If I had eight hours to chop down a tree, I'd spend six hours sharpening my ax.

Abraham Lincoln

Fonctions de hachage sur l'espace des sous-arbres

Sommaire

La recherche de similarité sur du code source représenté sous la forme d'arbres de syntaxe nécessite des méthodes efficaces et de complexité praticable pour mettre en évidence des sous-arbres similaires. La comparaison exhaustive de toutes les paires de sous-arbres, soit n^2

paires pour deux projets représentés par des arbres de n nœuds, n'est pas envisageable en pratique, d'autant plus si au-delà de la détermination d'une similarité exacte, des techniques d'alignement d'arbres sont mises en oeuvre. Il devient alors indispensable de réduire le nombre de comparaisons coûteuses d'arbres en utilisant des métriques ou valeurs de hachage afin de regrouper des sous-arbres de similarité supposée.

Dans ce chapitre, nous évoquons quelques métriques utilisables sur les sous-arbres ainsi que certaines méthodes de hachage. Nous présentons également quelques techniques d'abstraction d'arbres de syntaxe permettant ainsi l'obtention de valeurs de hachage communes pour des sous-arbres présentant la même abstraction.

9.1 Quelques méthodes de hachage de sous-arbres

9.1.1 Métriques

Les métriques sur des unités structurelles du code source sont des mesures permettant de quantifier certaines propriétés du code. Elles s'expriment le plus souvent dans un espace vectoriel réel. Nous passons maintenant en revue quelques unes des métriques les plus couramment utilisées.

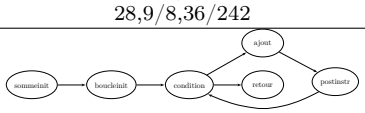
Métriques de quantification de la complexité du code et du style de programmation

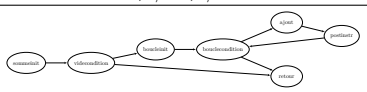
Mesurer la complexité du code source a toujours été une préoccupation majeure en ingénierie logicielle. Ces métriques permettraient d'estimer la complexité du code et donc son coût de maintenabilité. Nous présentons ici les métriques de Halstead et la métrique de complexité cyclomatique pour finir sur une métrique basée sur le vecteur de comptage des types de nœuds. La figure 9.1 présente deux versions d'un court code de calcul de moyenne sur lesquels les métriques présentées ont été appliquées.

Métriques de Halstead

Les métriques de Halstead [108] sont calculées à partir du comptage des opérateurs et opérandes totaux et distincts présents dans l'unité structurelle analysée : le vecteur $V = (\alpha, \bar{\alpha}, \beta, \bar{\beta})$ de l'unité comportant α opérateurs distincts, $\bar{\alpha}$ opérateurs totaux, β opérandes distincts et $\bar{\beta}$ opérandes totaux est calculé. Des mesures de longueur $(\bar{\alpha} + \bar{\beta})$, de vocabulaire $(\alpha + \beta)$, de volume $((\bar{\alpha} + \bar{\beta}) \log_2(\alpha + \beta))$, de difficulté $(\frac{\alpha\bar{\beta}}{2\beta})$ et d'effort (produit du volume et de la difficulté) sont dérivées de ce vecteur. Les opérateurs sont alors définis comme l'ensemble des opérateurs arithmétiques, des appels de fonction et des opérations d'affectation. Les opérandes sont quant à eux les variables locales, membres de structures et constantes.

Il est aisé de constater que l'ajout ou la suppression de code inutile ainsi que la réécriture d'expressions peut avoir un impact important sur le vecteur V. D'autre part, il existe des risques importants que plusieurs unités structurelles de vecteurs accidentellement proches n'abritent aucune similarité réelle.

<table>
<tr><th></th><th>Code original</th><th>Code modifié</th></tr>
<tr><td></td><td><pre>
double moyenne(double[] tab)
{
double somme = 0.0;
for (int k = 0; k < tab.length; k++)
 somme += tab[k];
return somme / (double)tab.length;
}
</pre></td><td><pre>
double moyenne(float[] tab)
{
 float somme = 0.0f;
 if (tab.length == 0) return 0.0f;
 else
  for (int k = tab.length; k >= 0; k−−)
   somme += tab[k];
 return somme / (double)tab.length;
}
</pre></td></tr>
<tr><td>Opérateurs
Opérandes
Longueur/vocabulaire
Volume/difficulté/effort</td><td>11 (9 distincts)
13 (7 distincts)
24/16
28,9/8,36/242</td><td>14 (10 distincts)
16 (7 distincts)
30/17
36,9/11,4/421</td></tr>
<tr><td>Flux de contrôle
Métrique cyclomatique</td><td> 

$E - N + 2P = 6 - 6 + 2 = 2$</td><td> 

$E - N + 2P = 8 - 7 + 2 = 3$</td></tr>
</table>

FIG. 9.1 – Une fonction et sa version obfusquée avec le calcul de leurs métriques de Halstead et cyclomatique

Nous pouvons noter que les métriques de Halstead ont servi de base à la réalisation du (très probable) premier outil de recherche de similarité sur du code source (en langage Fortran) proposé par Ottenstein [75] en 1976.

Complexité cyclomatique de McCabe

McCabe a proposé une métrique [110] sur le graphe de contrôle de flux d'une unité structurelle. Un tel graphe représente l'ensemble des instructions d'un programme (nœuds), deux instructions étant liées par une arête s'il existe un chemin d'exécution où ces deux instructions sont exécutées séquentiellement. La complexité cyclomatique C est définie par $C = E - N + 2P$ où E est le nombre d'arêtes, N le nombre de nœuds et P le nombre de composantes connexes du graphe. Cette valeur reflète le nombre d'expressions conditionnelles issues de structures de contrôles présentes dans le code.

Si la complexité cyclomatique peut être difficilement réduite, une augmentation artificielle par l'ajout de structures de contrôle inutiles (cf 4.6) est possible. Au-delà d'une analyse statique, une détermination de la complexité cyclomatique dynamique par le suivi des chemins d'exécution serait plus fiable pour localiser les conditionnelles suspectées invariantes. D'autre part, cette métrique dispose d'un pouvoir discriminateur réduit : l'espace de valeurs de complexité cyclomatique est trop faible afin de pouvoir disposer de groupes de sous-arbres d'effectifs suffisamment petits. Il est en effet rare de trouver des fonctions dont la complexité cyclomatique est supérieure à 20.

Vecteur de comptage de nœuds ou de sous-arbres

Étant donnée une unité structurelle représentée par un arbre de syntaxe, il est possible d'exprimer le nombre de chaque type de nœuds présents au sein de cet arbre : nous pouvons ainsi en extraire un vecteur de comptage de nœuds. La taille de ce vecteur, représentant le

nombre de types de nœud différents peut être réduite par une opération d'abstraction des types : nous traiterons de ce procédé en 9.2. Ce vecteur de comptage peut être généralisé au comptage des différents sous-arbres de taille ou de hauteur bornée. Cette technique est notamment employée par l'outil de recherche de similitudes Deckard [70] qui génère pour chaque sous-arbre à hacher un vecteur caractéristique comptant les petits sous-arbres inclus. Les différents sous-arbres vectorisés sont ensuite regroupés en utilisant une technique de hachage localement sensible (Locality Sensitive Hashing, LSH) [52]. Cette technique permet de regrouper des vecteurs de distance faible et donc des sous-arbres suffisamment voisins dans l'espace vectoriel des effectifs de sous-arbres caractéristiques.

Tout comme la cardinalité du nombre de k-grams pour k suffisamment faible est assez réduite par rapport à la cardinalité théorique des $|\Sigma|^k$ k-grams, il existe également un nombre limité de petits arbres, qu'ils soient définis par une taille k ou une hauteur h limite. Il est toutefois nécessaire de maintenir la taille ou la hauteur limite suffisamment basse pour restreindre la taille des vecteurs à manipuler.

L'utilisation d'un vecteur de comptage est totalement insensible aux opérations de transposition de code. Ce comportement ensembliste peut également induire des cas de structures non-similaires ayant un vecteur de comptage identique ou proche selon une fonction de hachage localement sensible. D'autre part, des modifications localisées comme la réécriture d'expressions peuvent modifier le type des petits sous-arbres considérés pour le comptage et donc le vecteur généré. L'ajout ou la suppression de code inutile peut également avoir une influence sur le vecteur.

Le vecteur de comptage peut également être projeté sur une unique valeur entière en sommant ses valeurs coefficientées par un représentant entier aléatoire de chacun des types (cf 9.1.2). Cette méthode se rapproche alors d'un hachage exact dans la mesure où les valeurs ne peuvent être comparées autrement que par une égalité stricte.

9.1.2 Hachage exact

Quelques propriétés du hachage exact

Les métriques sur les unités structurelles présentent la caractéristique d'être à valeurs sur un espace limité avec une sensibilité réduite en rapport avec certaines opérations d'édition du code. Ce qui peut être un avantage dans certaines situations d'obfuscation telle que la transposition de code, peut également générer des collisions entre structures manifestement différentes mais de métrique proche. D'autre part l'apparition de faux négatifs est inévitable pour certaines opérations d'édition agissant directement sur les propriétés du code utilisées pour le calcul de la métrique. *A contrario*, l'utilisation de méthodes de hachage exact ne permet de regrouper que des sous-arbres exactement similaires — moyennant l'abstraction choisie sur les types de nœuds — avec une probabilité de collision maîtrisable par le choix de la cardinalité de l'espace de hachage.

La valeur de hachage exact d'un sous-arbre ne possède aucune signification sémantique particulière : les valeurs de hachage de deux sous-arbres ne permettent pas de les comparer sur aucun critère. Dans ce sens, seule l'égalité de deux valeurs de hachage possède une signification :

si la fonction de hachage est parfaite et à valeurs sur un espace de cardinalité N, l'égalité signifie que les deux sous-arbres ont une probabilité $1 - \frac{1}{N}$ d'être identiques (moyennant l'abstraction usitée). Cette remarque conditionne le choix de la cardinalité de l'espace de hachage pour la mise en place d'une base d'empreintes : un compromis doit être alors trouvé sur la taille de la valeur pour minimiser la taille de la base tout en réduisant la probabilité de collision $\frac{1}{N}$. Nous introduirons dans le chapitre 10 un processus d'indexation utilisant des tailles de valeur de hachage adaptatives en fonction des collisions rencontrées.

Nous proposons ici deux méthodes récursives de hachage exact sur les arbres. Connaissant la valeur de hachage des sous-arbres enfants $C_1, C_2, \cdots, C_k$ d'un arbre A, nous pouvons obtenir la valeur de hachage de A en temps global $\Theta(k)$. Il en découle la possibilité de paralléliser le calcul de la valeur de hachage de A par l'emploi de fils d'exécution indépendants pour les calculs des valeurs de hachage de sous-arbres enfants.

On notera que pour des petits sous-arbres dont la cardinalité est réduite, l'usage d'une méthode de hachage peut être remplacé par l'énumération exhaustive de ceux-ci. Pour des sous-arbres de taille intermédiaire, une fonction de hachage exacte peut être utilisée mais nous pouvons prendre en compte la cardinalité réduite de l'espace de ces sous-arbres afin de réduire la taille de la valeur. La probabilité de collision théorique est de $\frac{1}{N}$ pour une fonction de hachage parfaite sur un espace de sous-arbres de cardinalité infinie. En revanche, sur un espace de sous-arbres de cardinalité C, la probabilité de collision s'élève à $p = \frac{(C \mod N)\cdot\alpha^+ + (2^k - C \mod N)\alpha^-}{C^2}$ avec $\alpha^+ = \lceil \frac{C}{N} \rceil$ et $\alpha^- = \lfloor \frac{C}{N} \rfloor$. Nous réalisons quelques tests présentés en section 9.1.3 avec certaines fonctions de hachage afin de quantifier les collisions (faux positifs) survenant lors du traitement d'arbres de différentes tailles.

Vecteur de valeurs de type

Préalablement à une opération de hachage exact d'une séquence de lexèmes ou d'un arbre, il est nécessaire de représenter chaque type de lexème ou de nœud par une valeur entière. L'ensemble des types manipulés est ainsi représenté par un vecteur de valeurs entières. Afin de limiter les possibilités de collision entre valeurs de hachage, le vecteur de valeurs devrait être généré aléatoirement. D'autre part, ce vecteur devrait être recalculable facilement avec la seule donnée des types, exprimés par exemple sous la forme de chaînes de caractères ou alors d'un identificateur entier séquentiel (qui peut être obtenu par tri lexicographique des chaînes). Nous pouvons ainsi utiliser un générateur pseudo-aléatoire linéaire congruentiel afin d'obtenir séquentiellement des valeurs pour les types d'identifiants séquentiels $1, 2, \cdots, n$.

Hachage de Karp-Rabin sur les arbres

Mots de Dyck Il est possible de représenter chaque arbre A par une séquence de lexèmes unique issue de son parcours en profondeur : il s'agit du mot de Dyck $\mathcal{DW}(A)$ de cet arbre. Nous pouvons choisir la convention suivante pour représenter l'arbre A dont la racine est de type a et ayant pour enfants les sous-arbres $C_1, C_2, \ldots, C_k$:

$$\mathcal{DW}(A) = (\rightarrow a) \cdot \mathcal{DW}(C_1) \cdot \mathcal{DW}(C_2) \cdots \mathcal{DW}(C_k) \cdot (a \leftarrow)$$

Si les arbres de syntaxe utilisent un alphabet Σ de types non terminaux ainsi qu'un alphabet Θ de types terminaux, le mot de Dyck est exprimé sur l'alphabet $(\rightarrow \Sigma) \cup (\Sigma \leftarrow) \cup \Theta$ afin

d'exprimer l'ensemble des lexèmes ouvrants et fermants pour chaque type de nœud non terminal ainsi que les lexèmes terminaux (d'arité nulle). Si chaque type de nœud possède une arité constante, l'utilisation de lexèmes fermants est inutile car l'interprétation du mot de Dyck n'est pas ambiguë.

Hachage de Karp-Rabin sur les mots de Dyck Les fonctions de hachage classiques manipulent des chaînes de lexèmes : il est possible de les utiliser pour calculer une valeur de hachage d'un arbre par l'intermédiaire de son mot de Dyck. Nous nous intéressons en particulier aux fonctions de hachage polynomiales, dites de Karp-Rabin, offrant des propriétés d'incrémentalité standard et d'incrémentalité forte (cf 8.1.2). Nous pouvons ainsi calculer, en temps linéaire de son nombre de nœuds et donc de la longueur de son mot de Dyck, la valeur de hachage d'un arbre. Lorsqu'à la fois la valeur de hachage de l'arbre et de tous ses sous-arbres sont nécessaires, comme c'est le cas pour les applications de recherche de sous-arbres similaires, le calcul peut être mené en temps $O(n \log n)$ pour un arbre de n nœuds. Nous présentons en figure 9.2 un exemple de calcul des valeurs de hachage de tous les sous-arbres de l'arbre de syntaxe déjà figuré 3.5.

Hachage cryptographique adapté aux arbres

Des fonctions de hachage cryptographique peuvent être utilisées afin de générer une valeur de hachage pour un arbre de syntaxe. Une fonction de hachage est dite de qualité cryptographique si, outre la minimisation du risque de collision accidentelle, elle permet de limiter les attaques portant sur la fabrication *a posteriori* de données correspondant à une valeur de hachage donnée ou la création simultanée d'un couple de données différentes de même valeur de hachage. Cette dernière propriété n'est cependant pas cruciale pour des applications de recherche de sous-arbres similaires. D'autre part, les fonctions de hachage cryptographiques possèdent une propriété d'effet d'avalanche stricte [57] : la modification de n'importe quel lexème ou type de nœud a pour conséquence une probabilité de $\frac{1}{2}$ d'inversion de chaque bit de l'empreinte. Ce n'est pas le cas d'une fonction de hachage polynomiale où la modification d'un des entiers d'entrée aura un effet moins important si celui-ci est proche de la fin de la séquence d'entrée. Toutefois l'usage de vecteurs de valeur de type pseudo-aléatoires permet de remédier à ce problème. La propriété d'avalanche stricte permet de réduire la taille d'une valeur de hachage de k bits à k' bits ($k' < k$) en sélectionnant k' bits quelconques : pour la suite, nous choisissons de garder les k' bits de poids faible (soit le modulo $2^{k'}$).

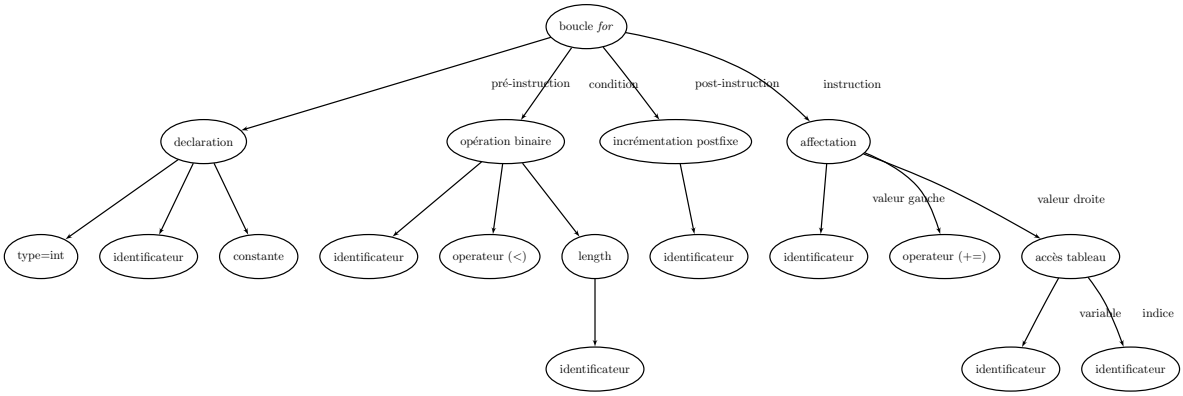
En règle générale, les fonctions de hachage cryptographique ne permettent pas de calculer rapidement la valeur de hachage de la concaténation de deux chaînes connaissant les valeurs individuelles liées à chacune des chaînes : elles ne satisfont pas les propriétés d'incrémentalité simple et forte. Elles ne sont donc pas adaptées au hachage de k-grams.

Nous proposons d'utiliser des fonctions de hachage cryptographique pour le calcul de valeur de hachage d'un arbre et de tous ses sous-arbres. À cet effet, nous calculons récursivement du bas de l'arbre vers le haut de l'arbre la valeur de hachage des sous-arbres. Si A est l'arbre traité de nœud racine a, $C_1, C_2, \cdots, C_k$ ses sous-arbres enfants et $\mathcal{C}$ la fonction de hachage cryptographique utilisée, nous calculons ainsi la valeur de hachage de A :

$$h(A) = \mathcal{C}(a \cdot h(C_1) \cdot h(C_2) \cdots h(C_k))$$

```
for (int i=0 ; i < tab.length ; i++) somme += tab[i] ;
```

(a) Code source



(b) Arbre de syntaxe abstrait (cf 3.5)

$$\begin{aligned}\mathcal{DW}(A_{\text{for}}) = {} & (\rightarrow \text{for }) \\ & (\rightarrow \text{declaration }) \,\text{type identificateur constante } (\text{declaration } \leftarrow) \\ & (\rightarrow \text{opbin }) \cdots (\text{opbin } \leftarrow) \\ & (\rightarrow \text{incr }) \cdots (\text{incr } \leftarrow) \\ & (\rightarrow \text{affectation }) \cdots (\text{affectation } \leftarrow) \\ & (\text{for } \leftarrow)\end{aligned}$$

(c) Mot de Dyck

$$h(A_{\text{for}}) = (((v((\rightarrow \text{for })) \times B + v((\rightarrow \text{declaration }))) + v(\text{type })) \times B + \cdots) + v((\text{for } \leftarrow))$$

(d) Hachage séquentiel polynomial du mot de Dyck

$$\begin{aligned}h(A_{\text{for}}) = {} & \Big(\Big(h(A_{\text{declaration}}) \times B^{|\mathcal{DW}(A_{\text{opbin}})|} + h(A_{\text{opbin}})\Big) * B^{|\mathcal{DW}(A_{\text{incr}})|} + h(A_{\text{incr}})\Big) \\ & * B^{|\mathcal{DW}(A_{\text{affectation}})|} + h(A_{\text{affectation}})\end{aligned}$$

(e) Hachage récursif polynomial des sous-arbres

$$h(A_{\text{for}}) = \mathcal{C}\left((\rightarrow \text{for }) \cdot h(A_{\text{declaration}}) \cdot h(A_{\text{opbin}}) \cdot h(A_{\text{incrémentation}}) \cdot h(A_{\text{affectation}})\right)$$

(f) Hachage récursif cryptographique des sous-arbres

FIG. 9.2 – Calcul des valeurs de hachage de Karp-Rabin pour les sous-arbres d'un arbre

Pour des sous-arbres réduits à une feuille, la valeur de hachage $h(a)$ se confond avec celle de la valeur entière de a. On notera que l'on ne conservera qu'une partie des bits de la valeur de hachage pour le stockage des empreintes en base.

Parmi les fonctions de hachage cryptographiques testées, nous pouvons citer MD5 [55] et SHA-1 [51] qui sont obsolètes pour des applications cryptographiques pures mais s'avèrent assez rapides et peu sujettes à des collisions pour le hachage d'arbre de syntaxe.

9.1.3 Évaluation expérimentales de fonctions de hachage exact d'arbres

Évaluation de la fréquence des collisions

Afin d'évaluer la qualité de différentes fonctions de hachage, nous quantifions le nombre de collisions générées sur des arbres de taille diverses afin de les comparer aux collisions attendues par l'usage d'une fonction de hachage parfaite. p étant la probabilité de collisions théoriques pour une fonction de hachage parfaite pour une paire arbitraire d'arbres, le nombre de collisions attendues sur un ensemble de n arbres s'élève à $\frac{pn(n-1)}{2}$.

Nous réalisons des tests sur des arbres binaires complets générés aléatoirement avec l'utilisation d'un alphabet de $t_i = 10$ types pour les noeuds internes et $t_l = 10$ types pour les feuilles. En ne considérant que la structure des arbres, il existe $\mathcal{C}_{\lfloor n/2 \rfloor}$ arbres binaires complets de n noeuds (où $\mathcal{C}_k$ est le k-ième nombre de Catalan), la cardinalité étant multipliée par ${t_i}^{\lfloor n/2 \rfloor} + {t_l}^{\lfloor n/2 \rfloor + 1}$ si les types sont pris en comptes. Les fonctions de hachage comparées sont les fonctions de hachage polynomiales avec deux bases différentes 32 et 33. 33 est un nombre premier jugé empiriquement efficace et assez populaire pour le hachage polynomial tandis que la base 32 étant un diviseur des cardinalités des espaces de hachage utilisées est suspectée de générer de nombreuses collisions. Les deux fonctions de hachage cryptographiques choisies sont MD5 et SHA-1.

Nous notons que pour des petits sous-arbres, les fonctions de hachage comparées peuvent occasionner des collisions pour des espaces de sous-arbres de cardinalité inférieure à l'espace de hachage ; là où une fonction parfaite serait bijective. Lorsque l'espace des valeurs est de cardinalité largement inférieure à l'espace des sous-arbres (sous-arbres de volume plus important et/ou valeurs de faible longueur), le nombre de collisions relevées entre fonctions est très similaire et s'approche de la valeur théorique d'une fonction de hachage parfaite. En revanche, comme attendu, l'utilisation d'une valeur de base présentant un PGCD non unitaire avec la cardinalité de l'espace de hachage handicape le hachage de type Karp-Rabin.

Évaluation des performances temporelles de hachage

Nous cherchons à comparer le temps d'exécution de différentes méthodes de hachage proposées. Le hachage par fonction polynomiale étant incrémental, hacher un chaîne est réalisée en un temps proportionnel à sa longueur. Lorsqu'il s'agit d'un arbre, nous hachons une chaîne qui est la représentation sérialisée de celui-ci : des exponentiations de la base sont nécessaires (cf 8.1.2)) ; le hachage se déroule dans le pire des cas en temps $\Theta(N \log N)$ pour un arbre de

Collisions sur arbres de 7 nœuds ($\log_{10}(x)$)					
Longueur	KR ($B = 32$)	KR ($B = 33$)	MD5	SHA-1	Hachage parfait
55 bits	3,381 656	$-\infty$	$-\infty$	$-\infty$	$-\infty$
41 bits	6,190 223	$-\infty$	$-\infty$	0	$-\infty$
39 bits	6,472 682	0	$-\infty$	0	$-\infty$
38 bits	6,612 903	0,301 030	0	0,477 121	$-\infty$
32 bits	7,223 124	2,113 943	2,079 181	2,056 904	$-\infty$
16 bits	8,908 199	6,882 315	6,882 251	6,882 445	6,881 385
8 bits	10,175 686	9,290 793	9,290 747	9,290 726	9,290 726

Collisions sur arbres de 15 nœuds ($\log_{10}(x)$)					
Longueur	KR ($B = 32$)	KR ($B = 33$)	MD5	SHA-1	Hachage parfait
63 bits	2,143 015	$-\infty$	$-\infty$	$-\infty$	$-\infty$
41 bits	5,401 569	$-\infty$	$-\infty$	0	$-0{,}643\ 263$
39 bits	5,709 803	0,301 030	0,301 030	0,301 030	$-0{,}041\ 675$
32 bits	6,377 303	1,986 772	2,113 943	2,045 323	2,066 010
16 bits	8,879 466	6,882 563	6,882 547	6,882 554	6,882 490
8 bits	10,175 052	9,290 744	9,290 735	9,290 733	9,290 730

Collisions sur 10^6 arbres de 127 nœuds ($\log_{10}(x)$)					
Longueur	KR ($B = 32$)	KR ($B = 33$)	MD5	SHA-1	Hachage parfait
64 bits	1,698 970	$-\infty$	$-\infty$	$-\infty$	$-7{,}567\ 030$
40 bits	5,381 287	$-\infty$	$-\infty$	$-\infty$	$-0{,}341\ 989$
39 bits	5,418 374	0	0	0	$-0{,}041\ 436$
32 bits	6,321 192	2,041 393	2,037 426	2,037 426	2,064 458
24 bits	2,426 876	4,473 764	4,474 012	4,473 385	4,474 245
16 bits	8,878 016	6,882 354	6,882 908	6,882 375	6,882 490
8 bits	10,175 532	9,290 719	9,290 716	9,290 721	9,290 730

FIG. 9.3 – Évaluation du nombre de collisions sur un jeu d'arbres binaires aléatoire pour différentes fonctions de hachage exactes

Opération	Temps (UA)
Désérialisation de l'arbre de syntaxe	17,76
Indexation des empreintes	8,23
Hachage additif (structure agnostique)	0,127
Hachage polynomial en base 33	0,210
Hachage cryptographique MD5	0,231
Hachage cryptographique SHA-1	0,329

FIG. 9.4 – Temps d'exécution[a] de différentes méthodes de hachage d'arbre comparé à d'autres opérations

[a]Temps d'exécution expérimentaux pour le traitement du paquetage Netbeans-Javadoc par une version de développement de Plade. Une UA équivaut à 1 seconde en mono-fil avec le JRE Sun 1,6 32 bits sur un CPU Intel Pentium 4 3 Ghz (cache : 2 Mio, RAM : 1 Gio, 5985 bogomips).

N nœuds. En pratique cependant, contrairement au hachage par fonction cryptographique, celui-ci se révèle plus rapide pour des arbres de taille modeste régulièrement rencontrés comme arbres de syntaxes. Il convient toutefois de relativiser l'importance de la rapidité de hachage : si ces opérations sont multipliées par le nombre de profils d'abstraction utilisés, elles sont de coût négligeable rapportées aux opérations d'entrée-sortie de manipulation de base de sous-arbres indexés. Le hachage peut également être facilement parallélisé sur plusieurs processeurs, le calcul de valeurs de hachage de sous-arbres pouvant être mené indépendamment pour le hachage d'un arbre.

La figure 9.4 présente une évaluation de temps d'exécution pour les méthodes de hachage polynomiales ainsi que par fonction cryptographique MD5 et SHA-1. Nous constatons que le temps consacré au hachage est négligeable par rapport au coût temporel d'autres tâches telle que la désérialisation de l'arbre. Même si le hachage polynomial se révèle asymptotiquement moins avantageux, en pratique la taille limitée des sous-arbres manipulés lors du traitement d'arbres de syntaxe occasionne l'usage d'un nombre plus faible d'opérations que les fonctions cryptographiques. Considérant une dispersibilité presque équivalente des fonctions testées, les performances temporelles pratiques, la facilité d'implantation du hachage polynomial de type Karp-Rabin ainsi que sa propriété de hachage incrémental fort constituent des arguments en faveur de son utilisation.

9.2 Abstraction des arbres

9.2.1 Motivations

L'utilisation de métriques permet la création de classes d'équivalence pour les différentes unités structurelles hachées, classes qui ne reflètent pas toujours la similitude structurelle entre les arbres de syntaxes comparés mais plutôt une similitude portant sur des critères ensemblistes. Le hachage exact est lui totalement sensible à toute modification structurelle ce qui ne permet de l'utiliser que pour la recherche d'arbres de syntaxe exactement similaires. Afin de pallier à ce problème et réaliser des recherches de sous-arbres approchés, nous souhaitons introduire un niveau d'abstraction sur les sous-arbres et leur type. L'objectif est ainsi de représenter des sous-arbres relativement proches par une même représentation commune ce

qui garantirait alors une valeur de hachage similaire par un procédé de hachage exact.

9.2.2 Abstraction des types

Principe

L'abstraction des types de nœuds manipulés est déjà utilisée pour la production d'un arbre de syntaxe abstrait. Il s'agit ainsi de représenter des types de nœuds différents par un unique type. En pratique ceci peut être réalisé en attribuant, au sein du vecteur de valeurs de type, des valeurs identiques à plusieurs types.

Exemples

Parmi les choix d'abstraction de types envisageables, nous citons ici quelques exemples que nous avons déjà évoqués en 3.2.2. Certaines opérations d'obfuscation utilisant le remplacement de types par des supertypes, il peut être intéressant des les abstraire partiellement (conservation d'une hiérarchie réduite de types) ou totalement. Les identificateurs peuvent également être abstraits pour lutter contre leur renommage : conserver un profil d'abstraction avec identificateurs non abstraits peut néanmoins demeurer utile pour le dépistage de copies exactes.

D'autres catégories d'abstraction peuvent être réalisées, quoique celles-ci soient moins efficaces qu'un procédé, plus coûteux sémantiquement, de normalisation des arbres de syntaxes. Ainsi des types de boucle peuvent être abstraits mais il faut noter que l'obfuscation par la transformation d'un type de boucle en une autre implique plus de modifications que la simple substitution du type de boucle. Certains opérateurs peuvent également faire l'objet d'une abstraction.

Une abstraction totale des types peut également être réalisée : elle utilise un vecteur de valeurs de type uniforme. Cette abstraction conduit à l'obtention d'une valeur de hachage ne dépendant que de la structure de l'arbre haché.

9.2.3 Abstraction de sous-arbres élémentaires

Abstraire de petits sous-arbres par l'utilisation d'une valeur de hachage commune présente un intérêt pour conserver une valeur de hachage identique pour des arbres n'ayant fait l'object que d'opérations d'éditions locales n'affectant la valeur de hachage que de sous-arbres élémentaires. Si l'abstraction considère comme élémentaires des sous-arbres de taille relativement conséquente, celle-ci permet la recherche de similitudes sur des clones creux (cf 2.1.2) afin de détecter potentiellement certains patrons de conception. Un sous-arbre élémentaire est un sous-arbre qui peut formellement être défini de plusieurs manières :

1. Il peut s'agir d'un sous-arbre dont la racine est d'un type particulier. Ainsi par exemple, des sous-arbres représentant une expression, une instruction voire même le corps d'une fonction entière peuvent être définis comme élémentaires afin d'obtenir plusieurs niveaux d'abstraction.
2. Un sous-arbre élémentaire peut être défini par un seuil de hauteur maximale. Cependant il peut exister des sous-arbres de hauteur faible mais couvrant un volume important du

code source avec un nombre de nœuds conséquent : si seules des opérations d'édition locales doivent être ignorées, le seul critère de hauteur peut être inadapté.

3. Ainsi nous pouvons également définir un sous-arbre élémentaire par un seuil de nombre de nœuds maximal.

9.2.4 Effacement de feuilles et sous-arbres

Effacement complet d'un sous-arbre

Certains sous-arbres des arbres de syntaxe manipulés n'ont quelquefois pas de véritable intérêt sémantique pour la recherche de similitudes. Dans cette catégorie, nous pouvons situer les feuilles liées à l'expression de modificateurs (dont les mots-clés de visibilité d'entités), des sous-arbres liés à des commentaires, à des opérations d'importation de structures externes ou des sous-arbres très fréquents témoignant de code idiomatique. Ces sous-arbres peuvent être aisément supprimés par obfuscation : un procédé de hachage des arbres doit donc de préférence les ignorer.

Effacement d'une racine d'un sous-arbre

Dans certains cas, il peut être utile de supprimer le nœud racine d'un sous-arbre C_i ayant pour parent l'arbre A. Les sous-arbres enfants de C_i sont alors reconnectés directement en tant qu'enfants de leur ex-grand-parent A. Cette modification peut être utile afin de supprimer des structures de contrôle de gestion d'exception ou de délimitation de zones de synchronisation. À l'extrême, il peut être envisagé d'applatir le code en supprimant toutes les structures de contrôle afin de contrer des procédés d'obfuscation liés à la modification de structures de contrôle.

9.2.5 Normalisation de l'ordre des sous-arbres enfants

L'ordre des sous-arbres enfants de certains types de nœuds ne présente pas d'importance. Cette remarque est valable pour les opérateurs commutatifs où l'ordre des opérandes enfants peut être modifié ainsi que pour certaines unités structurelles telles que les unités de compilation ou les classes où, pour certains langages, l'ordre des déclarations de fonctions et de variable peut être modifié sans conséquence sémantique. Les valeurs de hachage obtenues pour deux arbres dont l'ensemble des sous-arbres enfants est identique mais présenté dans un ordre différents ne sont pas égales sauf collision accidentelle. Nous pouvons alors normaliser l'ordre des sous-arbres enfants des nœuds dont le type est commutatif, par exemple en les ordonnant par valeur de hachage croissante.

On note que l'on pourrait normaliser l'ordre des instructions d'un bloc de code après transformation de l'arbre de syntaxe par regroupement des instructions par blocs indépendants pouvant ensuite être transposés.

La généralisation de ce procédé à tous les nœuds permet d'obtenir des classes d'équivalence d'arbres de syntaxe non-ordonnés. Deux arbres de syntaxe non-ordonnés identiques ne caractérisent pas nécessairement deux portions de code source pouvant être considérées comme similaires.