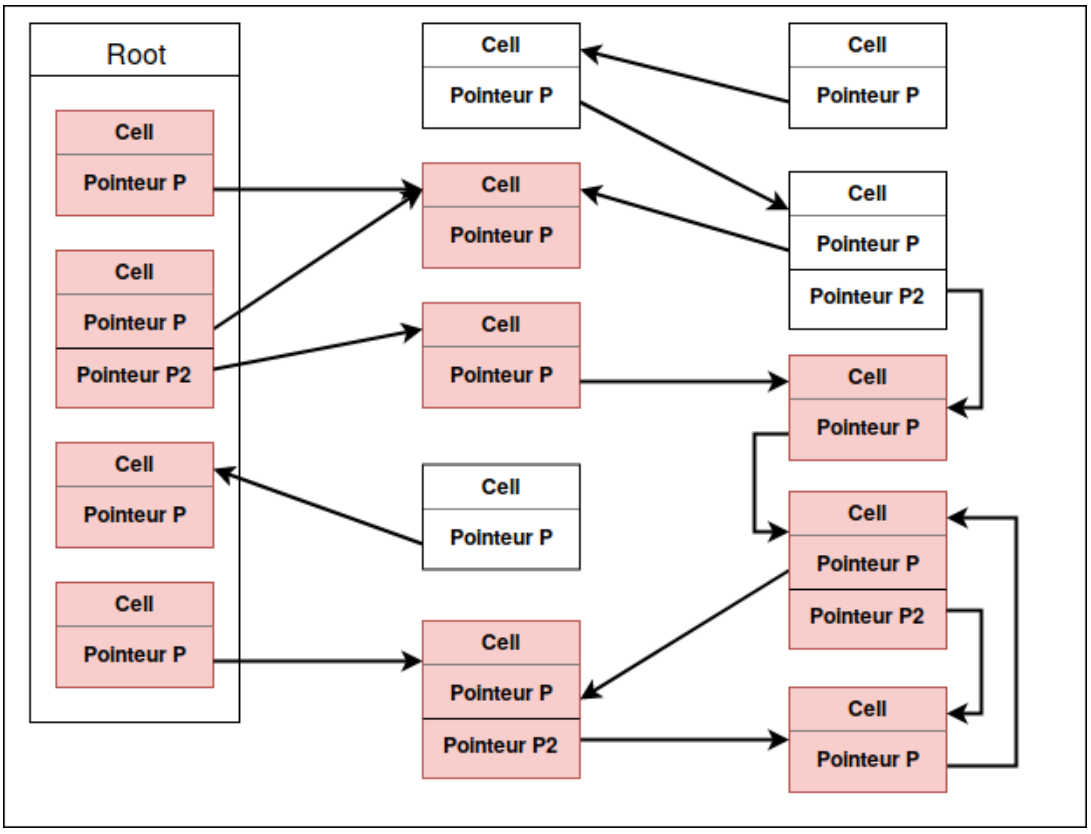

FIGURE 5 – Etape *mark* du *garbage collector*

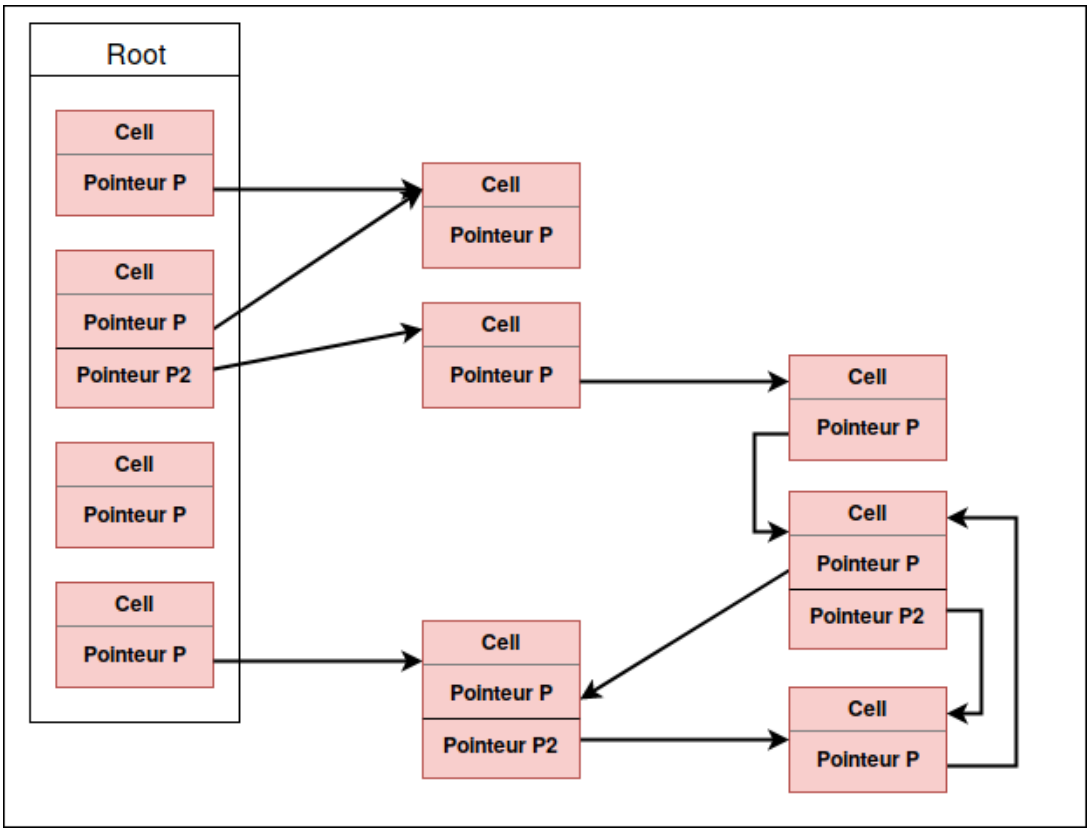

FIGURE 6 – Etape *sweep* du *garbage collector*

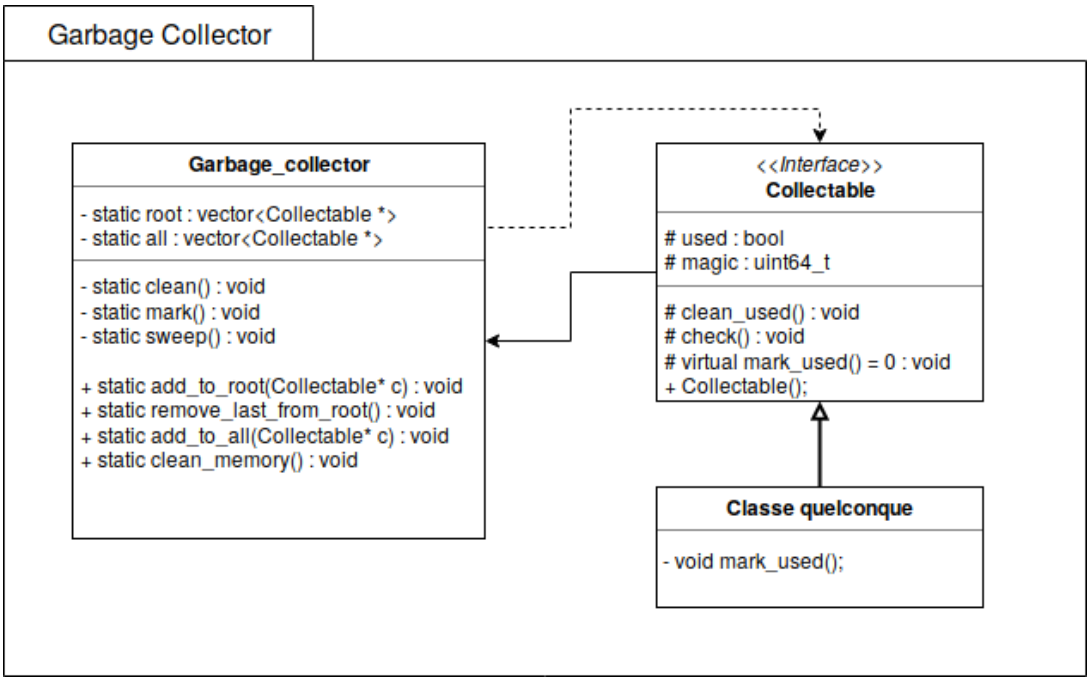

FIGURE 7 – Diagramme UML du *garbage collector*