Indexation et recherche de sous-arbres

Sommaire

10.1 Indexation de sous-arbres par empreinte unique 166
10.1.1 De la longueur des empreintes et de la duplication de sous-arbres . . 166
10.1.2 Sélection des empreintes à indexer 167
10.1.3 Table de classes d'équivalence et ensembles de membres de classe . . 168
10.1.4 Sérialisation des sous-arbres 169
10.1.5 Algorithme d'indexation 169
Détermination des classes d'équivalence de sous-arbres 170
Résolution des collisions 172
Indexation 173
Complexité 173
10.1.6 Exemple 174
10.2 Familles de classe d'équivalence 175
10.2.1 Famille de classes d'équivalence 175
10.2.2 Graphe de familles de classe d'équivalence 176
Graphe 176
Un exemple de graphe de familles de classes d'équivalence 177
10.3 Indexation selon un graphe de familles d'équivalence 178
10.3.1 Structures d'indexation 178
10.3.2 Procédure d'indexation 178
10.3.3 Implantation des structures d'indexation 179
10.3.4 Exemple d'indexation 180
10.4 Recherche de similarité sur arbre requête 181
10.4.1 Problématique 181
10.4.2 Recherche de sous-arbres individuels égaux 182
10.4.3 Recherche par hachage de chaînes de sous-arbres égaux 182
Transformation des arbres n-aires en arbres binaires 182
Hachage exhaustif de sous-chaînes 182
10.4.4 Détermination des farmax sur chaînes de sous-arbres frères 183

Principe général 183
Support d'opérations d'abstraction 184
10.4.5 Quantification de l'exactitude de paires de chaînes d'arbres correspondantes 185
Distance d'édition sur les arbres 185
10.5 Évaluation de quelques familles d'abstraction 187
10.6 Limitations de l'indexation par profil 194

Nous avons étudié au cours du chapitre précédent différentes méthodes de hachage d'arbres de syntaxe ainsi que plusieurs stratégies d'abstraction afin de pouvoir adopter la recherche de similarité à un niveau de détail plus ou moins important. Au-delà de l'obtention des valeurs de hachage pour différents profils d'abstraction se pose la question de leur stockage en vue de rechercher des sous-arbres similaires sur un ensemble conséquent d'unités structurelles de code. À cet effet, nous proposons ici une méthodologie pour l'indexation de différentes valeurs de hachage liées à des sous-arbres hachés pour des profils d'abstraction différents. Cette indexation permet le regroupement des sous-arbres par classes d'équivalence selon différents critères de similarité. L'étude des sous-arbres similaires internes à un ensemble de projets peut ainsi être réalisée par exploitation directe d'une base de classes d'équivalence obtenue après l'indexation de ces projets. Une unité structurelle externe requête peut également être confrontée, après calcul de ses empreintes, à une base afin de déterminer les sous-arbres similaires entre l'unité requête et les projets indexés de la base.

10.1 Indexation de sous-arbres par empreinte unique

Dans un premier temps, nous souhaitons représenter chaque sous-arbre par une empreinte unique correspondant à un profil d'abstraction fixé. Un procédé classique d'indexation implique la maintenance d'une table d'association entre empreintes de hachage et identificateur du sous-arbre correspondant. Les clés de cette table peuvent être indexés par une structure de B+-k-tree, arbre k-aire adapté au stockage de masse limitant les opérations d'entrée-sortie à $\Theta(\log_k(N))$ opérations pour la recherche ou l'ajout d'une empreinte pour N empreintes déjà indexées.

10.1.1 De la longueur des empreintes et de la duplication de sous-arbres

Longueur des empreintes Le principal écueil réside dans la difficulté à choisir une longueur d'empreinte adéquate. En supposant la fonction de hachage sous-jacente parfaite et l'espace des sous-arbres de cardinalité infinie, la probabilité que deux sous-arbres représentés par la même valeur de hachage de longueur k bits soit différents est de $\frac{1}{2^k}$. La probabilité de l'existence d'au moins une collision accidentelle sur une base de N empreintes ($N \leq 2^k$) s'élève à $p_c(k, N) = 1 - \frac{2^k(2^k-1)\cdots(2^k-N+1)}{2^{NK}}$. En supposant que chaque ligne de code soit représentée par 5 empreintes en moyenne[1], la probabilité de l'existence d'une collision accidentelle pour certains projets avec différentes longueurs de valeurs de hachage parfaites est exprimée en

[1]Il s'agit d'une estimation approximative, la densité d'empreintes par ligne de code dépendant du langage, du style de programmation et du seuil de volume minimal pour la conservation d'une empreinte. À titre d'exemple, le paquetage *netbeans-javadoc* (14 360 lignes) possède un volume cumulé d'arbres syntaxiques de 74 351 nœuds (soit 5,20 nœuds par ligne) ; 11 528 empreintes portent sur des sous-arbres d'au moins 20 nœuds).

Lignes de code	thttpd 2.25 11,1K	Apache httpd 2.2 341K	Noyau Linux 2.6.33 11,2M	Projets SourceForge $\sim$ 1G
$k = 32$	$0,30$	$1 - \epsilon$	$1 - \epsilon$	1
$k = 64$	$8,4 \cdot 10^{-11}$	$7,9 \cdot 10^{-8}$	$8,5 \cdot 10^{-5}$	$0,49$
$k = 80$	$1,2 \cdot 10^{-15}$	$1,2 \cdot 10^{-12}$	$1,3 \cdot 10^{-9}$	$1,0 \cdot 10^{-5}$

FIG. 10.1 – Estimation des probabilités de l'existence d'au moins une collision accidentelle d'empreintes de k bits pour différents projets

figure 10.1. Une longueur de valeur de hachage confortable peut rendre négligeable le risque de collision accidentelle et dispense ainsi d'une vérification approfondie de l'égalité de sous-arbres hachés identiquement. Cependant un souci d'économie de mémoire de masse peut nous conduire à limiter cette longueur : un compromis doit être réalisé. Nous proposons donc de choisir une longueur de valeur de hachage faible au démarrage de la constitution de la base et de vérifier incrémentalement, plutôt qu'*a posteriori* lors d'interrogations, la présence de collisions accidentelles. Dans ce cas, nous augmentons la longueur des nouvelles empreintes. La similarité des sous-arbres référencés en base par une valeur de hachage commune et garantie : une telle valeur définit ainsi une classe d'équivalence de sous-arbres selon le profil choisi.

10.1.2 Sélection des empreintes à indexer

Une unité de compilation est généralement représentée par des arbres de syntaxe de grand volume : indexer chacun des nœuds correspondant à un sous-arbre peut s'avérer peu pertinent. Un compromis doit être adopté quant au volume minimal des sous-arbres à indexer. Un volume trop faible conduira à l'indexation de petits sous-arbres comprenant de multiples occurrences, peu utiles pour la mise en évidence de similarités intéressantes s'inscrivant au-delà de clones idiomatiques. Quant à un volume trop élevé, il ne peut permettre la localisation que des clones les plus massifs, laissant masqués des clones plus modestes. L'adoption de certaines techniques d'abstractions comme l'abstraction de petits sous-arbres d'expression peut permettre d'augmenter le seuil de volume minimal d'indexation.

Lorsqu'un volume seuil d'indexation est fixé, les sous-arbres possédant un grand nombre de sous-arbres enfants de volume faible peuvent être eux-mêmes indexés mais pas leurs enfants. Cette situation est rencontrée pour de gros blocs d'instructions où chaque instruction prise individuellement n'est pas indexée mais le bloc entier l'est sous la forme d'une unique empreinte. La granularité d'indexation est alors trop importante. Nous introduisons alors une indexation des sous-arbres enfants sur fenêtre de taille variable : il s'agit d'une généralisation de l'indexation sur fenêtre de taille 1 utilisée jusqu'ici. Pour chaque sous-arbre d'une fratrie, nous démarrons une fenêtre d'indexation. Si le sous-arbre est de volume supérieur au seuil d'indexation, une empreinte le concernant est créée et indexée (fenêtre de taille 1). Dans le cas contraire, la fenêtre d'indexation est étendue aux sous-arbres frères à droite jusqu'à ce que celle-ci englobe une séquence de sous-arbres dont le volume dépasse le seuil d'indexation. Ainsi, l'ensemble d'une fratrie de sous-arbres est couvrable par des empreintes indexées, à l'exception éventuelle des sous-arbres les plus à droite dont le volume cumulé est inférieur au seuil d'indexation[2]. On notera que des empreintes peuvent concerner des séquences chevau-

[2]On pourra, afin de couvrir l'ensemble de la fratrie, ajouter en surplus de l'empreinte concernant la fenêtre la plus à droite une nouvelle empreinte débutant au mêmes sous-arbre mais s'étendant jusqu'à la fin de la

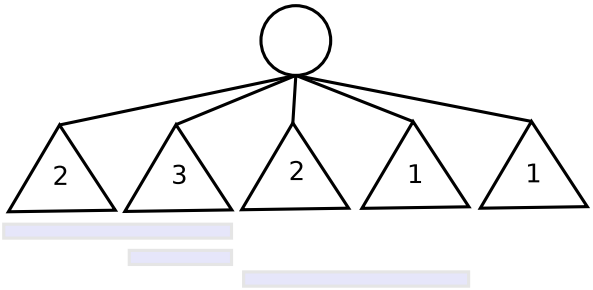


FIG. 10.2 – Indexation sur fenêtre d'une fratrie avec un seuil de volume de 3

chantes de sous-arbres. La figure 10.2 illustre l'indexation d'une fratrie (le dernier frère n'est pas indexé et deux empreintes chevauchent le deuxième frère).

Duplication de sous-arbres Deux sous-arbres $A(a_1, a_2, \cdots, a_k)$ et $A'(a'_1, a'_2, \cdots, a'_k)$ identiques ne doivent pas cacher la forêt de sous-arbres n'ayant pas pour parents A ou A' et pourtant identiques avec un des sous-arbres enfants de A ou A'. Toutefois si les sous-arbres $a_1 = a'_1, a_2 = a'_2, \cdots, a_k = a'_k$ ne sont partagés que par A et A', il paraît superflu de les mentionner explicitement dans la base d'indexation. Nous pourrions ainsi associer leurs valeurs de hachage à un pointeur vers la classe d'équivalence de leur parent (classe contenant A et A') avec leur place dans la fratrie de nœuds. Ainsi, si un nouveau sous-arbre A'' est indexé, celui-ci étant égal à A et A' et rejoignant ainsi leur classe d'équivalence, aucune opération d'indexation n'est nécessaire pour les descendants de A''.

10.1.3 Table de classes d'équivalence et ensembles de membres de classe

Une classe d'équivalence identifie l'ensemble des sous-arbres égaux selon le profil choisi. Une condition nécessaire à l'appartenance de sous-arbres à une même classe réside dans le partage d'une même valeur de hachage. Elle n'est pas suffisante en raison de possibles collisions accidentelles : on considère toutefois l'égalité des sous-arbres acquise lorsque la longueur des valeurs est suffisamment importante.

Au sein d'une structure de table de classes d'équivalence nous associons valeurs de hachage avec un identifiant de classe. Les valeurs de hachage peuvent être de longueur variable ; toutefois il n'existe pas de classe dont la valeur de hachage est préfixe de celle d'une autre classe, ces valeurs représentant donc un code préfixe.

Chaque classe est identifiée par un entier séquentiel. Nous associons à chacune de ces classes l'ensemble de ses membres. Un sous-arbre membre peut être explicité de deux manières :

1. Soit par un pointeur vers lui-même (explicitation directe). Il s'agit généralement d'expliciter l'identificateur de l'arbre englobant d'appartenance, ainsi que la place du sous-arbre membre lors d'un parcours en largeur de l'arbre englobant.

chaîne de sous-arbres.

2. Soit par l'identifiant de la classe d'équivalence de son arbre parent avec spécification de la place du sous-arbre pointé dans sa fratrie (explicitation par classe du parent). Cette solution est naturellement exclue pour les racines d'arbres n'ayant pas de parent.

Nous pouvons récupérer l'ensemble des membres d'une classe par obtention des pointeurs d'explicitation directe et en récupérant récursivement les parents membres dans le second cas pour en déduire par leur rang les enfants membres. Il est toutefois nécessaire de disposer d'une représentation de l'arbre englobant afin de déterminer l'identificateur du sous-arbre à partir de celui de son parent et de sa position dans la fratrie. En raison de l'indirection requise, nous réservons l'explicitation par classe du parent au cas où celle-ci permet une économie mémorielle : c'est uniquement le cas lorsque la classe d'équivalence du parent contient au moins deux membres, ce qui évite la spécification de deux pointeurs pour chacun de leurs sous-arbres respectifs.

10.1.4 Sérialisation des sous-arbres

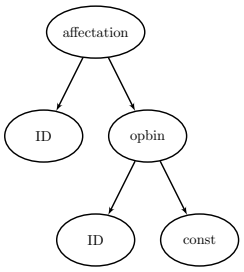
Il est utile de sérialiser les sous-arbres indexés, parallèlement aux tables de classes et de membres. L'objectif est de pouvoir récupérer l'intégralité des nœuds d'un sous-arbre représentatif d'une classe sans disposer d'un référentiel externe ou avoir à re-analyser syntaxiquement une unité de compilation. La forme sérialisée est utilisée pour obtenir explicitement des membres de classe désignés par leur parent et leur rang dans la fratrie ainsi que pour rehacher des sous-arbres en cas de collision accidentelle.

Deux types d'information sont essentielles pour la reconstitution d'un arbre de syntaxe : le type de chacun de ses nœuds (représentable par une valeur entière) et les relations de parenté entre eux. À cet effet, on pourra représenter un arbre sous la forme d'une liste de tuples où le tuple d'indice i explicite l'arbre englobant d'appartenance, le rang du parent ainsi que le rang et le type du nœud racine du sous-arbre de rang i. Le rang d'un nœud est défini par son ordre d'accès lors d'un parcours en largeur. Ainsi les nœuds d'une même fratrie possèdent des rangs consécutifs et le nœud parent d'un nœud est de rang plus bas.

Nous présentons par exemple en figure 10.3 une représentation sérialisée d'une instruction simple. Cette représentation autorise un accès direct aux informations du nœud de rang i ainsi qu'une récupération en temps logarithmique par dichotomie des indices des enfants d'un nœud (les tuples étant triés par rang croissant de sous-arbre parent). Dès lors, la désérialisation d'un sous-arbre peut être menée de façon paresseuse en accédant dans un premier temps à sa racine, puis en récupérant à la demande, niveau par niveau, ses descendants. Nous adjoignons aux informations de type et de parenté un lien de retour vers le code source original afin de pouvoir y localiser des correspondances définies par similarité de sous-arbre de syntaxe.

10.1.5 Algorithme d'indexation

L'algorithme 5 résume le processus d'indexation en base d'un arbre en utilisant les structures précédemment décrites. Celui-ci se décompose en deux étapes fondamentales. La première consiste à déterminer la classe d'équivalence d'appartenance des sous-arbres par leur valeur de hachage. Cette classe étant trouvée ou créée, s'il s'agit du premier exemplaire de ce sous-arbre, nous déterminons dans un second temps si le parent de ce sous-arbre existe déjà en plusieurs exemplaires en base pour adopter son type de spécification dans le classe d'équivalence.



Arbre	Rang racine	Rang parent	Type	Retour
A	1		affectation	k
	2	1	ID	k
	3	1	opbin	$[k..k+1]$
	4	3	ID	$k+1$
	5	3	const	$k+1$

FIG. 10.3 – Forme sérialisée du sous-arbre de syntaxe correspondant à l'instruction `somme = \n somme + 1 ;` (lignes $k \rightarrow k+1$ du code source)

Dans un souci d'économie mémorielle, chaque classe d'équivalence de la base est représentée par une valeur de hachage réduite de longueur variable. Lors de la création d'une nouvelle classe d'équivalence, cette longueur est choisie de telle sorte que la valeur de hachage réduite de la classe ne soit pas préfixe d'une valeur d'une autre classe.

Nous décrivons maintenant plus en détails le processus d'indexation d'un arbre A. Tout d'abord des valeurs de hachage longues sont calculées pour chacun des sous-arbres et sont associées à leur racine : celles-ci peuvent être obtenues en temps linéaire en nombre de nœuds en démarrant des feuilles en remontant vers la racine de l'arbre global comme décrit au chapitre précédent.

Détermination des classes d'équivalence de sous-arbres

Afin de pouvoir indexer chacun des sous-arbres, il est nécessaire de déterminer leur classe d'équivalence d'appartenance, voire d'en créer une nouvelle lorsqu'aucun exemplaire de ce sous-arbre n'existe en base. Pour déterminer la classe d'un sous-arbre $A[k]$ de valeur de hachage longue $\mathcal{H}(A[k])$, on recherche une valeur de hachage réduite dans la table des classes qui soit préfixe de $\mathcal{H}(A[k])$. Soit celle-ci n'est pas trouvée ce qui garantit qu'aucun exemplaire de ce sous-arbre n'a encore été indexé : la création d'une nouvelle classe sera donc nécessaire. Soit une classe est trouvée et il est nécessaire de vérifier l'appartenance réelle de $A[k]$ à celle-ci.

La détermination de classes d'équivalence est réalisée du plus petit sous-arbre de A (feuilles ou plus petits sous-arbres à indexer) au plus grand sous-arbre (A lui-même). Nous cherchons à spécifier pour chaque sous-arbre sa classe d'équivalence ou si celui-ci doit être ajouté dans une nouvelle classe. À cette fin, nous nous basons sur le fait que deux sous-arbres appartiennent à la même classe ssi les trois propriétés suivantes sont respectées :

- leur type de nœud est considéré comme similaire ;
- ils disposent du même nombre de sous-arbres enfants ;
- et les enfants de même rang appartiennent aux mêmes classes d'équivalence.

Ainsi, lorsqu'une classe d'équivalence candidate C_i pour accueillir $A[k]$ est trouvée, nous en extrayons un sous-arbre membre que nous notons r et déterminons si son type de nœud racine est similaire et si ses enfants appartiennent bien au mêmes classes d'équivalence que les enfants de $A[k]$. L'appartenance à une classe d'équivalence étant basée sur une relation de simi-

Données : Ensemble incrémental des classes d'équivalence et de leurs membres $\mathcal{C}$
Données : Arbre A à indexer
début
  Détermination des classes d'équivalence d'appartenance ;
  pour *chaque sous-arbre $A[k]$ de A du plus petit au plus grand (parcours en largeur inverse)* **faire**
    $\mathcal{H}(A[k]) \longleftarrow$ calcul de valeur de hachage longue de $A[k]$ (à partir des valeurs de ses enfants) ;
    $\mathcal{C}_i \longleftarrow$ classe d'équivalence de valeur de hachage préfixe de $\mathcal{H}(A)$;
    $a \longleftarrow$ faux ;
    si $\mathcal{C}_i \neq \emptyset$ **alors**
      $a \longleftarrow$ vérification de non-collision avec un membre indexé de $\mathcal{C}_i$;
      si $\neg a$ **alors**
        Augmentation de la longueur de la valeur de hachage de $\mathcal{C}_i$;
    si $\neg a$ **alors**
      $\mathcal{C}_i \longleftarrow$ nouvelle classe d'équivalence créée ;
  Ajout des sous-arbres comme membres des classes ;
  pour *chaque sous-arbre $A[k]$ de A du plus grand au plus petit (parcours en largeur)* **faire**
    ζ est la cardinalité de la classe d'équivalence $\mathcal{C}_j$ de $\mathcal{P}(A[k])$ ($\zeta \geq 1$) ou 0 si $A[k]$ est la racine ;
    si $\zeta = 0 \vee \zeta = 1$ **alors**
      Spécification explicite d'un pointeur vers α_k ;
    si $\zeta = 2$ **alors**
      Suppression du seul membre explicite de $\mathcal{C}_i$;
      Spécification de la classe du parent $\mathcal{C}_j$ et du rang ξ ;
    si $\zeta > 2$ **alors**
      La classe du parent $\mathcal{C}_j$ et le rang ξ sont déjà mentionnés ;
      Arrêt de l'indexation pour $A[k]$ et ses descendants ;
    $\zeta \longleftarrow \zeta + 1$;
fin

Algorithme 5 : Algorithme d'indexation

larité transitive, la vérification de similarité de $A[k]$ avec un membre quelconque représentant de C_i suffit à démontrer l'appartenance de $A[k]$ à C_i.

Si r est spécifié indirectement par référence à sa classe d'équivalence parent (r représente en fait plusieurs occurrences de sous-arbres similaires), il en va de même de ses enfants $r_1, \cdots, r_n$ mentionnés par lien vers la classe d'équivalence C_i : on recherche ces mentions sur les classes d'équivalence validées de $A[k]_1, \cdots, A[k]_n$.

Si r est spécifié explicitement par mention de son arbre d'appartenance (R) et de son rang (j), alors nous recherchons sur les classes d'équivalence de $A[k]_1, \cdots, A[k]_n$ les enfants de $R[j]$. Cela nécessite de connaître les rangs des enfants de $R[j]$ dans R par désérialisation paresseuse de R.

Résolution des collisions

Pour chacun des sous-arbres $A[k]$ distincts de A à indexer, l'étape précédente a permis de trouver ou non une classe d'équivalence candidate de valeur préfixe de $\mathcal{H}(A_i)$. Si une classe candidate a été trouvée, son adéquation à accueillir $A[k]$ a été vérifiée.

Si aucune classe n'est trouvée, une nouvelle classe doit être créée. Nous choisissons une valeur de hachage réduite pour la représenter de longueur au minimum égale à la plus longue valeur déjà présente en base (la longueur courante l) ; dans le cas contraire, cette valeur réduite pourrait être préfixe d'une autre valeur en base. Cette valeur est mise en correspondance avec un identificateur séquentiel s créé pour la classe et l'on indique l'appartenance de $A[k]$ à s.

Si une classe de valeur de hachage préfixe de $A[k]$ est trouvée et que ses membres (dont un membre r) sont égaux à $A[k]$, $A[k]$ peut intégrer cette classe d'équivalence. Dans le cas contraire, une nouvelle classe doit être créée avec $A[k]$ et la classe de r doit voir sa valeur de hachage représentative allongée pour éviter une collision avec cette nouvelle classe. À cet effet le membre représentatif r fait l'objet d'une opération de rehachage (après désérialisation) pour réobtenir sa valeur de hachage longue afin de compléter la valeur de hachage représentative de la classe.

La longueur des valeurs de hachage représentatives de $A[k]$ et r doit être :
- au minimum égale à la longueur courante l ;
- et plus grande que la longueur du préfixe commun à $\mathcal{H}(A[k])$ et $\mathcal{H}(r)$ afin de pouvoir distinguer les deux classes.

À nombre de classes constant, augmenter la longueur de valeur de hachage de α bits équivaut approximativement à diviser par $2^{\alpha/2}$ la probabilité de l'existence de collision. D'un autre point de vue si une collision accidentelle est rencontrée sur une valeur de hachage de longueur l avec C classes distinctes en bases, allonger sa valeur à $l + \alpha$ bits conduira à une nouvelle collision accidentelle pour un ordre de grandeur de $2^{\alpha/2}C$ classes présentes en base. Le choix de la longueur d'extension α conditionne la fréquence nécessaire de rehachage de représentants de classe.

Indexation

L'indexation des sous-arbres est réalisée par parcours en largeur du plus grand sous-arbre au plus petit sous-arbre de A. Afin de décider de mentionner explicitement le sous-arbre membre $A[k]$ ou plutôt un lien vers la classe d'équivalence C_j de son parent $\mathcal{P}(A[k])$, nous examinons la cardinalité de la classe de C_j (préalablement connue car $\mathcal{P}(A[k])$ a été antérieurement indexé) notée ζ. Quatre situations peuvent être rencontrées en fonction de la cardinalité de C_j :

1. Cas où $A[k]$ est la racine de l'arbre ($A[1]$) et ne possède donc pas de parent. Nous ajoutons alors explicitement un pointeur vers $A[k]$ dans la classe C_i.
2. Cas $\zeta = 1$. Dans cette situation la classe C_j ne contient que le parent $\mathcal{P}(A[k])$ antérieurement indexé. Comme pour le cas précédent, nous ajoutons un pointeur vers $A[k]$ dans C_i.
3. Cas $\zeta = 2$. Avant l'indexation de $\mathcal{P}(A[k])$, C_j était de cardinalité unitaire (avec la mention explicite d'un membre $\mathcal{P}(e)$) ce qui signifie que C_i contient au moins par référence explicite le sous-arbre e. Avec l'indexation de l'arbre A, C_j perd sa cardinalité unitaire et contient deux membres : il n'est donc plus utile de mentionner explicitement sur C_i les enfants de ses membres. Le sous-arbre e explicité sur C_i est donc supprimé et remplacé par la spécification de la classe d'équivalence parent C_j et du rang dans la fratrie, spécification englobant également le nouveau sous-arbre $A[k]$.
4. Cas $\zeta \geq 3$. Lors de l'indexation antérieure d'une occurrence de sous-arbre similaire à $A[k]$ appartenant à C_i, une mention vers la classe d'équivalence parent et le rang fraternel a déjà été réalisée. Aucune action n'est nécessaire pour indexer $A[k]$ déjà présent implicitement par la mention explicite en base de son plus proche ancêtre de parent non-dupliqué.

Complexité

Les opérations d'accès, d'ajout et de suppression d'une classe peuvent être menées en temps logarithmique du nombre de classe $\log |C|$ par un arbre d'indexation. Le hachage d'un arbre A nécessite de s'interroger sur l'appartenance de chacun de ses sous-arbres $A[k]$ à une classe d'équivalence. Dans le pire des cas, il peut être nécessaire de rechercher tous les enfants d'un représentant de la classe d'équivalence candidate dans les classes d'équivalence des enfants de $A[k]$ ce qui est réalisé en $\Theta(\log P)$ où P est le nombre maximal de pointeurs d'une classe d'équivalence. Il faut y ajouter le temps de désérialisation nécessaire à l'obtention du rang des enfants du représentant en $\Theta(\log N)$ où N est le nombre de sous-arbres indexés en base. Globalement, le processus d'indexation pour A requiert un temps en $\Theta(|A| \log(|C|PN))$.

Nous avons toutefois omis de discuter du coût lié au rehachage. Celui-ci peut être évité par l'emploi, dès la création de la base, d'une longueur de valeur de hachage suffisamment longue au prix d'un coût mémoriel plus important. Cette longueur initiale ainsi que le nombre de bits ajoutés par rehachage α conditionne le nombre de rehachage prévisible. Le coût asymptotique de rehachage amorti pour $|C|$ classes d'équivalence est équivalent à $\frac{|C| \log_2 |C|}{\alpha/2} t_d$ où t_d est le coût de désérialisation moyen d'un sous-arbre.

10.1.6 Exemple

Afin d'illustrer le principe d'indexation précédemment décrit, nous introduisons un petit exemple par le hachage et l'indexation des arbres A et B suivants où a et b sont deux types de nœuds utilisés :

$$\begin{aligned} A = &\quad a(b,a) \\ B = &\quad b\,(a(b,a), a(b,a), a) \end{aligned}$$

A est composé de trois sous-arbres b, a et l'arbre complet $a(b,a)$. Nous leur assignons des valeurs de hachage longues (arbitraires pour l'exemple) de 4 bits : $\mathcal{H}(b) = 1100$, $\mathcal{H}(a) = 0100$ et $\mathcal{H}(a(b,a)) = 1010$. La longueur initiale de valeur de hachage réduite est fixée à 2 : aucune collision n'est alors relevée pour les valeurs réduites des sous-arbres de A, le plus long préfixe commun étant de longueur 1. Nous obtenons la table de classes et la table de membres de la figure 10.4

Valeur de hachage	Classe	Sous-arbre	Membres explicites
10	1	$a(b,a)$	$A[1]$
11	2	b	$A[2]$
01	3	a	$A[3]$

FIG. 10.4 – Membres des classes après indexation de $A = a(b,a)$

Nous indexons maintenant B qui comporte 4 sous-arbres distincts dont nous spécifions les valeurs de hachage longues : $\mathcal{H}(a) = 0100$, $\mathcal{H}(b) = 1100$, $\mathcal{H}(a(b,a)) = 1010$ (nous utilisons une abstraction ignorant l'ordre des enfants d'un sous-arbre) et $\mathcal{H}(B = b(a(b,a), a(b,a), a)) = 1001$. Pour chacune de ces valeurs, nous vérifions s'il existe une classe sur la base de valeur de hachage préfixe : si celle-ci existe, nous déterminons si la classe est adaptée ou s'il s'agit d'une collision accidentelle. Pour a et b les classes de valeurs de hachage préfixes 11 et 01 existent et sont adéquates. Pour le sous-arbre $a(b,a)$ de valeur 1010, la classe de valeur préfixe trouvée est celle d'identifiant 1 avec le préfixe commun 10 : son unique membre a les mêmes sous-arbres enfants b et a. Nous pouvons ajouter les deux sous-arbres $a(b,a)$ de B à cette classe. Enfin pour $\mathcal{H}(\mathcal{B})$, nous relevons la classe d'équivalence 1 ayant également pour valeur de hachage préfixe 10. Les trois sous-arbres enfants de B ne correspondent cependant pas aux deux sous-arbres enfants b et a d'un membre de la classe 1, $A[1] = a(b,a)$. Une collision accidentelle est relevée : elle nécessite un rehachage de la valeur de hachage de la classe 1 en 101 et la création d'une nouvelle classe pour ajouter B avec pour valeur 100.

Lorsque nous ajoutons les occurrences de sous-arbres de B comme membres des classes d'équivalence adéquates de la base, nous explorons les sous-arbres de B du plus grand au plus petit. B est d'abord ajouté dans une classe nouvelle. Les deux occurrences de $a(b,a)$ ont pour parent commun le sur-arbre B présent en unique exemplaire dans la base : ils sont donc ajoutés explicitement.

Concernant le sous-arbre b, lors de l'ajout de sa première occurrence dans B nous constatons que la classe d'équivalence de son parent (A) contient déjà deux membres, $A[1]$ ainsi que le

premier sous-arbre $a(b, a)$ $B[2]$ de B. Nous supprimons donc la mention explicite vers l'enfant $A[2]$ de $A[1]$ et ajoutons une référence vers la classe d'équivalence parent 1 et le rang fraternel (1). Lors de la rencontre de la seconde occurrence de b dans B également enfant de $a(b, a)$ ($B[3]$) de classe d'équivalence parent de cardinalité désormais 3, aucune opération d'indexation n'est réalisée ; $B[3]$ étant référencé explicitement dans la classe d'équivalence 1.

Les occurrences du sous-arbre a dans B sont indexés analoguement aux occurrences de b. Nous obtenons finalement les tables de classe d'équivalence et de membres spécifiées ci-après en figure 10.5 :

Hachage	Classe	Sous-arbre	Membres explicites	Membres par classe du parent
101	1	$a(b, a)$	$A[1], B[2], B[3]$	
11	2	b		$(1, 1)$
01	3	a	$B[3]$	$(1, 2)$
100	4	$b(a(b, a), a(b, a), a)$	$B[1]$	

FIG. 10.5 – Membres des classes après indexation de $A = a(b, a)$ et $B = b(a(b, a), a(b, a), a)$

10.2 Familles de classe d'équivalence

Plutôt que d'être contraint à la recherche de similitudes avec un niveau immuable d'abstraction, il pourrait être utile d'indexer les sous-arbres des arbres de syntaxe selon plusieurs abstractions afin de proposer des critères de recherche de similarité plus flexibles. Une première méthode consiste à conserver des bases de valeurs de hachage indépendantes pour chaque abstraction. Dans cette optique, nous pourrions par exemple stocker pour chaque sous-arbre une empreinte pour un profil abstrayant les identificateurs, un second les types et les identificateurs, un troisième ajoutant une abstraction des sous-arbres d'expression de taille inférieure à un seuil... Cette première approche, bien que fonctionnelle, induit l'apparition d'informations redondantes et ne permet pas directement de déduire des relations d'inclusion entre classes d'équivalence de différents profils.

Proposer des méthodes de génération d'empreintes utilisant uniquement des propriétés caractéristiques ensemblistes de l'arbre manipulé pourrait s'avérer utile. Ces empreintes peuvent être obtenues par hachage de vecteurs caractéristiques de chaque sous-arbre. L'objectif n'est cependant ici pas de proposer des valeurs de hachage approchées dont la proximité signifierait une certaine similarité des sous-arbres qu'elles représentent. Il s'agit plutôt d'utiliser l'égalité exacte de telles valeurs afin de déduire des sous-classes d'équivalence plus précises entre arbres appartenant à une même classe d'équivalence plus générale. On pourra par exemple adjoindre à un profil abstrayant les types, identificateurs et commentaires des valeurs de hachage sur des vecteurs d'existence de types, d'identificateurs et de mots de commentaire.

10.2.1 Famille de classes d'équivalence

Définition 10.1. Famille de classes d'équivalence. Une famille de classes d'équivalence est un ensemble de classes d'équivalence contenant des sous-arbres définie par une fonction booléenne

transitive f d'égalité de sous-arbres. Pour tout couple de sous-arbres (A, B) d'une classe d'équivalence $\mathcal{C}_i$ de la famille, $f(A, B) =$ vrai alors que pour tout couple de sous-arbres (C, D) de classes distinctes, $f(A, B) =$ faux.

Afin d'indexer les sous-arbres, nous les classons dans un ensemble de familles de classes d'équivalence. Chaque famille correspond à un profil d'abstraction déterminé.

10.2.2 Graphe de familles de classe d'équivalence

Graphe

Afin de caractériser la similarité de sous-arbres selon des critères plus ou moins précis, nous utilisons plusieurs familles de classes d'équivalence. Il s'agit de créer une taxonomie hiérarchique des sous-arbres à l'aide de familles de classes. Nous organisons les familles de classes d'équivalence sous la forme d'un graphe orienté acyclique avec une relation de spécialisation des profils d'abstraction. Nous distinguons parmi ces familles plusieurs types selon leur arité entrante et sortante :

- les familles sources d'arité entrante nulle ;
- les familles feuilles d'arité sortante nulle ;
- et les familles composites d'arité entrante d'au moins deux.

Familles sources Les familles sources contiennent les classes d'équivalence racines des sous-arbres ; elles représentent ainsi les profils d'abstraction les plus généraux. Elles utilisent généralement une méthode de hachage pour le classement des sous-arbres en classe d'équivalence.

Familles feuilles Les familles feuilles représentent les profils les plus spécialisés. Elles sont les seules à accueillir des tables de sous-arbres membres. Lorsque l'énumération de l'ensemble des membres d'une classe d'équivalence c_1 d'une famille non feuille f_1 est requise, il est nécessaire de trouver une branche menant de f_1 vers une famille feuille f_n. Nous déterminons ensuite itérativement l'ensemble des sous-classes c_2 de c_1 sur f_2, des sous-classes de c_3 de c_2 sur f_3, ..., jusqu'à obtenir l'ensemble des sous-classes c_n sur la famille feuille f_n et déterminer ensuite les membres de c_n représentant les membres de la classe c_1.

Familles composites et index Les familles composites permettent l'obtention d'un profil d'abstraction plus spécialisé avec la prise en considération de plusieurs familles entrantes. La fonction d'égalité F d'une famille composite issue de k parents de fonctions f_1, f_2, ..., f_k est définie par $F = f_1 \wedge f_2 \wedge \cdots \wedge f_k$. Ainsi, deux sous-arbres A et B appartiennent à la même classe de la famille composite si leurs classes d'appartenance sont identiques pour toutes les familles entrantes. Une classe d'équivalence de c est donc définie par le k-uplet de ses classes d'équivalence entrantes.

Il est utile depuis une classe d'équivalence de c de f_i d'obtenir les sous-classes d'équivalence sur la famille composite F : cela nécessite l'indexation des classes de F par un critère de tri sur la classe de f_i.

Un exemple de graphe de familles de classes d'équivalence

Nous présentons ici un exemple de graphe de familles de classes d'équivalence adapté au langage Java. Nous définissons à cet effet les profils d'abstractions suivants correspondant chacun à une famille de classes d'équivalence que nous organisons en graphe de spécialisation :

- Le profil *funcAbstr* d'abstraction des corps de fonction. Seules les signatures des déclarations de méthodes (avec abstraction des identificateurs) ainsi que les autres membres des classes sont conservées. Ce profil permet de répérer des schémas de conception spécifiques. Il s'agit d'une famille source.
- Le profil *subAbstr* d'abstraction des petits sous-arbres de taille inférieure (en nombre de nœuds) à un seuil t fixé. Seuls les petits sous-arbres contenu dans le corps d'une méthode sont abstraits alors que les nœuds d'unités ultra-fonctionnels sont ignorés.
- Le profil *funcSubAbstr* est une famille composite issue de *funcAbstr* et *subAbstr*. Les sous-arbres infra-fonctionnels et fonctionnels similaires à de petits sous-arbres près abstraits sont regroupés dans les mêmes classes.
- Le profil *nodeAbstr* d'abstraction des types de nœuds. Seule la structure de l'arbre de syntaxe, sans des éléments sémantiquement inintéressants, est conservée.
- Le profil *nodeSubAbstr* est une famille composite de classes d'équivalence obtenue depuis *funcSubAbstr* et *nodeAbstr* : cette famille abstrait uniquement les types de nœud (et considère ainsi la structure uniquement) des petits sous-arbres.
- La famille source *contAbstr* permet l'obtention d'empreintes supprimant les nœuds de structures de contrôles et abstrayant types, identificateurs et éléments sémantiquement inintéressants (commentaires, modificateurs).
  On en déduit une famille composite *typeAbstr* prenant pour parents *contAbstr* mais aussi *nodeSubAbstr*. Cette famille abstrait également types, identificateurs et éléments sémantiquement inintéressants mais ne supprime plus les structures de contrôles. Toutefois si celles-ci étaient de taille inférieure au seuil d'abstraction de sous-arbres introduit par *subAbstr*, seule leur structure serait conservée.
- La famille source *typeSet* est basée sur la génération d'empreintes sur l'ensemble des types spécifiés dans un sous-arbre. Analoguement, la famille source *idSet* se base sur des empreintes d'ensemble d'identificateurs de sous-arbre tandis que *commentSet* hache des vecteurs de présence de mots dans les commentaires.
- Nous déduisons, à partir de *typeAbstr*, trois familles composites spécialisées utilisant pour autre parent pour l'une *typeSet* (*typeAbstr+typeSet*), pour l'autre *idSet* (*typeAbstr+idSet*) et enfin pour la troisième *commentSet* (*typeAbstr+commentSet*).
- Enfin une dernière famille *minAbstr* composite d'arité sortante nulle utilise pour parent *typeAbstr+{typeSet,idSet,commentSet}*. Les classes d'équivalence qu'elle contient sont les plus spécialisées et caractérisent des sous-arbres de structures et de types de nœuds identiques avec ensembles partagés de types, identificateurs et mots de commentaires. Nous notons que des nœuds de structures de contrôle racines de petits sous-arbres peuvent être ignorés.

Le graphe de ces familles de classes d'équivalence peut être ainsi exprimé :

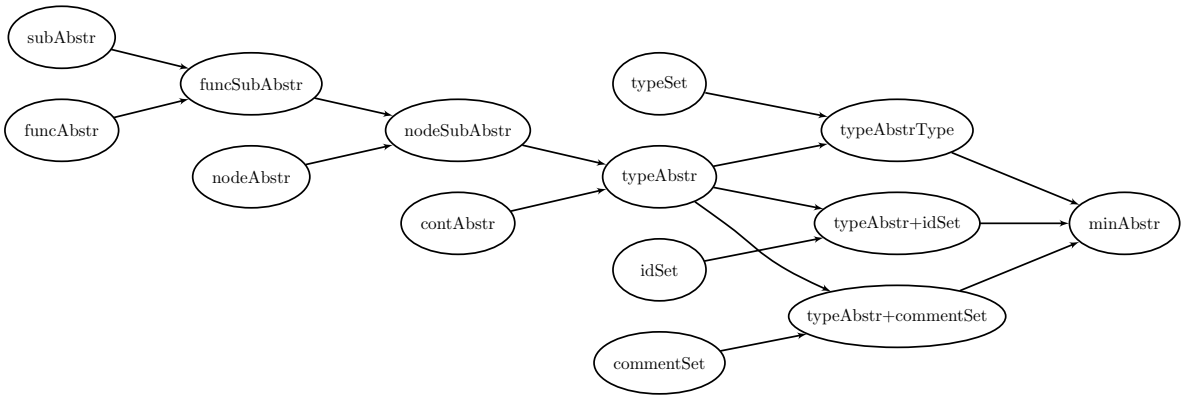


FIG. 10.6 – Un graphe de familles d'abstraction

10.3 Indexation selon un graphe de familles d'équivalence

10.3.1 Structures d'indexation

L'indexation d'arbres de syntaxe selon un graphe de familles nécessite la connaissance des classes des familles sources, la composition de chacune des classes des familles composites en terme de familles entrantes ainsi que les membres des classes des familles sources.

Famille source Comme décrit en 10.1.3, les familles sources, dont la définition des classes d'équivalence est basée sur des valeurs de hachage, maintiennent chacune une table associant valeur de hachage à l'identifiant de la classe.

Famille feuille Une famille feuille représente un profil le plus spécialisé. Nous lions à ces classes d'équivalence la liste de ses sous-arbres membres soit par spécification directe, soit par spécification de la classe d'équivalence du sur-arbre parent avec rang dans la fratrie comme discuté en 10.1.3.

Famille composite Une famille composite possède un index liant les k-uplets des identifiants de ses familles entrantes à un identifiant de classe sur la famille composite. Un index ne considère pour le tri des classes qu'une des permutations pour l'ordre de spécification des identifiants du k-uplet. Ainsi, si l'on souhaite connaître les sous-arbres membres d'une classe d'équivalence d'une famille entrante f_i sur la famille composite F, il est nécessaire que le premier critère de tri de l'index sur les k-uplets porte sur la famille f_i. Dans le cas contraire, la recherche nécessiterait le parcours exhaustif de toutes les classes d'équivalence de la famille composite pour y déceler celles de famille entrante f_i. Il faut donc maintenir au moins k tables de tri afin que chaque famille entrante fasse l'objet d'un premier critère de tri.

10.3.2 Procédure d'indexation

La procédure d'indexation sur un graphe de familles s'avère comparable à celle pour un profil unique décrite en 10.1.5. Les familles feuilles peuvent être ainsi assimilées à des profils uniques et la concaténation de classes d'équivalence des familles entrantes comme une valeur de hachage (sans problématique de collision). Quelques précisions spécifiques sont néanmoins

à apporter sur l'étape préliminaire de détermination de classe d'équivalence pour chaque sous-arbre d'un arbre A à indexer.

Classes d'équivalence sur les familles sources Les classes d'équivalence d'un sous-arbre donné pour toutes les familles sont définies sans ambiguïté par la connaissance des classes d'équivalence de ce sous-arbre selon les familles sources.

Pour chaque famille source, nous déterminons la classe d'appartenance candidate d'un sous-arbre $A[k]$ en calculant la valeur de hachage longue de ce sous-arbre pour l'abstraction considérée et en obtenant la classe dont la valeur de hachage représentative est préfixe de la valeur longue calculée pour le sous-arbre. Soit une classe candidate de valeur préfixe est trouvée : il faut alors valider l'appartenance du sous-arbre $A[k]$ à cette classe ; soit aucune classe n'est trouvée et le sous-arbre $A[k]$ appartient à une nouvelle classe.

Pour vérifier que la classe candidate est adaptée au sous-arbre $A[k]$, comme présenté en 10.1.5 nous analysons chaque sous-arbre de l'arbre à indexer du plus petit au plus grand afin de pouvoir sélectionner sur une classe candidate un sous-arbre représentant déjà indexé pour lequel nous vérifierons qu'il existe des enfants appartenant aux mêmes classes d'équivalence que les enfants de A_i.

Cette procédure nécessite de déterminer un membre représentatif d'une classe d'équivalence d'une famille, ce qui nécessite de suivre une branche menant à une feuille pour obtenir la hiérarchie des sous-familles ainsi que tous leurs membres par une famille feuille. Il est nécessaire de prendre en compte les opérations d'abstraction de la famille considérée : certains sous-arbres peuvent en effet être supprimés par le profil choisi voire une racine de sous-arbre supprimée. La complexité temporelle de recherche des sous-arbres enfants d'un arbre représentant d'une classe d'une famille est multiple du nombre de sous-classes de cette classe sur la famille feuille.

10.3.3 Implantation des structures d'indexation

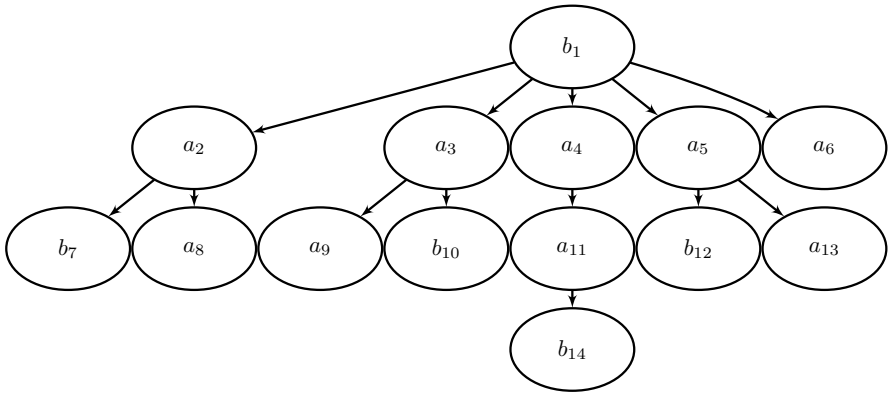
Arbre d'indexation adapté aux supports de masse L'implantation des tables classes d'équivalence pour chaque famille ainsi que les tables de membres pour les profils feuilles doit être réalisé en utilisant des structures d'indexation qui puissent être adaptées à l'usage de mémoire de masse. L'utilisation de structures d'arbres binaires équilibrés classiques peut nécessiter $N \log_2 N$ opérations de lecture ou écriture de bloc disque pour l'accès ou l'écriture d'un élément de l'index sans compter les opérations d'équilibrage. Nous pouvons plutôt opter pour l'usage de k-B+-tree qui sont des arbres d'arité k à $2k$ (avec k de valeur ajustée à la taille d'un bloc de disque) dont les nœuds internes contiennent des clés d'indexation et les feuilles les valeurs indexées. Cette structure permet de réduire le nombre d'accès disque (division par un facteur de $\frac{\ln k}{\ln 2}$) liés au parcours de l'arbre et à son équilibrage. Il est possible de déléguer la tâche d'indexation à un système de gestion de base de données généraliste (tel que PostgreSQL) avec un surcoût lié au traitement des requêtes SQL et aux opérations de communication.

Répartition sur plusieurs supports Les index peuvent aisément être répartis sur K supports distincts par l'usage d'un critère de répartition simple des valeurs de hachage et des identificateurs. Ainsi, les classes d'équivalence dont i est préfixe de la valeur de hachage peuvent être indexés sur le support i alors que les membres d'une classe d'équivalence d'identificateur i peuvent être stockés sur le support $i \mod K$. La répartition par famille est sans doute moins

avantageuse car offrant moins de garantie d'équilibrage du volume des données stockés sur chaque support : chaque famille comporte un nombre de classes d'équivalence hétérogène.

10.3.4 Exemple d'indexation

Nous présentons un exemple d'indexation d'un arbre de syntaxe, avec, dans un souci de simplification, l'utilisation de trois familles de classes d'équivalence dont deux familles sources et une famille composite feuille. La première famille ϕ_1 réalise une abstraction de tous les types de feuilles par une feuille représentante unique : seule la structure des sous-arbres est considérée. Une seconde famille ϕ_2 représente les sous-arbres par la suite brute[3] de ses nœuds sérialisés par un parcours en largeur. Enfin, ϕ_3 est une famille composite feuille ayant pour parents ϕ_1 et ϕ_2 et dont chacune des classes contient les occurrences d'un sous-arbre non-abstrait (structure et types sont pris en compte). Une branche d'intérêt $\phi_1 \longrightarrow \phi_3$ est définie par la présence d'un index sur ϕ_3 triant les couples de classes de ϕ_1 et ϕ_2 selon ϕ_1. Seuls deux types de nœuds a et b sont utilisés. Nous indexons l'arbre $A = b(a(b,a), a(a,b), a(a(b)), a(b,a), a)$. Nous présentons ci-dessous en figure 10.7 les classes d'équivalences de la famille feuille ϕ_3 avec leur composition en classes de ϕ_1 et ϕ_2. Cette table induit la présence de deux index (que nous n'explicitons pas), le premier trié selon la classe de la famille entrante ϕ_1 et le second selon ϕ_2.



Classes de ϕ_3	Composition en classes de ϕ_1 et ϕ_2	Sous-arbre	Membres explicites	Membres par classe du parent
1	$(\phi_1 : 1, \phi_2 : 1)$	b	$A[10], A[14]$	$(4, 1)$
2	$(\phi_1 : 1, \phi_2 : 2)$	a	$A[6], A[9]$	$(4, 2)$
3	$(\phi_1 : 2, \phi_2 : 3)$	$a(b)$	$A[11]$	
4	$(\phi_1 : 3, \phi_2 : 4)$	$a(b, a)$	$A[2], 4[5]$	
5	$(\phi_1 : 4, \phi_2 : 5)$	$a(a(b))$	$A[4]$	
6	$(\phi_1 : 5, \phi_2 : 5)$	$a(a, b)$	$A[3]$	
7	$(\phi_1 : 6, \phi_2 : 6)$	A	$A[1]$	

FIG. 10.7 – Classes d'équivalence d'une famille composite feuille pour l'indexation de l'arbre A

[3]Cette suite brute de nœuds sérialisés ne comporte pas d'information sur la structure de l'arbre : un nœud n'est pas relié à son parent.

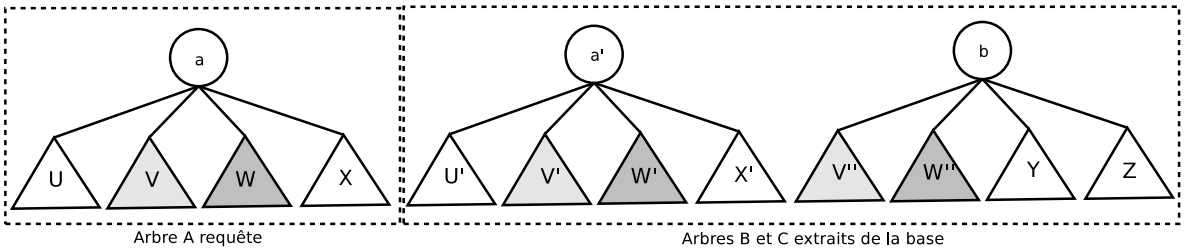


FIG. 10.8 – Un arbre requête A, son dupliqué B sur la base et un arbre de la base C comportant des similarités.

10.4 Recherche de similarité sur arbre requête

10.4.1 Problématique

De part l'organisation des index, la recherche de classes d'équivalence d'un sous-arbre requête sur l'ensemble des arbres indexés est presque immédiate pour une famille donnée. Il suffit de parcourir la branche des familles vers la famille feuille afin de récupérer l'arbre des classes d'équivalence ainsi que les sous-arbres membres directs et les sous-arbres membres par similarité de leur parent.

Nous présentons ici une méthode de recherche de chaînes de sous-arbres frères similaires appartenant à un arbre requête A présents dans la base indexée d'arbres $\mathcal{B}$. Une première étape de récupération de sous-arbres unitaires similaires est réalisée. Pour tout sous-arbre $A[k]$ de A, nous recherchons l'ensemble des sous-arbres de $\mathcal{B}$ similaires selon le profil considéré.

Nous imposons une condition de non-recouvrement des correspondances dans l'arbre. En d'autres termes si le sous-arbre $A[k]$ de A est égal au sous-arbre $B[k']$ de $B \in \mathcal{B}$, cette correspondance est reportée ssi $\mathcal{P}(A[k])$ et $\mathcal{P}(B[k'])$, les sur-arbres parents respectifs de $A[k]$ et $B[k']$ ne sont pas égaux. Pour l'exemple présenté en figure 10.8, nous reportons une correspondance entre les sous-arbre A et B mais pas entre leurs sous-arbres enfants égaux.

Notons que si nous trouvons dans $\mathcal{B}$ une classe d'équivalence de cardinalité non nulle C pour un sous-arbre $A[k]$ (ayant des enfants $A[k]_1, \cdots, A[k]_n$), cela ne signifie pas que la recherche de sous-arbres similaires doit être interrompue pour les descendants de $A[k]$. Il peut être possible de trouver, sur $\mathcal{B}$ des sous-arbres égaux à un descendant de $A[k]$ qui ne soient pas eux-mêmes des descendants des arbres membres de C. Par exemple sur les arbres de la figure 10.8, les classes d'équivalences contenant V, V', V'', et W, W', W'' doivent être prises en considération.

Dans un second temps, nous déterminons les facteurs répétés sur les séquences de sous-arbres. À cet effet, nous utilisons une structure d'indexation de suffixes (cf chapitre 6) pour obtenir un graphe des farmax des chaînes de nœuds frères similaires. Pour l'exemple traité, nous trouverions une correspondance entre les arbres A et B ainsi qu'une correspondance sur les chaînes de sous-arbres V, W, V', W' et V'', W''.

10.4.2 Recherche de sous-arbres individuels égaux

Pour rechercher des sous-arbres de A similaires à des sous-arbres indexés en base, nous procédons comme si nous souhaitions indexer A en base. Nous déterminons pour chacun des sous-arbres de A leur classe d'équivalence d'appartenance pour chacun des profils en s'assurant de l'absence de collision accidentelle entre la valeur de hachage de la classe préfixe binaire de la valeur de hachage longue du sous-arbre de A. Pour les besoins de la recherche, nous nous intéressons uniquement aux sous-arbres égaux selon une certaine famille f assez générale : nous notons les classes d'équivalence de cette famille auxquelles appartient chaque sous-arbre $A[k]$ de A.

Chaque sous-arbre $A[k]$ est ainsi lié à une classe d'équivalence de f avec une hiérarchie de sous-classes afférentes selon les familles plus spécialisées. À partir de cette hiérarchie, nous pouvons obtenir les sous-classes de famille feuille avec leurs membres. Pour chacun des membres de la classe sur la base, nous vérifions si son parent appartient à la même classe que le parent de $A[k]$: si c'est le cas, nous l'ignorons car il est déjà englobé par la similarité de son parent.

10.4.3 Recherche par hachage de chaînes de sous-arbres égaux

Disposant des correspondances entre sous-arbres unitaires de A et classes de correspondance sur la base $\mathcal{B}$, nous souhaitons déterminer les facteurs répétés maximaux de chaînes de sous-arbres tels que définis au chapitre 6. La recherche de facteurs répétés maximaux est intéressante pour les nœuds d'arité importante pour lesquels l'ordre des sous-arbres enfants possède une signification sémantique. Ainsi par exemple, en considérant le langage Java, il est inutile de rechercher des facteurs répétés sur les enfants de nœuds tels que les classes alors que la recherche s'avère incontournable pour les blocs d'instruction afin de localiser des instructions consécutives similaires. On notera toutefois que la recherche de facteurs répétés est inefficace contre des opérations d'obfuscation par insertion ou suppression de code inutile lorsque ces opérations segmentent les chaînes de sous-arbres frères similaires en sous-chaînes de trop faible longueur. Nous introduisons dans le chapitre suivant une méthode d'extension plus adaptée à ces situations permettant de réunir des similarités spatialement proches dans leur arbre de syntaxe.

Transformation des arbres n-aires en arbres binaires

Une première approche pourrait consister à transformer les arbres d'arité forte en arbres binaires avec des pointeurs vers le fils gauche et le frère droit (ou symétriquement vers le fils droit et le frère gauche). Ceci augmente le nombre de sous-arbres et donc la taille de la base pour l'indexation mais permet la recherche de préfixes (ou suffixes) communs sur la séquences des enfants de nœuds des arbres originaux d'arité forte. Toutefois la recherche de facteurs quelconques demeure impossible.

Hachage exhaustif de sous-chaînes

Une deuxième approche, introduite par Baxter et al. [63] pour leur outil de recherche de similitudes sur des arbres de syntaxe, utilise le hachage exhaustif des sous-chaînes de sous-arbres d'une même fratrie. Nous déterminons tout d'abord les fratries candidates susceptibles de participer à des correspondances. Une fratrie est candidate (aussi bien sur A que sur les

arbres de la base $\mathcal{B}$) ssi au moins deux arbres de la fratrie participent à des correspondances. Une base d'indexation est alors créée pour accueillir le résultat du hachage selon les différents profils d'abstraction de toutes les sous-chaînes de sous-arbres des fratries candidates. Nous pouvons généraliser les méthodes de hachage sur les arbres présentées au chapitre 9 pour les chaînes d'arbre en considérant une chaîne d'arbre $A_1 A_2 \cdots A_k$ tel un arbre de racine virtuelle comportant pour enfants les sous-arbres $A_1, A_2, \cdots, A_k$.

Les valeurs de hachage de chaque sous-chaîne d'arbres de la chaîne $A_1 A_2 \cdots A_k$ d'une fratrie étant précalculées, $\frac{k(k+1)}{2}$ empreintes doivent être déterminées correspondant chacune à une sous-chaîne. En calculant les empreintes par longueur croissante des sous-chaînes, il est possible d'exploiter la propriété d'incrémentalité de la fonction de hachage utilisée en calculant l'empreinte d'une sous-chaîne de longueur $i+1$ en utilisant l'empreinte de la sous-chaîne préfixe de longueur i. Chaque empreinte peut ainsi être obtenue en temps constant.

En parcourant les classes d'équivalence de la base temporaire des sous-chaînes de fratrie et en ne considérant que les classes contenant au moins une sous-chaîne de fratrie de A, nous obtenons l'ensemble des sous-chaînes de fratrie similaires. Toutefois, aucune information sur les recouvrements de ces sous-chaînes n'est exploitable. De plus, s'il existe de nombreuses fratries candidates d'arité importante, la complexité temporelle peut devenir rédhibitoire. Si nous considérons des fratries d'instructions au sein de blocs, sauf à effacer les instructions les plus fréquentes des arbres de syntaxe, le ratio de fratries d'instructions candidates sur le nombre total de fratries sur $\mathcal{B}$ peut approcher 1.

10.4.4 Détermination des farmax sur chaînes de sous-arbres frères

Principe général

Une structure d'indexation de suffixes telle qu'un arbre ou une table de suffixes est employée afin d'obtenir les facteurs répétés maximaux complets de sous-chaîne de fratrie.

Détermination de classes d'équivalence de nœuds La première étape consiste à obtenir les classes d'équivalence selon la famille considérée contenant les sous-arbres de l'arbre requête A ainsi que tous ceux de $\mathcal{B}$ correspondant à un sous-arbre de A. Chacune des classes prise en compte contient donc au moins une occurrence de sous-arbre de A. De plus, nous éliminons les classes dont tous les membres partagent un sous-arbre parent similaire.

Inventaire des chaînes de sous-arbres frères consécutifs Les chaînes de sous-arbres frères consécutifs sur A et $\mathcal{B}$ appartenant aux classes sélectionnées sont reportées. Ceci est réalisé en triant l'ensemble des sous-arbres α participant aux classes par couple (rang($\mathcal{P}(\alpha)$), rang(α)) : une chaîne de l sous-arbres frères consécutifs est modélisée par une suite de couples $(i,j),(i,j+1),\cdots,(i,j+l-1)$, sans possibilité d'extension sur la gauche ou la droite. Ces frères ont des identificateurs consécutifs par parcours en largeur et possèdent le même arbre parent d'identificateur i.

Génération de la table de suffixes Une table de suffixes pour toutes les chaînes de sous-arbres frères consécutifs est ensuite calculée. Deux sous-arbres sont considérés comme égaux

ssi ils appartiennent à la même classe d'équivalence selon la famille d'abstraction f considérée. On s'aide à cet effet d'un index liant chaque sous-arbre à sa classe d'équivalence.

Obtention des farmax La table de suffixes obtenue, nous pouvons calculer le graphe des farmax par l'intermédiaire de l'arbre des intervalles selon la méthode décrite en 6.4.4. Ce graphe peut être filtré en éliminant les facteurs ne comprenant aucune occurrence extraite de l'arbre requête. Ainsi, si l'arbre $a(e, c, d, b, f)$ est recherché sur la base constituée des arbres $a(b, c, d, e)$ et $a(f, b, c, d)$, les chaînes de sous-arbre frères consécutifs communs *cd*, *bcd* et *bcd* sont prises en comptes pour la calcul du graphe de farmax $cd \longrightarrow bcd$ (les nœuds correspondant à des chaînes de longueur inférieure à 2 sont ignorés). On notera que le farmax *bcd* ne comprenant aucune occurrence sur l'arbre requête peut être supprimé : seul le farmax *cd* subsiste.

Farmax avec relation de parenté Nous pouvons ajouter au graphe de farmax de chaînes de fratrie une information de lien de parenté lorsqu'un facteur de fratrie est l'enfant d'un autre facteur. Concrètement, pour une occurrence de facteurs de fratrie $a_1a_2 \cdots a_n$ au sein d'un facteur répété α, nous cherchons s'il existe une occurrence de facteur de fratrie $b_1b_2 \cdots b_n$ au sein d'un facteur répété β avec b_i parent de $a_1, a_2, \cdots, a_n$. Dans l'affirmative un lien parent de α vers β est créé. Si la cardinalité de α est égale à β, le facteur répété α peut être supprimé car toutes ses occurrences sont par relation de parenté comprises dans β.

À titre d'illustration, considérons l'arbre requête $A = a(a(a, b), b(b, a), a(b, a))$ et l'arbre de la base $B = b(a(a, b), b(b, a))$. Nous relevons les classes d'équivalences suivantes pour a, b, $a(a, b)$ et $b(b, a)$. Le farmax des chaînes de fratrie comporte les facteurs ab et ba ainsi que les facteurs de longueur 1 $a(a, b)$ et $b(b, a)$ comprenant chacun deux occurrences. ab peut être lié au parent $a(a, b)$: ces deux facteurs ayant le même cardinal d'occurrences, ab peut être supprimé. ba peut être lié au parent $b(b, a)$ comprenant moins d'occurrences : ba ne peut être supprimé.

Support d'opérations d'abstraction

Problématique La recherche de chaînes de sous-arbres frères ne pose pas de difficulté spécifique avec l'usage d'une famille d'abtraction f se limitant à des opérations d'abstraction sur les types des nœuds, la structure de l'arbre n'étant pas impactée. Nous discutons maintenant des opérations non-neutres sur la structure telle que la suppression d'un sous-arbre (décrite en 9.2.4) et la suppression d'une racine de sous-arbre (décrite en 9.2.4). La non-considération de ces opérations a pour conséquence de fractionner des correspondances qui n'auraient pas lieu de l'être. Par exemple les deux sous-arbres $A = a(b, c(b, a), a)$ et $B = d(b, a)$ avec un profil d'abstraction supprimant les sous-arbres de racine c voient deux correspondances reportées sur leur premier et dernier fils de profondeur 2 alors qu'une correspondance complète entre $b, c(b, a), a$ et b, a pourrait être reportée.

Extension de la consécutivité des nœuds par jointures Pour éviter le fractionnement de correspondances en présence d'opérations d'abstraction sensibles à la structure, nous mémorisons parallèlement à l'indexation, selon le profil, des jointures indiquant lorsque deux nœuds deviennent consécutifs suite à une opération d'abstraction. Pour l'exemple précédent, sur A une jointure est créée entre le sous-arbre b de rang 1 et le sous-arbre a de rang 3. Ainsi dans le cas présent, pour une opération de suppression de sous-arbre, la jointure lie des sous-arbres de même parent mais non consécutifs. Lorsqu'une racine de sous-arbre est supprimée,

ses enfants remontent d'un niveau : deux jointures sont créées pour relier le frère gauche de la racine supprimée avec le premier enfant de la racine et le dernier enfant de la racine avec le frère droit de la racine. La condition de consécutivité de deux nœuds est modifiée pour prendre en compte les jointures : deux nœuds sont consécutifs ssi ils partagent le même parent et sont de rang consécutif ou sont liés par une jointure.

Commutativité des sous-arbres d'une fratrie Certaines abstractions peuvent introduire une normalisation de l'ordre de sous-arbres enfants susceptibles d'être commutatifs. Dans cette situation, nous pouvons également normaliser l'ordre des sous-arbres frères consécutifs obtenus après inventaire. La recherche de chaînes de sous-arbres frères consécutifs répétés alors que ceux-ci sont commutatifs ne présente cependant pas un intérêt important dans la mesure où ces éléments (typiquement des membres de classes ou unité de compilation) sont généralement mutuellement indépendants et leur suite peuvent faire l'objet, outre d'opérations de transposition, d'opérations d'insertion et de suppression.

10.4.5 Quantification de l'exactitude de paires de chaînes d'arbres correspondantes

Chaque facteur répété de sous-arbres frères consécutifs selon une famille d'abstraction f obtenu par la méthode précédemment décrite contient un ensemble d'occurrences de chaînes de sous-arbres consécutifs égales selon f. Si f correspond au profil d'abstraction nul, l'égalité sur l'arbre de syntaxe original utilisé est exacte. Si un certain niveau d'abstraction est utilisé pour f, les chaînes de sous-arbres du facteur répété peuvent présenter des différences à des niveaux d'abstraction plus faibles. Nous proposons d'introduire une métrique d'exactitude $\mathcal{E}(u, v)$ afin de quantifier la similarité entre deux chaînes u et v de sous-arbres d'un même facteur répété.

Distance d'édition sur les arbres

Une distance d'édition peut être calculée entre les occurrences de chaînes de sous-arbres frères des facteurs répétés afin de quantifier leur exactitude. Nous utilisons à cet effet trois types d'opérations d'édition élémentaires symétriques, chacune associée à un coût spécifique :

1. La suppression $\delta(A)$ de la racine a (ou son ajout) de l'arbre $A = a(C_1, \cdots, C_l)$ de coût $C_\delta(a)$.
2. La suppression $\Delta(A)$ du sous-arbre complet A (ou son ajout) de coût $C_\Delta(A)$. Typiquement, le coût $C_\Delta(A)$ de suppression de A est égal à la somme des coûts $C_\delta(a[k])$ de suppression de chaque des nœuds $a[k]$ de A.
3. Le changement d'un type de nœud de a en a' $\kappa(a, a')$ (ou symétriquement de a' en a) de coût $C_\kappa(a, a') = C_\kappa(a', a)$. Nous choisissons C_κ tel que $C_\kappa(a, a') < C_\delta(a) + C_\delta(a')$: il est plus avantageux de changer le type de nœud que de le supprimer pour en ajouter un du nouveau type.

Pour deux occurrences $A = A_1 A_2 \cdots A_m$ et $B = B_1 B_2 \cdots B_m$ d'un facteur répété de sous-arbres frères, avec $A_1 = B_1, A_2 = B_2, \cdots, A_m = B_m$ selon la famille f utilisée, nous cherchons à calculer la distance d'édition minimale $D(A, B)$. Tout d'abord, nous notons que nous pouvons trouver entre deux sous-arbres enfants T_i et T_{i+1} de A ou B des opérations de suppression

intersticielles de sous-arbre complet (si T_i et T_{i+1} sont liés par une jointure et ont le même parent). Des opérations de suppression de racine de sous-arbre sont caractérisés par la présence d'une sous-chaînes de nœuds $T_i \cdots T_j$ ayant le même parent, différent du parent des nœuds environnants T_{i-1} et T_{j+1}.

Dans un premier temps, nous calculons le coût associé aux opérations de suppression intersticielles de sous-arbres ou racines entre A et B. Les fratries ou sous-arbres intersticiels entre A_i, A_{i+1} et B_i, B_{i+1} sont comparées. Si seul l'interstice sur A ou B comportent des éléments ignorés par la famille, le calcul de la distance d'édition est immédiat et correspond à la création (ou destruction) de ces éléments. Si les deux interstices sur A et B sont occupés par des éléments, ceux-ci sont comparés après désérialisation par une méthode d'alignement de forêt d'arbres (cf 5.5).

Dans un second temps, nous calculons récursivement les distances des sous-arbres enfants explicites $D(A_i, B_i)$.

Pour le calcul récursif de la distance d'édition sur les sous-arbres, nous comparons les types de racine afin d'éventuellement sommer un coût de changement de type. Pour les sous-arbres enfants, deux situations peuvent être rencontrées :

- Les sous-arbres enfants sont indexés selon la famille : nous pouvons donc déduire les opérations implicites de suppression de sous-arbre et de racine en utilisant les informations de jointure et la connaissance des rangs des sous-arbres enfants par désérialisation paresseuse. Nous pouvons ensuite procéder comme indiqué précédemment pour le calcul de la distance d'édition sur une chaîne de sous-arbres.
- Les sous-arbres enfants ne sont pas indexés selon la famille et aucune information de jointure n'est disponible. Ceci peut survenir, par exemple, pour des sous-arbres de petite taille. Il est alors nécessaire de désérialiser les sous-arbres afin de les comparer selon une méthode d'alignement d'arbre décrite antérieurement en 5.5.

Exactitude normalisée La distance d'édition $D(\alpha, \beta)$ entre les arbres α et β est bornée par $C_\delta(\alpha) + C_\delta(\beta)$ correspondant au coût de destruction de α et de construction complète de β. Nous obtenons ainsi une métrique d'exactitude normalisée : $\mathcal{E}(\alpha, \beta) = 1 - \frac{D(\alpha,\beta)}{C_\delta(\alpha)+C_\delta(\beta)}$.

Exemple

Nous considérons comme exemple deux fonctions de calcul de nombre de Fibonacci déjà évoquées en figure 4.4 afin d'illustrer différentes méthodes d'obfuscation : la fonction originale et celle avec réécriture d'expressions. Nous utilisons un profil d'abstraction adapté à cette obfuscation : les petits sous-arbres de trois nœuds ou moins qui ne sont pas des instructions sont abstraits (remplacés par le nœud ζ), les commentaires sont ignorés et les structures de boucle sont supprimées. Nous en déduisons donc, en figure 10.9, les deux chaînes f_1 (fonction originale) et f_2 (fonction avec réécriture d'expression) de sous-arbres instructions frères similaires selon la famille d'abstraction, ayant pour parent le bloc d'instructions de la fonction.

f_1	f_2	Abstraction
commentaire		ignoré
int k = l ;	int k = l ;	decl
int l = 1 ;	int l = 1 ;	decl
int m = 0 ;	int m = 0 ;	decl
commentaire		ignoré
if (n == 1) return k	if (n == 1) return k*1 ;	if(ζ, return ζ)
if (n == 2) return l	if (n == 2) return l+0	if(ζ, return ζ)
for	*for*	racine supprimée
int i = 3 ;	int i = 3 ;	decl
i < n ;	i < n ;	cond
i++ ;	i += 1 ;	aff
m = k + l	m = k + 2*l - l ;	f_1 : aff(ζ, ζ), f_2 : aff$(\zeta, \zeta + \zeta - \zeta)$
k = l	k = l/1 + 0 ;	f_1 : aff(ζ, ζ), f_2 : aff$(\zeta, \zeta + \zeta)$
l = m	l = m * m / (m*1) ;	f_1 : aff(ζ, ζ), f_2 : aff$(\zeta, \zeta/\zeta)$
return m	return pgcd(m, m) ;	return ζ

FIG. 10.9 – Abstractions de deux versions de fonctions de calcul de nombre de Fibonacci

Nous constatons que l'abstraction des sous-arbres permet de confondre les instructions conditionnelles initiales de f_1 et f_2, cependant la réécriture trop étendue des expressions opérées pour les affectations de la boucle ne permet pas de les classer dans une même classe d'équivalence. Nous obtenons ainsi pour plus grand facteur répété sur f_1 et f_2 (récupérable par la table des LCP) la sous-chaîne suivante :

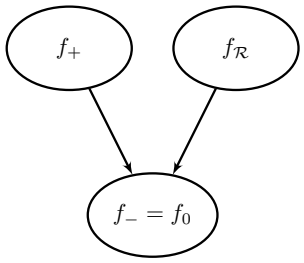
decl	decl	decl	if(ζ, return ζ)	if(ζ, return ζ)	decl	cond	aff

Nous calculons ensuite la distance d'édition $D(A, B)$ entre les deux occurrences A de f_1 et B de f_2 du plus long facteur répété. Celle-ci est composée des coûts de suppression des deux commentaires, ainsi que de la suppression de la racine de l'arbre de boucle *for*. Nous devons également comparer les sous-arbres abstraits par ζ : ainsi pour les instructions conditionnelles de retour k est transformé en $k * 1$ et l en $l + 0$, deux transformations nécessitant l'ajout d'une racine (opérateur binaire) et d'une opérande. Si toutes les opérations élémentaires d'édition ont un coût unitaire, $D(A, B) = 9$. En considérant $|A| = 32$ et $|B| = 35$, l'exactitude est estimée à $\mathcal{E}(A, B) = \frac{58}{67} \sim 0,87$.

10.5 Évaluation de quelques familles d'abstraction

Nous étudions expérimentalement des profils d'abstraction typiques à l'aide de graphes de famille de classes d'équivalence à trois nœuds. Une première famille de classe d'équivalence f_+ utilise le profil d'abstraction étudié tandis qu'une famille $f_\mathcal{R}$ est utilisée pour raffiner cette classe et produire depuis f_+ la famille $f_\mathcal{A} \cup f_\mathcal{R} = f_-$ de faible niveau d'abstraction.

Les cas étudiés de graphes de familles de classes d'équivalence sont introduits dans le tableau de la figure 10.10. Nous supposerons pour la suite que le niveau de plus faible abstraction des



Désignation	f_+
primTypeAbstr	f_0 avec abstraction des types primitifs
blockAbstr	f_0 avec abstraction des corps de méthodes et commutativité des membres
subAbstr4	f_0 avec abstraction des arbres de moins de 5 nœuds
nodeAbstr	f_0 avec abstraction de tous les types de nœuds
structAbstr	f_0 avec abstraction de la structure

FIG. 10.10 – Cas étudiés de graphes de familles de classes d'équivalence

arbres de syntaxe, noté f_0 prend en considération tous les types de nœuds et leurs propriétés sauf certains nœuds et sous-arbres sémantiquement non significatifs (commentaires, instructions d'importation et modificateurs) et réalise une abstraction des identificateurs (types non-primitifs et noms de variables) et constantes littérales ainsi qu'une normalisation de l'ordre des sous-arbres enfants de classes et des opérandes d'opérateurs infixes. Les tests sont réalisés sur des sous-arbres de volume d'au-moins 30 nœuds afin d'éviter de relever des similarités élémentaires.

Pour chacun des cas étudiés, nous supposons que le niveau d'abstraction f_- définit l'oracle de pertinence des clones : aucun couple de clones des classes d'équivalence de f_- n'est faux-positif.

La figure 10.11 présente des propriétés des classes d'équivalence des familles f_+ selon le nombre de sous-classes d'équivalence dont elles sont une généralisation. Deux projets développés en langage Java sont analysés : le paquetage de visualisation et d'édition de Javadoc pour l'éditeur Netbeans (environ 19K lignes de code dans 101 fichiers) ainsi que l'ensemble des paquetages spécifiques à la plate-forme de recherche de similarité Plade (environ 82K lignes de code dans 928 fichiers), présentés en annexe B.

Famille *primTypeAbstr* L'existence de classes à sous-classes multiples pour la famille abstrayant les types primitifs met en relief l'absence d'un mécanisme de généricité de code paramétré par des types primitifs en Java. La duplication de code pour différents types primitifs est ainsi notable dans l'API JDK de Java (méthodes de tri, de gestion de buffers dupliquées). Ce phénomène est cependant peu présent dans les projets étudiés (12 classes à 2 ou 3 sous-classes pour Plade, une pour Netbeans-Javadoc). Les clones différents par leur types primitifs sont de faible volume et concernent principalement pour Plade des constructeurs avec instructions répétées d'affectation ainsi que d'autres morceaux idiomatiques tel que celui présenté en figure 10.12 pour Netbeans-Javadoc présent en 18 exemplaires en 3 variantes de types (*boolean*, *long* et *int*).

Famille *blockAbstr* La famille *blockAbstr* permet de regrouper les méthodes de même signature ainsi que les classes équivalentes par les signatures de méthodes qu'elles contiennent. En particulier, les classes partageant un même ancêtre hiérarchique sans ajout de méthode

Profil ↓ / k →	1	2	3	4	5	[6..10]	[11..20]	> 20	Max
primTypeAbstr	209	0	1						3
	324	10	2						3
blockAbstr	8	24	9	1	4	9	3	2	55
	53	109	49	26	12	21	8	16	83
subAbstr4	194	22	2	3	0	0	0	0	4
	299	114	13	5	4	8	5	1	24
nodeAbstr	209	2	1						3
	322	22	2						3
structAbstr	211	1	2						3
	325	17	0						2

FIG. 10.11 – Nombre de classes de f_+ comportant k sous-classes de f_- pour chaque graphe de familles sur les projets Netbeans-Javadoc (1re ligne) et Plade (2e ligne)

```
109     public void setUse (boolean b) {
110         boolean old = use;
            use = b;
            firePropertyChange("use", new Boolean(old), new Boolean(use)); }
```

(a) StdDocletSettingsService

```
71      public void setMembers( long l ) {
            long old = members;
            members = l;
            firePropertyChange("members", new Long(old), new Long(members)); }
```

(b) ExternalJavadocSettingsService

FIG. 10.12 – Deux membres d'une même classe de f_+ = *primTypeAbstr* mais de sous-classes différentes de f_0 extraits de Netbeans-Javadoc

Cas	Classes concernées
Clones pertinents de code vivant factorisable	7
Clones pertinents avec un exemplaire de code mort	2
Clones idiomatiques difficilement factorisables	6
Clones non-pertinents	5

FIG. 10.13 – Évaluation de 20 classes de clones utilisant *subAbstr4*

additionnelle partagent la même valeur de hachage. Des classes de même valeur sans lien de parenté pourrait potentiellement bénéficier de la spécification d'un ancêtre commun. Concernant les méthodes de même signature, les plus nombreuses sont de prototype `void ()` pour Netbeans-Javadoc et Plade. Les types non-primitifs étant confondus, seul le nombre d'arguments est considéré : ce type d'abstraction n'a donc pas d'utilité réelle pour la recherche de schéma de conception. La prise en compte du type non-primitif déclaré ainsi que de la hiérarchie des types afin d'établir une relation de compatibilité entre signatures s'avère alors incontournable.

Famille *subAbstr4* La famille *subAbstr4* permet de regrouper des sous-arbres présentant des différences de nœuds à forte profondeur. Les sous-arbres comportant 4 nœuds ou moins sont abstraits ce qui induit une résistance à la réécriture simple d'expressions. Un effectif important de classes à deux sous-classes est présent pour Plade et dans une moindre mesure pour Netbeans-Javadoc. Pour Plade, la classe aux sous-classes les plus nombreuses concerne des interfaces comportant une ou plusieurs méthodes avec des paramètres abstraits car représentés par des sous-arbres de moins de 4 nœuds. Les classes à deux sous-classes comprenant un effectif réduit de clones s'avèrent les plus intéressantes dans une optique de factorisation de code comme le couple de clones de Plade présenté en figure 10.14. Nous pouvons être confronté également à des familles regroupant des clones apparemment faux-positifs de squelette structurel identique, avec quelquefois une similarité conceptuelle (pour 10.15 la conversion d'un tableau d'objets en un autre) sans possibilité de factorisation. Concernant Plade, nous avons évalué les clones de 20 classes parmi les 114 classes comportant 2 sous-classes avec un jugement subjectif humain sur la pertinence de ces familles. Le résultat de cette évaluation est résumé par le tableau en figure 10.13 (chaque classe contenant deux sous-classes d'un exemplaire de code). L'abstraction de petits sous-arbres montre son intérêt pour trouver des clones pertinents (9/20). La fréquence des clones idiomatiques trouvés pourrait être réduite par un filtrage supprimant les clones les plus fréquemment rencontrés sur une base importante de projets.

Famille *nodeAbstr* Concernant l'abstraction totale des types de nœuds (famille *nodeAbstr*), nous constatons que le nombre de classes à effectif élevé de sous-classes est particulièrement faible. Par exemple pour Netbeans-Javadoc, seules trois classes comportent plus d'une sous-classe. Les clones concernés diffèrent uniquement par des types utilisés (*primTypeAbstr* aurait pu suffire pour les regrouper) ou par l'emploi de littéraux à la place de types. Nous pouvons en conclure, pour les exemples considérés, que ne considérer que la structure améliore le rappel et se révèle peu générateur de faux-positifs. En effet, l'arité d'un nœud ainsi que l'arité des nœuds de son sous-arbre est assez prédictif de son type. Toutefois, la seule considération de la structure s'avère insuffisante pour une distinction fine des sous-arbres de faible volume.

```
66  if (mode.equals(OpeningMode.CREATE))
    {
     File dir = new File(path);
     boolean created = dir.mkdir();
70   if (! created) throw new IOException("Cannot␣create␣the␣directory␣'" + dir + "'");
    }
```

(a) DefaultTypeRepository

```
54  if (mode.equals(OpeningMode.CREATE))
55  {
     File f = new File(path);
     boolean created = f.mkdir();
     if (! created)
      throw new IOException("Cannot␣create␣directory␣" + path);
60  }
```

(b) LiveParsingRepresentationRepository

FIG. 10.14 – Deux exemplaires de code de Plade dans la même classe d'abstraction *subAbstr4* différant par une expression

```
108   String[] components = str.split(DEFAULT_ARRAY_SEPARATOR);
      Object[] data = new Object[components.length];
110   for (int k=0; k < components.length; k++)
       data[k] = fromString(components[k], cl.getComponentType(), dependencies);
```

(a) DefaultStringConverter

```
43  public Match[] getSelectedMatches()
    {
45   int[] selection = graphViewer.getSelection();
     Match[] matches = new Match[selection.length];
     for (int k=0; k < matches.length; k++)
      matches[k] = graphViewer.getGraph().getDataNode(selection[k]);
     return matches;
50  }
```

(b) MatchGraphViewer

FIG. 10.15 – Morceaux de code réunis par *subAbstr4* mais difficilement factorisables

Profil d'abstraction	Volume de couverture global
primTypeAbstr	39 790 85 928
blockAbstr	72 051 236 518
subAbstr4	47 330 158 530
nodeAbstr	39 928 99 355
structAbstr	40 440 86 142

FIG. 10.16 – Volume de couverture global pour les différents profils d'abstraction

Famille *structAbstr* La famille *structAbstr* réalise une abstraction de la structure des sous-arbres hachés en prenant en considération uniquement le types des nœuds : cela revient à considérer le sac (ensemble avec multiplicité) des nœuds d'un sous-arbre. Pour des sous-arbres de volume non négligeable, cette approche apparaît d'une bonne précision. Sur le projet Netbeans-Javadoc, ce profil d'abstraction, tout comme le précédent *nodeAbstr* se révèle pratiquement équivalent à f_0. Pour Plade, plus de classes multi-composées sont relevées. Après analyse exhaustive, l'ensemble de ces classes multi-composées apparaît comme pertinente même si dans certains cas l'intérêt d'une factorisation est discutable.

Rappel Concernant le rappel, la figure 10.16 peut servir de base comparative des différents profils d'abstraction. Le volume de couverture global des projets étudiés y est exposé ; ce volume est défini par la somme des volumes d'exemplaires de clone participant à une classe d'équivalence comportant au moins deux exemplaires de code. En faisant exception du cas particulier du profil *blockAbstr*, même si la précision exacte des exemplaires supplémentaires obtenus est inconnue, *subAbstr4* semble proposer le meilleur rappel. *nodeAbstr* et *structAbstr* présentent des couvertures disparates pour les deux projets et soulignent des caractéristiques différentes des clones de chacun des projets ; Plade présente plus de clones structurels creux que Netbeans-Javadoc. Ce dernier projet manipule intensément la bibliothèque d'affichage graphique Swing ce qui est susceptible de générer de nombreux clones d'un degré d'idiomaticité plus ou moins fort. On prendra pour exemple (figure 10.17) deux implantations de constructeur de composant graphique différant par des transpositions d'instructions rassemblées dans une même classe par *structAbstr* (un troisième exemplaire non présenté concerne le constructeur de `StandardTagPanel`). *structAbstr* permet également de regrouper des exemplaires dont des instructions auraient été remontées ou descendues dans l'arbre de syntaxe (changement de parent).

Quelques statistiques sur les classes À titre informatif afin de mieux appréhender la nature des clones des deux projets réunis par le profil d'abstraction minimal f_0, la figure 10.18 présente quelques statistiques de ses classes d'équivalence. Elle fait apparaître que les clones de Plade sont de volume moyen plus important, sans doute lié à la coexistence de code actif recopié depuis du code devenu mort et en instance de suppression. D'autre part les clones de Netbeans-Javadoc sont regroupés dans des classes d'effectif plus important confirmant l'hypothèse de clonage idiomatique lié à l'utilisation de Swing. Le ratio de couverture global par des clones

```
31      public ParamTagPanel( final JavaDocEditorPanel editorPanel ) {
            super( editorPanel );
            initComponents();
            initAccessibility();
35          jLabel2.setDisplayedMnemonic(org.openide.util.NbBundle.getBundle(ParamTagPanel.class).getString("
                CTL_ParamTagPanel.jLabel2.text_Mnemonic").charAt(0)); // NOI18N
            jLabel1.setDisplayedMnemonic(org.openide.util.NbBundle.getBundle(ParamTagPanel.class).getString("
                CTL_ParamTagPanel.jLabel1.text_Mnemonic").charAt(0)); // NOI18N
            addHTMLComponent( descriptionTextArea );
            editorPanel.registerComponent( descriptionTextArea );
            parameterComboBox.getEditor().getEditorComponent().addFocusListener(
40          new java.awt.event.FocusAdapter() {
                public void focusLost(java.awt.event.FocusEvent evt) {
                    commitTagChange(); }
            }); }
```

(a) ParamTagPanel

```
67      public ThrowsTagPanel( JavaDocEditorPanel editorPanel ) {
            super( editorPanel );
            initComponents ();
70          jLabel2.setDisplayedMnemonic(org.openide.util.NbBundle.getBundle(StandardTagPanel.class).getString("
                CTL_ThrowsTagPanel.jLabel2.text_Mnemonic").charAt(0)); // NOI18N
            jLabel1.setDisplayedMnemonic(org.openide.util.NbBundle.getBundle(StandardTagPanel.class).getString("
                CTL_ThrowsTagPanel.jLabel1.text_Mnemonic").charAt(0)); // NOI18N
            editorPanel.registerComponent( descriptionTextArea );
            addHTMLComponent( descriptionTextArea );
            exceptionComboBox.getEditor().getEditorComponent().addFocusListener(
75          new java.awt.event.FocusAdapter () {
                public void focusLost (java.awt.event.FocusEvent evt) {
                    commitTagChange(); }
            });
            initAccessibility(); }
```

(b) ThrowsTagPanel

FIG. 10.17 – Deux constructeurs de composants graphiques avec transposition d'instructions de Netbeans-Javadoc réunis par *structAbstr* (volume des clones : 91 nœuds)

Propriété	Min	Q1	Médiane	Moyenne	Q3	Max
Volume moyen des classes ($\mathcal{V}$)	29,50 17,86	36,00 35,00	49,00 51,00	73,14 120,0	85,38 93,0	525,0 3 434
Cardinalité des classes (c)	2,000 2,000	2,000 2,000	2,000 2,000	3,024 2,151	3,000 2,000	48,00 7,000
Couverture des classes ($\mathcal{C} = \mathcal{V}c$)	59,00 48,00	90,00 74,00	125,0 114,0	187,5 246,5	214,0 192,0	1 584,0 6 868,0
Paires dans chaque classe ($\frac{c(c-1)}{2}$)	1,000 1,000	1,000 1,000	1,000 1,000	13,58 1,39	3,000 1,0	1 128 21,00
Nombre de classes	212 344					
Volume de couverture global ($\mathcal{C}_{\text{moyenne}} * C$)	39 757 84 790					

FIG. 10.18 – Valeurs aux quartiles (avec moyenne) pour les propriétés des classes de f_0 (volume, cardinalité des classes et proximité des exemplaires de clone) pour les projets Netbeans-Javadoc (1re ligne) et Plade (2e ligne)

est deux fois plus faible pour Plade que pour Netbeans-Javadoc. Il faut toutefois nuancer les résultats obtenus en rappelant que seuls les sous-arbres unitaires de l'arbre de syntaxe de plus de 30 nœuds sont considérés : le rappel pourraît être favorisé par l'utilisation d'un seuil plus faible pour collection des correspondances servant de germes à une consolidation puis en ignorant ensuite les germes de volume trop faibles non consolidés. Dans cette optique, nous tentons d'étudier dans l'annexe suivante quelques expérimentations de consolidation de germes.

10.6 Limitations de l'indexation par profil

La méthode d'indexation par famille de profils permet de constituer des bases d'empreintes permettant de déterminer des k-correspondances approchées entre arbres de la base et arbre requête. Cette méthode ne permet cependant que de gérer certains types d'abstraction courants bien délimités induisant un niveau de normalisation de l'arbre indexé. S'il est possible d'ignorer certains mots-clés méta-informatifs, de supprimer des commentaires, des racines de structures ou abstraire types ou commentaires, voire d'ignorer l'ordre de sous-arbres, il est impossible de gérer des opérations d'édition arbitraires consistant à ajouter ou supprimer des sous-arbres quelconques. Nous présentons au chapitre suivant une heuristique d'extension se basant sur un ensemble de k-correspondances trouvées sur une base d'indexation. Les 2-correspondances (paires de clones) en sont extraites et nous cherchons à les consolider suivant un critère de proximité dans l'arbre de syntaxe. Nous obtenons ainsi des paires d'arbres présentant une duplication approchée avec insertion, suppression ou transposition d'arbres sur une fratrie ou entre cousins de niveau hétérogène.