

L'état mémoire

Mohammad Nauval

10 décembre 2017

Dans cette partie, nous allons examiner l'implémentation de l'état mémoire dans notre projet de compilation. La mémoire est l'un des éléments importants du projet et elle sert à gérer le stockage de tous les données déclarées dans un programme Minijaja. Ces données peuvent être des variables des différents types, des tableaux, et des déclarations des méthodes.

1 Conception

L'état mémoire du projet est composé de trois éléments : le dictionnaire de données, le tas, et la pile. Nous allons regarder la conception de chacun de ces trois éléments.

1.1 Le Dictionnaire de Données

Comme son nom indique, le dictionnaire de données contient les données déclarées dans un programme Minijaja. Le dictionnaire de données est une table de hachage. Une table de hachage est une implémentation du tableau associatif, elle permet une association clé-valeur.

On peut accéder à chaque valeur du tableau par sa clé. L'accès se fait par une fonction de hachage qui transforme une clé en une valeur de hachage (un nombre) indexant les éléments de la table, ces derniers sont appelés *buckets* en anglais. Il nécessite de stocker dans les *buckets* la paire clé-valeur et pas uniquement la valeur.

Le fait de générer une valeur de hachage à partir d'une clé peut engendrer un problème d'une collision, c'est-à-dire que deux clé différents, pourront se retrouver associées à la même valeur de hachage et donc au même *bucket*. Pour résoudre ce problème, premièrement nous devons choisir une bonne fonction de hachage pour minimiser la collision. La fonction de hachage que nous avons choisi la fonction de hachage FNV1.

```
private int hash(final String key) {
    int fnv1Init32 = 0x811c9dc5;
    int fnv1Prime32 = 16777619;
    int hash = fnv1Init32;

    byte[] data = key.getBytes();
    for (byte aData : data) {
        hash ^= (aData & 0xff);
        hash *= fnv1Prime32;
    }
    return hash;
}
```

FIGURE 1 – Fonction de hachage FNV1

La fonction de hachage FNV1 n'est pas une fonction de hachage parfaite. Une fonction de hachage est dite parfaite si elle n'engendre aucune collision. En effet, nous avons toujours le problème de collision. Pour le résoudre, chaque case ou *bucket* contient une liste chaînée. Si deux clés se retrouvent au même *bucket*, les deux paires clé-valeur seront stockées dans cette liste.

Voici la représentation graphique de notre dictionnaire de données :

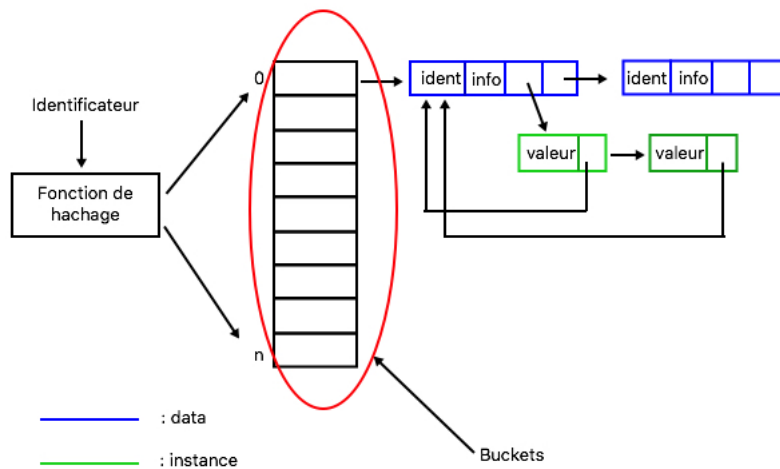


FIGURE 2 – Dictionnaire de données

L'image est notre dictionnaire de données. La table de hachage a pour capacité initiale n , c'est-à-dire qu'il contient n *buckets*. Comme mentionné avant, chaque *bucket* contient une liste de data. Data est un nœud de la

liste, chaque data contient un identificateur, les informations concernant la variable, le tableau, ou la méthode déclarée. Chaque data possède une liste d'instances. Chaque instance possède une valeur et un pointeur vers son data pour récupérer les informations concernant cette instance.

Notre dictionnaire de données sert à stocker les données déclarées dans un programme Minijaja. Par exemple, lors de la déclaration d'une variable de type entier :

```
int test = 3;
```

Cette déclaration a pour l'identificateur la chaîne de caractères "test". Pour stocker cette déclaration dans le dictionnaire, on prends l'identificateur et on implémente la fonction de hachage à cet identificateur. Le résultat de cette fonction est un nombre qui indique l'indice du *bucket* dans lequel la donnée sera stockée. Si, par exemple la fonction de hachage transforme la chaîne "test" en un nombre 0, on crée un nœud data, ce nœud a pour l'identificateur "test". La nature de l'objet est *variable* et le type d'objet est *integer*, ces deux informations sont aussi stockées dans le nœud data. Nous ajoutons aussi un nœud instance à la liste d'instances du nœud data, ce nœud instance a pour valeur 3. Nous ajoutons le nœud data à la liste chaînée du *bucket* à l'indice 0.

Notre table de hachage a pour *load factor* 0.75, c'est-à-dire que si le nombre d'éléments atteint 75% de la capacité initiale, nous augmentons la capacité de la table de hachage. Ceci est fait pour diminuer la taille de la liste chaînée que chaque *bucket* possède pour garder la complexité de la recherche d'un élément dans la table.

L'intérêt de l'utilisation de la table de hachage est la complexité de la recherche d'un élément. Comme les tables ordinaires, la table de hachage permet un accès en $O(1)$ en moyenne. Toutefois, comme plusieurs paires clé-valeur peuvent se trouver dans un même *bucket*, le temps d'accès dans le pire cas est de $O(n)$.

1.2 Le tas

Le tas sert à stocker les données d'un tableau. Lors que l'utilisateur déclare un tableau d'une taille dans le programme Minijaja, les données de ce tableau sont stockées dans le tas. Le tas lui-même est un tableau.

Lors que le tas est initialisé, Le tas est initialisé avec une capacité donnée ou une capacité par défaut. Le capacité par défaut est de 256. Voici le tas après l'initialisation. Supposons que le tas a pour capacité sa capacité par défaut. Le tas possède un bloc libre de taille 256 à l'adresse 0.

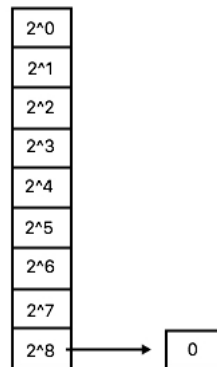


FIGURE 4 – La table d’adresses des blocs libres du tas après l’initialisation du tas

Imaginons que l’utilisateur déclare un tableau ”t1” de taille 10. Nous parcourons la table d’adresses pour chercher un bloc dont la taille est égale ou supérieure à 10. Nous avons trouvé un bloc de taille 256 à l’adresse 0. Ce bloc est ensuite découpé.

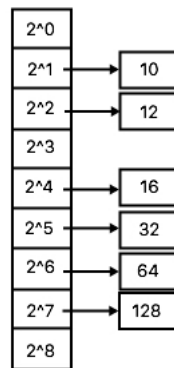


FIGURE 5 – Le tableau d’adresses des blocs libres du tas après la déclaration d’un tableau de taille 10

Nous pouvons regarder qu’il nous reste 6 blocs avec la taille totale de 246. Les éléments du tableau ”t1” seront stocké dans le tas dans le bloc de taille 10 situé à l’adresse 0 jusqu’à l’adresse 9.

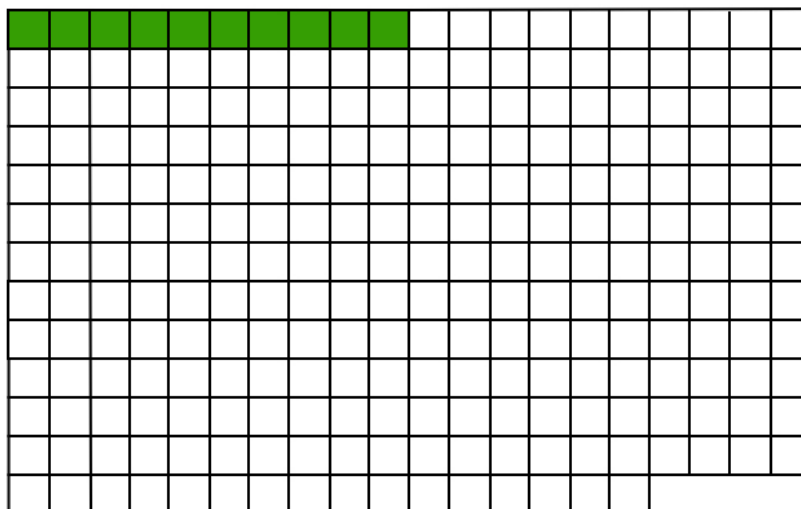


FIGURE 6 – Le tas après la déclaration du tableau de taille 10

Le bloc en vert est le bloc destiné à stocker les éléments du tableau "t1".

Supposons que l'utilisateur déclare ensuite un tableau "t2" de taille 3. Nous parcourons la table d'adresses de blocs libres, nous trouvons un bloc de taille 4 à l'adresse 12, nous découpons ce bloc.

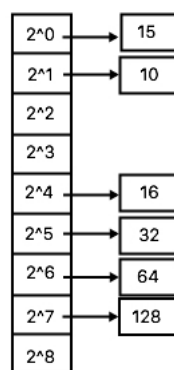


FIGURE 7 – La table d'adresses des blocs libres du tas après la déclaration d'un tableau de taille 3

En effet, les éléments du tableau "t2" seront stockés dans le bloc situé

l'adresse 12 jusqu'à l'adresse 14.

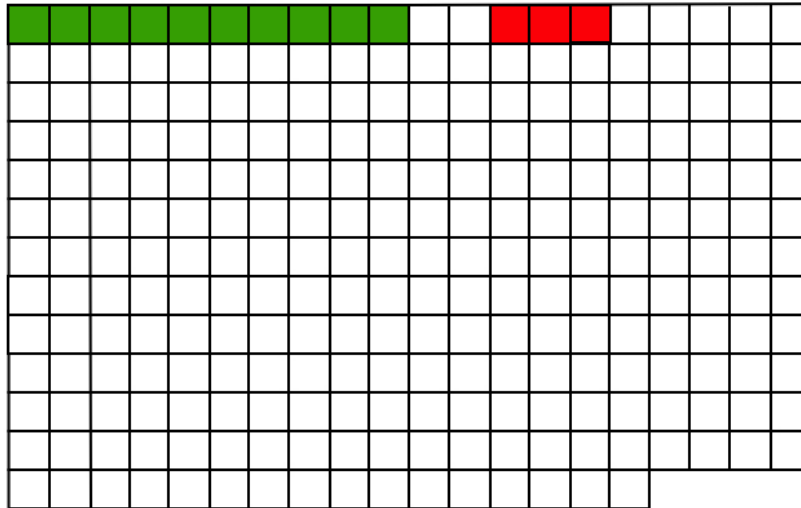


FIGURE 8 – Le tas après la déclaration du tableau de taille 3

Le bloc en rouge est le bloc destiné à stocker les éléments du tableau "t2".

Un problème apparaît. Comme ce que nous pouvons voir, il y a deux espaces libres entre le bloc en vert et le bloc en rouge, c'est le problème de fragmentation. Ces deux espaces risquent d'être perdus. Pour résoudre ce problème, nous faisons en sorte qu'à chaque déclaration du tableau, le bloc destiné à stocker ses données est aligné au plus gauche possible. Nous faisons aussi des fusionnement des blocs qui nécessitent d'être fusionnés.

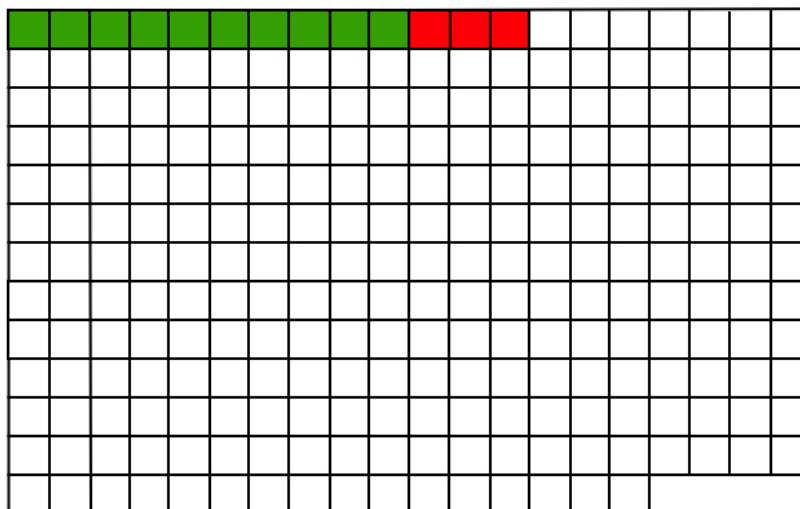


FIGURE 9 – Le nouveau bloc est aligné au plus gauche possible

2^0	→	13
2^1	→	14
2^2		
2^3		
2^4	→	16
2^5	→	32
2^6	→	64
2^7	→	128
2^8		

FIGURE 10 – La table d'adresses des blocs libres après l'alignement du bloc t2 au plus gauche possible

La même opération est faite lors de la suppression du bloc du tas. De cette manière, les blocs libres sont toujours regroupés à droite des blocs assignés. Et il n'y a pas de blocs libres situés entre des blocs assignés.

Nous avons aussi une table de symbole pour mémoriser les blocs assignés. Dans notre exemple, la table de symbole est comme ci-dessus :

Identificateur	adresse	taille	références
1	0	10	1
2	10	3	1

FIGURE 11 – La table de symbole du tas

L'identificateur du bloc est la clé de cette table. Pour chaque clé, nous avons l'adresse du bloc, la taille du bloc, et aussi le nombre de références pour ce bloc.

1.3 La pile

La pile sert à stocker les instances des variables, des tableaux, et des méthodes déclarées dans le programme Minijaja. Par exemple, lors de la déclaration des deux variables de type entier :

```
int i = 3;
int j = 4;
```

Lors de la déclaration de ces deux variables, nous ajoutons à la table de hachage deux nœuds data, et nous ajoutons aussi deux instances à la pile.

<i,INTEGER,VARIABLE,3>
<j,INTEGER,VARIABLE,4>

Toutes les opérations se baseront sur la pile. Dans le cas où l'instance est un tableau, sa valeur est l'identificateur du bloc contenant les éléments du tableau dans le tas. Et dans le cas où l'instance est une méthode, sa valeur est le nœud AST de la méthode.

1.4 La Mémoire

La mémoire contient trois éléments : le dictionnaire de données, le tas, et la pile. Lors de la déclaration d'une donnée, l'identificateur et les informations de la donnée sont stocké dans le dictionnaire de données, la valeur de la donnée est stockée dans la pile. Dans le cas de la déclaration d'une tableau, ses éléments sont stockés dans le tas.

Les opérations qui peuvent être effectuées par la mémoire sont :

```

public interface IMemory {
    void push(Instance instance);
    Instance pop();
    Instance peek();
    void swap();
    void declVar(String identifier, EnumType type, Value value);
    void identVal(String identifier, EnumType type, int index);
    void declCst(String identifier, EnumType type, Value value);
    void declTab(String identifier, EnumType type, Value value);
    void declMeth(String identifier, EnumType type, Value value);
    void retirerDecl(String identifier);
    void affecterVal(String identifier, Value value);
    void affecterValT(String identifier, Value value, int index);
    void affecterType(String identifier, EnumType type);
    void expParam(ListExpression listExpression, Parameters parameters);
    Value val(String identifier);
    Value valT(String identifier, int index);
    EnumObject objet(String identifier);
    EnumType sorte(String identifier);
    Parameters parametre(String identifier);
    Declarations declaration(String identifier);
    Instructions corps(String identifier);
}

```

FIGURE 12 – Les opérations de la mémoire

2 Traitement des erreurs

Les erreurs peuvent apparaître lors des opérations dans la mémoire. Lors que une erreur est produite, une exception est levée. Par exemple, dans le cas où l'utilisateur essaie d'affecter une valeur à une variable dans la pile, mais la pile est vide. Une exception du type `NullStackException` est levée.

```

/**
 * Assigns a value to a data instance.
 * @param identifier identifier of the data instance whose value will be modified.
 * @param value the value that will be assigned.
 */
@Override
public void affecterVal(String identifier, Value value) {
    if (stack.isEmpty()) {
        throw new EmptyStackException();
    }
}

```

FIGURE 13 – L'affectation de la variable dans le cas où la pile est vide

Il existe plusieurs autres exceptions dans la mémoire. Ces exceptions sont aussi utile pour le contrôle de type.

3 Choix de test

Nous effectuons des tests fonctionnels pour chaque fonctionnalité de la mémoire et ses composants. Par exemple, la table de hachage est un com-

posant de la mémoire, nous faisons des tests pour s'assurer que la table de hachage fonctionne correctement. Nous avons un test qui s'assure que la table de hachage augmente sa capacité si le nombre des éléments contenus dans la table de hachage atteint 75% de sa capacité, nous avons aussi un test qui s'assure que lors de l'insertion d'une valeur avec une clé qui existe déjà dans la table de hachage, l'ancienne valeur est supprimée et remplacée par la nouvelle valeur. Voici la capture de ce test.

```
@Test
public void putExistingElement() {
    HashMap<String, Data> table = new HashMap<>();
    Data data = new Data( Identifier: "test", EnumType.INTEGER, EnumObject.VARIABLE);
    table.put("test", data);
    Assert.assertTrue(table.get("test").getType().equals(EnumType.INTEGER));

    Data data2 = new Data( Identifier: "test", EnumType.BOOLEAN, EnumObject.VARIABLE);
    table.put("test", data2);
    Assert.assertTrue(table.get("test").getType().equals(EnumType.BOOLEAN));
}
```

FIGURE 14 – Un exemple du test fonctionnel de la table de hachage

Nous pouvons remarquer aussi que pour chaque test fonctionnel, nous déclarons de nouveau l'élément que nous testons. Dans l'image ci-dessus, nous déclarons la table de hachage au début du test. Nous n'utilisons pas une seule table de hachage pour tous les tests. Nous avons décidé de faire ce choix pour que chaque test ne dépend pas des autres test, autrement dit, chaque test est indépendant et peut être exécuté individuellement.

Cependant, dans certains cas, nous avons besoin de tester une fonctionnalité avant de tester une autre fonctionnalité parce que la deuxième fonctionnalité a besoin d'appeler la première fonctionnalité. Par exemple, pour tester la méthode d'affectation d'une valeur à une variable dans la pile, nous avons besoin d'utiliser la méthode de la déclaration de la variable. C'est la raison pour laquelle nous devons être sûrs que la méthode de la déclaration de la variable fonctionne normalement avant de pouvoir l'utiliser. Autrement dit, nous avons besoin de tester la déclaration de la variable avant l'affectation de la valeur à une variable.

```

@Test
public void testAffecterVal() throws MemoryException {
    Memory memory = new Memory();
    memory.declVar( identifier: "i", EnumType.INTEGER, new IntegerValue(5));
    Assert.assertTrue(memory.getHashMap().containsKey("i"));
    Assert.assertEquals( expected: "<i,INTEGER,VARIABLE,5>\n", memory.toString());

    memory.affecterVal( identifier: "i", new IntegerValue(4));
    Assert.assertEquals( expected: "<i,INTEGER,VARIABLE,4>\n", memory.toString());
}

```

FIGURE 15 – Le test de l’affectation d’une valeur à une variable