Détection de définitions OCaml similaires (ou comment ne plus voir double à dos de chameau)

Alexandre Moine et Yann Régis-Gianas

Université de Paris (UMR-CNRS 7126) - INRIA PI.R2

Résumé

L'absence de redondance est souvent un gage de qualité pour un code source. En effet, lorsqu'un fragment de code est répété, ses imperfections – pour ne pas dire ses erreurs – le sont elles-aussi.

Seulement, il arrive parfois que l'on constate de la redondance dans un grand corpus de code, typiquement quand ce corpus a été construit par des développeurs ne communiquant peu ou pas du tout entre eux. Deux instances de cette situation nous intéressent particulièrement : l'ensemble des codes sources des paquets OPAM et l'ensemble des copies d'étudiants répondant tous aux mêmes questions de programmation. Comment partitionner leurs définitions en fonction de leur "similarité" ?

Dans cet article, nous proposons un outil de partitionnement automatique d'un ensemble de définitions écrites en OCaml. Cet outil s'appuie sur une fonction dédiée de prise d'empreintes des arbres de syntaxe du langage intermédiaire Lambda ainsi que sur un algorithme de classification hiérarchique classique que nous avons adapté à notre usage.

Cet outil prend la forme d'une bibliothèque nommée Asak disponible sur OPAM. Nous l'avons utilisée d'une part pour partitionner automatiquement les réponses d'étudiants qui apprennent OCaml en utilisant la plateforme LearnOCaml, et d'autre part, pour détecter des redondances sur l'ensemble des codes sources des paquets OPAM disponibles aujourd'hui. Nous évaluons les résultats obtenus et formulons quelques limites de notre approche.

1 Introduction

"Ne vous répétez pas !" est une injonction permanente qui plane au dessus de tout programmeur cherchant à écrire un logiciel de qualité [10]. Ce principe vise une situation idéale où une connaissance donnée n'a qu'une seule représentation dans le logiciel : si cette connaissance est imparfaite – et c'est souvent le cas – on s'assure alors qu'elle pourra être corrigée efficacement.

Il arrive parfois qu'un corpus de code source contienne des redondances que les programmeurs n'ont pas pu ou pas voulu éviter. Par exemple, certaines fonctions utilitaires absentes de la bibliothèque standard sont ainsi réimplémentées à de multiples reprises par de nombreux paquets logiciels. L'implémentation de ces fonctions utilitaires répétées sont parfois identiques aux caractères près ; parfois, elles sont seulement très proches (à quelques renommages près ou plus généralement, à quelques réécritures "bénignes" près). Ainsi, si l'on utilise deux bibliothèques indépendantes pour construire un logiciel donné, il y a une probabilité non nulle que l'on retrouve dans l'exécutable final plusieurs implémentations quasi-identiques de la même fonction. Ne serait-il pas plus raisonnable d'introduire une bibliothèque commune offrant une unique implémentation de ces fonctions répétées ?

Il y a aussi des redondances dont les programmeurs n'ont pas conscience. En effet, on peut parfaitement appliquer "la tête dans le guidon" des schémas de calcul généraux en les spécialisant à des arguments particuliers, au cas par cas, sans réaliser les opportunités de factorisation qui découleraient de l'introduction d'une fonction d'ordre supérieur bien choisie. C'est seulement

lorsque les développeurs prennent un peu de recul qu'ils s'aperçoivent qu'une factorisation du code source est possible. Ne serait-il pas plus efficace d'alerter le programmeur à l'instant même où il introduit un fragment de code qui est fortement similaire à un autre fragment préexistant, soit dans son propre code, soit dans une autre bibliothèque ?

Lorsqu'un corpus est formé de réponses d'étudiants à une même question de programmation, il y a fort à parier que plusieurs étudiants répondront de façon similaire à une question donnée, soit parce qu'ils ont trouvé une réponse "naturelle", soit parce qu'ils ont commis des erreurs de raisonnement ou de conception classiques. Pour l'enseignant face à plusieurs centaines de réponses, il n'est pas toujours évident de réaliser quelles sont les classes qui partitionnent pertinemment l'ensemble de ces réponses. Lui apporter un tel partitionnement serait un outil précieux car il lui permettrait de différencier sa pédagogie en fonction des groupes de réponses fausses les plus courantes.

Pour répondre à ces trois cas d'usage, il faudrait tout d'abord que l'on sache évaluer la "similarité" entre deux fragments de code. Mais qu'entend-on exactement par similarité ? Peut-on formaliser cette notion et la réaliser calculatoirement ?

Notons tout d'abord que nous avons utilisé le terme de similarité et non d'équivalence. En effet, la notion d'équivalence (syntaxique, définitionnelle ou observationnelle) est trop "binaire" pour notre cadre : nous cherchons une mesure qui rapproche des calculs qui se ressemblent même s'ils ne se comportent pas tout à fait de la même façon calculatoirement. En d'autres termes, nous cherchons des fragments de programme qui ont des structures syntaxiques proches et qui s'appuient sur des ingrédients similaires plutôt que des fragments de programme ne pouvant pas être distingués par des contextes d'évaluation.

Ainsi, deux termes égaux syntaxiquement sont absolument similaires. Deux termes qui diffèrent par un renommage sont très similaires sans être syntaxiquement égaux. Deux analyses de motifs sont plus ou moins similaires en fonction des cas d'analyse qu'elles traitent de façon similaire. Par contre, les implémentations de deux algorithmes de tri distincts sont en général dissimilaires même si l'équivalence observationnelle ne permet pas de les distinguer [1].

La similarité semble donc être une notion très syntaxique mais qui cherche paradoxalement à faire peu de cas de certains détails d'écriture qui ne changent pas fondamentalement le programme. De toute évidence, toutes les différences purement textuelles (commentaires, indentations, ...) doivent être ignorées par une bonne notion de similarité. Le renommage des variables liées est aussi un exemple canonique de tels détails syntaxiques anodins. Plus généralement, toute transformation locale et purement syntaxique, i.e. toute élimination de sucre syntaxique, semble aussi rentrer dans cette catégorie des "détails syntaxiques". Où s'arrêter dans ce processus de simplification des termes sources ? Devrait-on par exemple aller jusqu'à comparer les codes machines obtenus par compilation des termes sources ? Cette démarche conduirait sans doute à un échec puisque le jeu de la sélection d'instructions et des différentes optimisations peut mener deux termes sources proches à des codes machines très différents et fourmillant de nouveaux détails peu importants (pensez à la diversité des instructions d'une architecture comme x86-64). Il faut donc trouver le langage intermédiaire offrant un bon niveau d'abstraction pour éliminer à la fois les détails syntaxiques du langage source et les détails de bas-niveau de l'architecture cible.

La première contribution de cet article est de considérer que les premières passes du compilateur OCaml, celles menant de sa syntaxe concrète au code Lambda, éliminent la plupart des détails syntaxiques des termes OCaml pour ne garder que leurs ingrédients calculatoires

1. Bien entendu, on suppose ici un langage de programmation incapable d'observer finement les exécutions des deux algorithmes.

principaux. Nous analyserons les avantages (sections 5 et 6) et les limitations (section 7) de ce choix de conception.

Comme son nom l'indique, LAMBDA est un λ-calcul (étendu). Nous nous sommes donc moralement ramené au problème de la construction d'une mesure de similarité syntaxique entre deux termes du λ-calcul. En utilisant une représentation de De Bruijn et en effectuant quelques réductions inoffensives, on gomme effectivement de cette façon certaines différences sans intérêt entre deux termes OCAML. Seulement, pour pouvoir comparer rapidement des milliers de $\lambda-$termes deux-à-deux, on doit se donner une représentation compacte des λ-termes. Nous introduisons donc un prétraitement des λ-termes pour calculer leurs *empreintes* respectives : l'empreinte d'un λ-terme est un ensemble d'entiers qui caractérisent ses aspects syntaxiques importants. C'est une idée déjà présente dans la littérature [5] mais nous la raffinons pour l'appliquer à des λ-termes. La définition de ce prétraitement sur des λ-termes est donc la seconde contribution présentée par cet article.

À ce stade, on peut donc mesurer la similarité entre chaque paire de termes d'un corpus. L'ensemble de ces mesures constitue une grande quantité d'information peu structurée qui est donc difficilement exploitable directement, en tout cas, pour nos cas d'usage. Pour pallier à ce problème, nous proposons de partitionner hiérarchiquement les termes en s'appuyant sur leur mesure de dissimilarité (section 4). Encore une fois, notre contribution consiste à raffiner et implémenter un algorithme déjà présent dans la littérature, la classification ascendante hiérarchique. Cette classification produit des dendrogrammes, particulièrement adaptés à l'exploration progressive des classes de programmes issues de nos corpus. Un dendogramme est un arbre binaire dont les nœuds représentent des classes d'individus.

Pour que ce travail soit réutilisable dans des situations différentes de nos propres cas d'usage, nous avons développé une bibliothèque nommée ASAK [2]. Cette bibliothèque est librement disponible sous la forme d'un paquet OPAM et sur GITHUB [1]. Elle nous a permis d'introduire un système de classification automatique des copies dans LEARNOCAML [3] (section 5). Elle nous a aussi permis d'implémenter un outil de recherche de redondance dans l'ensemble des paquets OPAM (section 6). Ces outils constituent la dernière contribution présentée par cet article.

Cet article décrit la première version d'ASAK : il s'agit d'un travail de recherche en cours dont les résultats préliminaires nous ont semblé suffisamment intéressants pour être communiqués. Il reste encore beaucoup à faire pour rendre l'outil plus performant et plus pertinent. Nous évaluons ses limitations (section 7), le comparons à l'état de l'art (section 8) et donnons les pistes à explorer pour tenter de les surmonter (section 9).

2 Vue d'ensemble de l'approche

Que produit ASAK ? ASAK compare entre elles les définitions globales d'un ensemble de modules OCAML. La figure 2 contient cinq exemples de telles définitions OCAML. Elles forment un corpus jouet qui va nous servir à illustrer le fonctionnement d'ASAK. Le lecteur aura reconnu d'un coup d'œil la fonction classique qui renvoie la liste miroir d'une liste prise en argument et l'enseignant aura reconnu des réponses typiques d'étudiants apprenti-programmeurs fonctionnels. Sur ce corpus, notre outil produit l'ensemble de dendrogrammes de la figure 2.

Un dendrogramme est un arbre binaire dont les nœuds représentent des classes d'individus. Les feuilles de ces dendrogrammes sont les différentes versions de `rev`. Un dendrogramme fournit un partitionnement hiérarchique d'un ensemble d'individus : en le parcourant d'une feuille vers

2. Cette bibliothèque fait des partitions (de codes similaires), son nom est donc tiré de la musique : ASAK est un genre de chansons touareg.

```
(* Code 1 *)
let rec rev l = match l with
    [] -> []
  |t::q -> rev q@[t]
(* Code 2 *)
let rec rev l =
  match l with
  | [] -> []
  | a::t -> (rev t)@[a]
(* Code 3 *)
let rec rev l = match l with
    [] | [_] -> l
  | t::q -> rev q@[t];;
```

```
(* Code 4 *)
let rev l=
  let rec rev_aux acc l=
    match l with
    |[]->acc
    |t::q->rev_aux (t::acc) q
  in rev_aux [] l
(* Code 5 *)
let rev l =
  match l with
    [] -> []
  |a::q -> let rec rev2 x y = match y with
         [] -> x
       |b::z -> rev2 (b::x) z in rev2 [] l
(* Code 6 *)
let rev l =
  List.fold_left (fun acc x -> x :: acc) [] l
```

FIGURE 1 – Un corpus jouet pour illustrer notre algorithme.

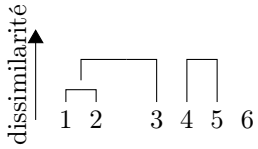


FIGURE 2 – Ensemble de dendrogrammes pour la classification du corpus jouet.

sa racine, on découvre des classes d'invididus de plus en plus dissimilaires à cette feuille ; en le parcourant de sa racine vers ses feuilles, on découvre des séparations successives en deux classes d'un ensemble d'individus maximisant la dissimilarité entre les membres des deux classes. Dans la sortie de notre outil, deux classes qui appartiennent à deux dendrogrammes distincts sont absolument dissimilaires, i.e. elles n'ont rien en commun.

Dans le cas de ce corpus, le regroupement des définitions 1 et 2 est naturel puisque ces deux définitions sont équivalentes à quelques détails purement textuels près : la présence du `|`, d'un retour à la ligne et d'un renommage. La définition 3 est proche de cette première classe : elle en diffère seulement à cause d'un cas d'analyse supplémentaire. Le regroupement des définitions 4 et 5 est assez naturel lui aussi puisqu'elles suivent la même (et bonne !) stratégie qui consiste à accumuler le résultat dans l'argument d'une fonction auxiliaire interne. Notons que ces deux définitions sont dissimilaires même si elles utilisent les mêmes "ingrédients" : la définition 5 effectue une analyse par cas supplémentaire pour traiter la liste vide de façon spécifique.

Comment procède ASAK ? Ce partitionnement semble pertinent mais comment notre outil l'a-t-il obtenu ? Comme nous l'avons déjà écrit dans l'introduction, le traitement de notre outil se décompose essentiellement en trois étapes : (i) les programmes sources sont normalisés pour ignorer les détails syntaxiques que nous jugeons inessentiels ; (ii) on calcule une empreinte pour caractériser la structure et les ingrédients principaux des programmes normalisés ; (iii) on applique un algorithme de partitionnement hiérarchique qui s'appuie sur les empreintes.

```
Définition 1: (function l/88 (if l/88
    (apply (field 36 (global Stdlib!)) (apply rev/87 (field 1 l/88))
      (makeblock 0 (field 0 l/88) 0a)) 0a))
Définition 2: (function l/88 (if l/88
    (apply (field 36 (global Stdlib!)) (apply rev/87 (field 1 l/88))
      (makeblock 0 (field 0 l/88) 0a)) 0a))
Définition 3: (function l/88 (catch
    (if l/88
      (if (field 1 l/88)
        (apply (field 36 (global Stdlib!)) (apply rev/87 (field 1 l/88))
          (makeblock 0 (field 0 l/88) 0a))
        (exit 12))
      (exit 12)) with (12) l/88))
Définition 4: (function l/88 (letrec (rev_aux/89
       (function acc/90 l/91
         (if l/91
           (apply rev_aux/89 (makeblock 0 (field 0 l/91) acc/90)
             (field 1 l/91))
           acc/90))) (apply rev_aux/89 0a l/88)))
Définition 5: (function l/88 (if l/88
    (letrec (rev2/91
         (function x/92 y/93
           (if y/93
             (apply rev2/91 (makeblock 0 (field 0 y/93) x/92) (field 1 y/93))
             x/92)))
      (apply rev2/91 0a l/88)) 0a))
Définition 6: (function l/88 (apply (field 20 (global Stdlib__list!))
    (function acc/146 x/147 (makeblock 0 x/147 acc/146)) 0a l/88))
```

FIGURE 3 – Traduction du corpus dans le code LAMBDA du compilateur OCAML.

Comment les définitions sont-elles normalisées ? L'analyse syntaxique est la solution canonique pour éliminer les artefacts textuels et ne garder que la structure d'un code source. Notre outil travaille donc sur des arbres de syntaxe abstraits.

Une fois que l'on a décidé de travailler sur un langage d'arbres distincts de la syntaxe concrète, il faut choisir ce nouveau langage. On aurait pu choisir de traduire les termes sources vers un langage conçu pour l'occasion mais ce serait beaucoup de travail. Il existe heureusement un langage intermédiaire adéquat dans le compilateur OCAML : le langage LAMBDA. Nos expérimentations nous portent à croire qu'il se place au bon niveau d'abstraction pour capturer l'essence de la structure calculatoire du programme source.

Ce langage sera présenté précisément dans la section 3.1 mais nous pouvons d'ores et déjà donner la traduction du corpus jouet dans la figure 3. Pour réaliser cette traduction, nous réutilisons la partie avant du compilateur OCAML et nous effectuons un post-traitement qui normalise encore un peu plus les termes. Les détails de cette traduction seront décrits dans la section 3.1. Sur nos exemples, on peut déjà remarquer que les définitions 1 et 2 sont identiques une fois normalisées. On note aussi que la définition 3 normalisée partage des sous-termes avec les définitions 1 et 2 normalisées. Des remarques similaires s'appliquent aux autres définitions.

Pourquoi calcule-t-on des empreintes ? Comparer deux arbres en itérant sur leurs structures respectives a un coût proportionnel à leur taille. Par ailleurs, la classification qui nous

intéresse doit idéalement savoir comparer l'ensemble des sous-arbres des deux arbres pour déterminer quelle quantité de code ils ont en commun. L'ordre d'apparition des sous-arbres n'est donc pas forcément important : bien entendu, deux arbres utilisant les mêmes sous-arbres dans le même ordre seront très similaires mais utiliser les mêmes sous-arbres dans un ordre différent est aussi une forme de similarité non négligeable même si elle est un peu moins forte.

Après ces remarques, l'implémentation d'une fonction d'évaluation de la similarité entre deux termes Lambda semble difficile. Nous introduisons les empreintes d'arbres pour simplifier cette implémentation et aussi pour la rendre efficace. Les empreintes sont des ensembles de clés de hachages des sous-termes (suffisamment gros) du terme Lambda. L'idée importante de cette notion d'empreinte est de prendre en compte l'ordre relatif des sous-termes dans le calcul de la clé de hachage tout en regardant aussi l'empreinte comme un ensemble de clés pour maintenir une certaine proximité entre les termes qui utilisent les mêmes sous-termes, mais dans un ordre différent. Ainsi, les deux programmes suivants :

```
1 let f () = e1; e2
2 let g () = e2; e1
```

ont pour empreintes :

$$\begin{array}{rcl} E(\texttt{f}) & = & \{H(\texttt{e1; e2}), H(\texttt{e1}), H(\texttt{e2})\} \\ E(\texttt{g}) & = & \{H(\texttt{e2; e1}), H(\texttt{e1}), H(\texttt{e2})\} \end{array}$$

où $E(t)$ est l'empreinte de t et $H(t)$ est la clé de hachage de t.

Ces deux empreintes sont distinctes mais partagent deux clés de hachage. Ce partage témoigne du fait qu'elles "utilisent les mêmes ingrédients". Les empreintes calculées pour les définitions de notre corpus jouet se trouvent dans la figure 4. On distingue la liste numérotée des clés de hachage dans la partie haute de la figure. En bas de la figure, on trouve les empreintes des définitions, ce sont des (multi-)ensembles de paires d'entiers. Dans cette figure, l'entier du haut est le numéro de clé de hachage et l'entier du bas est le nombre de nœuds de l'arbre de syntaxe du sous-terme haché. Nous donnons la définition précise de cette prise d'empreintes dans la section 3.

Comment le partitionnement hiérarchique est-il effectué ? Il existe deux grandes familles de partitionnement hiérarchique : les partitionnements ascendants et les partitionnements descendants. Les partitionnements ascendants sont adaptés aux corpus formés de petits groupes tandis que les partitionnements descendants à ceux formés de grands groupes. Nous avons fait l'hypothèse que les groupes de définitions similaires sont petits par rapport au corpus analysé et nous avons donc choisi un algorithme de partitionnement hiérarchique ascendant.

L'algorithme procède donc en partant d'un partitionnement où chaque individu est dans une classe distincte de celles des autres et choisit à chaque itération de fusionner les deux classes les moins dissimilaires. Nous définirons précisément la mesure de dissimilarité que nous utilisons dans la section 4 mais pour se donner une idée du fonctionnement de l'algorithme, le lecteur pourra en observer la trace d'exécution sur notre corpus jouet dans la figure 5.

3 Prise d'empreintes

3.1 Lambda normalisé

Présentation de Lambda La syntaxe du langage Lambda est donnée dans la figure 6. Il s'agit d'un λ-calcul avec quelques spécificités par rapport à celui que l'on trouve dans les présentations à saveur plus théoriques. Tout d'abord, les fonctions ne sont pas unaires : elles ont une arité potentiellement supérieure à 1. Ensuite, on distingue les primitives des constantes :

```
01 = 6cdbdd45d9fda8de7da9dc4d9add5d43d4ad80d44df9d9cd
02 = 58de8dc8d46d83d87d4bd23d24d93d82d69de2d32d64dc6d
03 = dddc1d3dd8adf7d7d41d4dd8bd9d2ed30d6edf4d9fd2d
04 = 64dc6d47d3adf8d15dc5d28d5d2bd12d6d7dda2d74dc6d
05 = 89db7df1d30df9d8adfddbfd6bdf8dc5decd66d13dbed5ad
06 = acdadd7ddebd3dd8fd21da4d71d4fdd2db4dbcde0d10d68d
07 = 72dfd3cdfad34d33de1d4fd46d4dd9cdcbd9cd77d12d32d
08 = dfddd6edefd9ddbddffde0dc3d11d58d8cdcdb5d73dafd
09 = 78da6d29de0d96dabd9fd2d71d9fdcdd9db7d3ad74dfcd
10 = 5ed72d97d35db8d71d1cdb6d79de7dbcd3ad19dfdd8dd56d
11 = 8bda1d14d86d29df2da7d7dda0d4dd5addad5fd2d1fdddd
12 = d4d1dd8cdd9d8fd0db2d4de9d80d9d98decdf8d42d7ed
13 = cbdccdbfdfcddbd7dd7ed77d39dc4d27d8dda1d25d4ed86d
14 = dad42decd1cd57df0d66df4ddadc1d72d29d42db3dfad7ed
15 = afd20d78d4fdbdd60d83d7fda3d30d5adcbdc3d3dd82de0d
16 = 72d64d45d84d18d55dabde6d2bd29dfad1fd55dc7de4d15d
17 = e6d95d5bdaadc8dc2d9fd33d7dd4dd1edacd10deed67d76d
18 = 36d80d66d3fd81defd16d12d16ddedb7dbbd94d2ed7dd11d
19 = bad43dccdcdd18d52d3bd82d2ada8d0d8ed90da8dc2dd0d
20 = cfd38df3df7d42dc0ddbdd5d53de6d9bd39d6adfcd6ad20d
21 = b2dfddb8d99dccd8adc5d6edacde2d26d66dcdd7cdaed39d
22 = 47db7dbed82dd0d84d17d51d8fdd9db1de1d45d39d6ad38d
23 = ad14d0d7cd4ed7de9dd3d4cde6d20de6d57df1dcade0d
24 = c5d34dccd9fdc4d9d96d6ddcfd44d82d5cd19db5dd6dd6d
25 = 1fd53d57dceda3df6daed5fdb8de6d7fdc4d61dafde8d96d
26 = 18dedb6dcddc4d69d33d9bd5bdcad46df0d56d8fd1dc5d
27 = f1d4fdad1ed86db0d7dd2fd41df1df4d1fd25d36d46d54d
28 = 42d45d6dd2fd86d91d1dd78dd6dfdd2ed4fd4ad1d20d24d
29 = 6bdcad8dd97d19dc8d86d87d95d1ddf1d49dedd17d91d5dd
30 = ebd8cd7ed71de9d71d65dc2d33d65d57d11de5df3d7ad73d
31 = e0dbbdefd1ddfed38d27d22d0df4dd6dddacde3dcbd77d
32 = 3adebdc5dc4d4ad5addadf6d9ad8fde4d38decd50d90da5d
33 = 39d7d29d5dcdd80ddddedd5de3d5dda3d2ad57d4ed96d
34 = a8d7bdccda2d93d67d2ddaed34d38df1de2dbad95d3ad41d
35 = 6adf1d16daedc3d80db8dc8d3da2dfbde9dd7d84d72d83d
36 = f1dead89d1cd13dd6db9d8fde6d73d19d44d2bd11d90dcbd
37 = 81d72d4dd8cdbcdd5db6d43d86d2fd75dddd2dd85d67dbcd
38 = 5edf5d2ed7cd37d4cdded6bd3edd8d94dd6d4cdbad79d95d
39 = 20d57df0dbfd84df5d8bdecd92d6dd5ed22d4fd6d58debd
```

$$E(D_1) = \{\begin{bmatrix}13\\20\end{bmatrix}\begin{bmatrix}01\\19\end{bmatrix}\begin{bmatrix}02\\01\end{bmatrix}\begin{bmatrix}03\\16\end{bmatrix}\begin{bmatrix}04\\05\end{bmatrix}\begin{bmatrix}05\\03\end{bmatrix}\begin{bmatrix}06\\02\end{bmatrix}\begin{bmatrix}02\\01\end{bmatrix}\begin{bmatrix}07\\06\end{bmatrix}\begin{bmatrix}08\\05\end{bmatrix}\begin{bmatrix}09\\01\end{bmatrix}\begin{bmatrix}05\\03\end{bmatrix}\begin{bmatrix}06\\02\end{bmatrix}\begin{bmatrix}02\\01\end{bmatrix}\begin{bmatrix}10\\03\end{bmatrix}\begin{bmatrix}11\\02\end{bmatrix}\begin{bmatrix}12\\01\end{bmatrix}\begin{bmatrix}09\\01\end{bmatrix}\}$$

$$E(D_2) = \{\begin{bmatrix}13\\20\end{bmatrix}\begin{bmatrix}01\\19\end{bmatrix}\begin{bmatrix}02\\01\end{bmatrix}\begin{bmatrix}03\\16\end{bmatrix}\begin{bmatrix}04\\05\end{bmatrix}\begin{bmatrix}05\\03\end{bmatrix}\begin{bmatrix}06\\02\end{bmatrix}\begin{bmatrix}02\\01\end{bmatrix}\begin{bmatrix}07\\06\end{bmatrix}\begin{bmatrix}08\\05\end{bmatrix}\begin{bmatrix}09\\01\end{bmatrix}\begin{bmatrix}05\\03\end{bmatrix}\begin{bmatrix}06\\02\end{bmatrix}\begin{bmatrix}02\\01\end{bmatrix}\begin{bmatrix}10\\03\end{bmatrix}\begin{bmatrix}11\\02\end{bmatrix}\begin{bmatrix}12\\01\end{bmatrix}\begin{bmatrix}09\\01\end{bmatrix}\}$$

$$E(D_3) = \{\begin{bmatrix}18\\29\end{bmatrix}\begin{bmatrix}14\\28\end{bmatrix}\begin{bmatrix}15\\26\end{bmatrix}\begin{bmatrix}02\\01\end{bmatrix}\begin{bmatrix}16\\22\end{bmatrix}\begin{bmatrix}05\\03\end{bmatrix}\begin{bmatrix}06\\02\end{bmatrix}\begin{bmatrix}02\\01\end{bmatrix}\begin{bmatrix}03\\16\end{bmatrix}\begin{bmatrix}04\\05\end{bmatrix}\begin{bmatrix}05\\03\end{bmatrix}\begin{bmatrix}06\\02\end{bmatrix}\begin{bmatrix}02\\01\end{bmatrix}\begin{bmatrix}07\\06\end{bmatrix}\begin{bmatrix}08\\05\end{bmatrix}\begin{bmatrix}09\\01\end{bmatrix}\begin{bmatrix}05\\03\end{bmatrix}\begin{bmatrix}06\\02\end{bmatrix}\begin{bmatrix}02\\01\end{bmatrix}\begin{bmatrix}10\\03\end{bmatrix}\begin{bmatrix}11\\02\end{bmatrix}\begin{bmatrix}12\\01\end{bmatrix}\begin{bmatrix}17\\02\end{bmatrix}$$
$$\begin{bmatrix}12\\01\end{bmatrix}\begin{bmatrix}17\\02\end{bmatrix}\begin{bmatrix}12\\01\end{bmatrix}\begin{bmatrix}02\\01\end{bmatrix}\}$$

$$E(D_4) = \{\begin{bmatrix}31\\22\end{bmatrix}\begin{bmatrix}19\\21\end{bmatrix}\begin{bmatrix}20\\16\end{bmatrix}\begin{bmatrix}21\\15\end{bmatrix}\begin{bmatrix}22\\14\end{bmatrix}\begin{bmatrix}23\\01\end{bmatrix}\begin{bmatrix}24\\11\end{bmatrix}\begin{bmatrix}25\\06\end{bmatrix}\begin{bmatrix}26\\05\end{bmatrix}\begin{bmatrix}27\\01\end{bmatrix}\begin{bmatrix}28\\03\end{bmatrix}\begin{bmatrix}29\\02\end{bmatrix}\begin{bmatrix}23\\01\end{bmatrix}\begin{bmatrix}28\\03\end{bmatrix}\begin{bmatrix}29\\02\end{bmatrix}\begin{bmatrix}23\\01\end{bmatrix}\begin{bmatrix}27\\01\end{bmatrix}\begin{bmatrix}30\\04\end{bmatrix}\begin{bmatrix}09\\01\end{bmatrix}\begin{bmatrix}02\\01\end{bmatrix}\}$$

$$E(D_5) = \{\begin{bmatrix}33\\25\end{bmatrix}\begin{bmatrix}32\\24\end{bmatrix}\begin{bmatrix}02\\01\end{bmatrix}\begin{bmatrix}19\\21\end{bmatrix}\begin{bmatrix}20\\16\end{bmatrix}\begin{bmatrix}21\\15\end{bmatrix}\begin{bmatrix}22\\14\end{bmatrix}\begin{bmatrix}23\\01\end{bmatrix}\begin{bmatrix}24\\11\end{bmatrix}\begin{bmatrix}25\\06\end{bmatrix}\begin{bmatrix}26\\05\end{bmatrix}\begin{bmatrix}27\\01\end{bmatrix}\begin{bmatrix}28\\03\end{bmatrix}\begin{bmatrix}29\\02\end{bmatrix}\begin{bmatrix}23\\01\end{bmatrix}\begin{bmatrix}28\\03\end{bmatrix}\begin{bmatrix}29\\02\end{bmatrix}\begin{bmatrix}23\\01\end{bmatrix}\begin{bmatrix}27\\01\end{bmatrix}\begin{bmatrix}30\\04\end{bmatrix}\begin{bmatrix}09\\01\end{bmatrix}\begin{bmatrix}02\\01\end{bmatrix}\begin{bmatrix}09\\01\end{bmatrix}\}$$

$$E(D_6) = \{\begin{bmatrix}39\\13\end{bmatrix}\begin{bmatrix}34\\12\end{bmatrix}\begin{bmatrix}35\\05\end{bmatrix}\begin{bmatrix}36\\04\end{bmatrix}\begin{bmatrix}37\\03\end{bmatrix}\begin{bmatrix}23\\01\end{bmatrix}\begin{bmatrix}38\\01\end{bmatrix}\begin{bmatrix}09\\01\end{bmatrix}\begin{bmatrix}02\\01\end{bmatrix}\begin{bmatrix}10\\03\end{bmatrix}\begin{bmatrix}11\\02\end{bmatrix}\begin{bmatrix}12\\01\end{bmatrix}\}$$

FIGURE 4 – Les empreintes des définitions de notre corpus jouet.

Matrice de dissimilarité :

$$\begin{pmatrix} 0 & 0 & 144 & \infty & \infty & \infty \\ 0 & 0 & 144 & \infty & \infty & \infty \\ 144 & 144 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & 71 & \infty \\ \infty & \infty & \infty & 71 & 0 & \infty \\ \infty & \infty & \infty & 71 & \infty & 0 \end{pmatrix}$$

Évolution du partitionnement :

Étape 1	$\{1,2\},\{3\},\{4\},\{5\},\{6\}$	car 1 et 2 ont la même empreinte.
Étape 2	$\{1,2\},\{3\},\{\{4\},\{5\}\},\{6\}$	car 4 et 5 sont les plus proches.
Étape 3	$\{\{1,2\},\{3\}\},\{\{4\},\{5\}\},\{6\}$	car 3 et $\{1,2\}$ sont les plus proches.
Étape 4	$\{\{1,2\},\{3\}\},\{\{4\},\{5\}\},\{6\}$	car toutes les classes sont absolument dissimilaires.

FIGURE 5 – Trace du partitionnement des définitions du corpus jouet.

t	::=	x	*Variable*
	|	c	*Constante*
	|	$t(\overline{t})$	*Application*
	|	$\lambda\overline{x}.t$	*Abstraction*
	|	**let** b **in** t	*Définition locale*
	|	**let rec** $\overline{b}$ **in** t	*Définitions récursives*
	|	$\delta(\overline{t})$	*Appel d'une primitive*
	|	**switch** $t\,\{\,\overline{\gamma};\overline{\gamma};t\,\}$	*Branchement*
	|	**staticraise** $n(\overline{t})$	*Saut local*
	|	**statictry** t **with** $n(\overline{x}) \to t$	*Expression étiquetée*
	|	**try** t **with** $x \to t$	*Expression étiquetée*
	|	**if** t **then** t **else** t	*Branchement conditionnel*
	|	$t;t$	*Séquencement*
	|	**while** t **do** t **done**	*Boucle non bornée*
	|	**for** $x = t$ **to** t **do** t **done**	*Boucle bornée ascendante*
	|	**for** $x = t$ **downto** t **do** t **done**	*Boucle bornée descendante*
	|	$x := t$	*Affectation*
	|	$t \# t(\overline{t})$	*Appel de méthode*
b	::=	$x = t$	*Définition*
γ	::=	$n \to t$	*Branche*

FIGURE 6 – La syntaxe de LAMBDA avec $n \in \mathbb{N}, x \in \mathcal{V}, c \in \mathcal{C}, \delta \in \mathcal{P}$.

les primitives sont nécessairement appliquées. Le langage contient un fragment impératif permettant d'affecter des variables, d'itérer *via* des boucles `for` et `while`, et enfin de détourner le flot du contrôle *via* les différents mécanismes de lancement et rattrapage d'exceptions. L'appel de méthode est la construction qui nous rappelle qu'OCAML est un aussi un langage à objets. Pour finir, LAMBDA n'a pas d'analyse de motifs mais est muni d'un branchement n-aire introduit par le mot-clé `switch`. Par manque de place, nous ne donnons pas les règles de sémantique de ce langage et nous laissons au lecteur le soin de réfléchir à ces dernières. Par contre, pour se convaincre que l'on ne perd pas trop de structure en calculant la redondance sur des termes de LAMBDA et non des termes OCAML, il faut prendre le temps d'expliquer les différences entre ces deux langages.

Absence de types LAMBDA est un langage non typé. On n'y retrouve donc aucune déclaration de types. Cette absence limite donc d'emblée le champ d'application de ASAK : nous ne pouvons pas détecter de déclarations de type redondantes. Cependant, cette limitation a un avantage : lorsque l'on oublie les types, on se donne la possibilité de détecter plus de redondances entre des programmes de types distincts mais partageant la même structure. En revenant aux définitions des types qui interviennent dans deux programmes ayant la même structure, on peut espérer détecter indirectement des redondances entre les définitions de types. Un exemple d'une telle situation sera présentée et discutée dans la section 6.

Absence de modules et de classes Les constructions de seconde classe (comme les définitions de modules et de classes d'objets) ont disparu dans le programme traduit en LAMBDA. Cela limite notre capacité à détecter des modules similaires ou des classes similaires. Comme pour les définitions de type, nous pensons qu'en nous focalisant uniquement sur les aspects calculatoires, nous pouvons détecter *a posteriori* des définitions de modules ou de classes similaires parce qu'elles partagent des définitions similaires.

Absence d'analyse de motifs L'analyse de motifs d'OCaml a été compilée en Lambda sous la forme d'arbres de décision, exprimés à l'aide d'expressions conditionnelles et de branchements. Deux analyses de motifs distinctes syntaxiquement en OCaml peuvent être envoyées vers le même arbre de décision et donc le même code Lambda : c'est par exemple le cas de deux analyses à trois branches sur le type `color = Red | Black | White` car quelque soit l'ordre des branches de l'analyse [3], celles-ci vont être traduites vers des branchements à trois cas où l'ordre des cas correspond à l'ordre des constructeurs de données dans la définition du type. Il s'agit donc encore une fois d'une simplification favorable des termes sources puisqu'elle envoie des termes sources syntaxiquement distincts mais sémantiquement équivalents vers un unique terme Lambda.

α-renommage Le nom des variables est pris en compte lors de la prise d'empreintes. Naturellement, nous avons choisi de nous abstraire de ces noms en détectant la redondance à α-équivalence près. Il est donc important de s'assurer que les noms de variables liées (par exemple lors d'une définition de fonction) ne jouent aucun rôle. Pour cette raison, nous renommons tous les identificateurs des variables liées en des identificateurs canoniques qui codent leurs indices de De Bruijn.

Réduction des définitions locales inoffensives En OCaml, et contrairement à Coq, on ne peut pas facilement comparer les termes modulo l'évaluation : en effet, l'évaluation d'un terme peut diverger ou produire des effets de bord incontrôlés. Par contre, le compilateur sait détecter des définitions locales très simples dont l'expansion ne pose pas de problème. Ces définitions sont marquées par une annotation du type suivant :

```
1 type let_kind = Strict | Alias | StrictOpt | Variable
```

`Strict` signifie que la définition peut faire des effets de bord et ne doit pas être réduite (sauf s'il s'agit d'une variable ou d'une constante) ; `Alias` signifie que la définition est pure et peut donc être réduite ; `StrictOpt` signifie que la définition dépend de la mémoire et ne peut donc pas être réduite ; `Variable` signifie que la définition sera masquée dans la suite. Notre normalisation réduit les définitions `Strict` simples et les `Alias` ce qui nous permet d'égaliser des termes sources qui sont équivalents modulo le dépliage de ces définitions. [4]

3.2 Définition de la prise d'empreintes

Notre méthode de prise d'empreintes est une variante d'un calcul d'empreinte de la littérature [5]. L'empreinte d'un arbre de syntaxe doit témoigner de la structure de cet arbre ainsi que de l'ensemble des sous-arbres qui la constitue. Pour des raisons d'efficacité, nous réduisons chaque sous-arbre à un entier correspondant à son image par une fonction de hachage. Par ailleurs, plus un sous-arbre a un grand nombre de nœuds et plus sa contribution à l'arbre global est important : nous pondérons donc la clé de hachage par ce nombre de nœuds.

Definition 1
On appelle *f-glyphe d'un arbre de syntaxe t*, le couple $H_f(t)$ formé d'un entier 63 bits correspondant à son nombre de nœuds et d'un entier 128-bits correspondant à l'image de cet arbre à travers une fonction de hachage f donnée. Pour un glyphe g donné, on note son poids $w(g)$ et sa clé de hachage $h(g)$.

3. En supposant qu'il n'y a pas de clause `when` en jeu.
4. Notons que cette normalisation peut changer la complexité ce qui d'un certain point de vue ne préserve pas la sémantique intentionnelle du programme source. Dans nos expérimentations, ces considérations ne se sont pas montrées pertinentes cependant.

Definition 2
On appelle *f-empreinte d'un arbre de syntaxe t*, le multi-ensemble $E_f(t)$ formé des f-glyphes de ses sous-arbres et de lui-même.

Il reste maintenant à décider quelle fonction de hachage utiliser. Ne nous intéressant pas à des questions de sécurité, nous avons décidé d'utiliser intensivement la fonction de hachage cryptographique MD5. Pour chaque nœud de l'arbre, on calcule une clé de hachage qui s'appuie sur le nom de la construction utilisée et, pour procéder incrémentalement [8], sur les clés de hachage des sous-arbres prise dans l'ordre, de gauche à droite.

Dans un programme, la valeur d'un littéral est très souvent liée au contexte d'application. Du point de vue de la recherche de redondance, nous estimons que ces valeurs constituent une forme de bruit à ignorer. Pour que la valeur exacte des littéraux n'influence pas notre recherche de redondance, nous avons donc décidé de ne pas prendre en compte leurs valeurs dans le calcul de la clé de hachage.

Pour finir cette section, il faut remarquer qu'il n'est pas pertinent de garder tous les glyphes des sous-termes dans l'empreinte finale. En effet, nous avons vu que la fonction de partitionnement utilise le multi-ensemble des glyphes des sous-arbres pour identifier les codes "proches". Comme nous travaillons en s'abstrayant les constantes (c'est-à-dire que toutes les constantes ont le même glyphe), presque tous les arbres partagent ces glyphes. Nous ne pouvons donc pas conserver les empreintes de tous les sous-arbres. Une approche consiste à ne conserver que les empreintes des "gros" sous-arbres, afin d'éviter la pollution induite par les empreintes des feuilles. La définition même de "gros" étant sujette à cautions, nous proposons deux possibilités : (i) ne retenir que les empreintes des sous-arbres ayant au moins un certain pourcentage p de nœuds comparé au nombre de nœuds de l'arbre tout entier (utile quand on ne connaît pas a priori la taille des définitions étudiées) ; ou bien (ii) ne retenir que les empreintes des sous-arbres ayant un nombre n de nœuds (utile quand on ne s'intéresse qu'aux fonctions dont on connaît la taille). Il s'agit d'un paramètre global de Asak.

4 Partitionnement de corpus dirigé par la similarité

Partitionner un corpus vis-à-vis de l'égalité entre empreintes permet déjà de regrouper des programmes syntaxiquement distincts mais qui ont des structures calculatoires très proches, sinon égales. Nous souhaitons aller plus loin en regroupant des programmes qui se ressemblent même si leurs structures respectives diffèrent plus significativement. Considérons par exemple :

```
1 let id1 x = print_endline "debug"; x and id2 x = x
```

`id1` et `id2` n'ont pas le même glyphe mais on peut difficilement ignorer leur ressemblance !

Comme nous l'avons expliqué précédemment, nous utilisons un algorithme de partitionnement hiérarchique ascendant qui a besoin d'une notion de dissimilarité pour fonctionner.

Definition 3
Soient X et Y deux empreintes. La *dissimilarité* $d(X,Y)$ entre X et Y est une valeur de $\mathbb{N}\cup\{\infty\}$ définie comme suit :

$$d(X,Y) = \sum_{g \in X \Delta Y} w(g) \text{ ou } \infty \text{ si } X \cap Y = \emptyset$$

où $X\Delta Y$ est la différence symétrique entre deux multi-ensembles X et Y.

Cette définition n'est pas très standard car elle sépare de façon très brutale les programmes qui ne partagent aucun sous-terme. Ce choix garantit que deux programmes sans rapport ne pourront jamais apparaître dans la même classe. Ainsi, le résultat final sera composé d'une union

disjointe de classes infiniment distantes les unes des autres. Nous pensons que cette propriété améliore la lisibilité des résultats.

Notez que cette notion de dissimilarité n'est pas une distance. En effet, elle ne vérifie pas l'inégalité triangulaire. Par contre, c'est une fonction de séparation ($\forall X, Y,\ d(X,Y) = 0 \iff X = Y$) et symétrique ($\forall X, Y,\ d(X,Y) = d(Y,X)$). Cette propriété est suffisante pour appliquer l'algorithme de partitionnement.

L'algorithme procède par itération sur une liste de classes d'empreintes. On suppose que l'on sait fusionner deux classes d'empreintes et que l'on garde trace de ces fusions pour pouvoir produire un dendrogramme par classe.

```
(* Types pour les classes d'empreintes et les partitionnements. *)
type cluster and clustering
(* L'opération de fusion. *)
val merge : cluster -> cluster -> clustering -> clustering
(* Un partitionnement initial avec une empreinte par classe. *)
val initial : fingerprint list -> clustering
(* Le nombre de classes d'un partitionnement. *)
val size : clustering -> int
```

Il faut étendre la notion de dissimilarité aux paires de classes.

Definition 4
La dissimilarité $d(\alpha, \beta)$ entre deux classes d'empreintes α et β est :

$$d(\alpha, \beta) = \max_{X \in \alpha, Y \in \beta} d(X, Y)$$

Cette définition de la dissimilarité inter-classes est classique et donne lieu à un partitionnement dit "à liaison complète" (*complete linkage clustering*) [15].

Enfin, on suppose l'existence d'une fonction capable de déterminer les deux classes les plus proches dans le partitionnement courant :

```
type dissimilarity = Regular of int | Infinity
(* Renvoie les deux clusters les plus proches selon d, ainsi que leur distance *)
val get_closest_with_d : clustering -> (dissimilarity * (cluster * cluster))
```

L'algorithme de partitionnement hiérarchique est donné par le programme OCaml suivant :

```
(* Renvoie une liste de classes deux à deux infiniment dissimilaires. *)
let rec make_clustering xs =
  if size xs <= 1 then xs else match get_closest_with_d xs with
    | (Infinity, _) -> xs
    | (Regular p, (u, v)) -> make_clustering (merge u v xs)
let clustering fs = make_clustering @@ initial fs
```

Cette formulation de l'algorithme est malheureusement trop naïve pour être implémentée directement. En effet, en pire cas, la complexité de cet algorithme est cubique en fonction de la longueur de la liste `fs`, c'est-à-dire du nombre de définitions du corpus et dans nos cas d'usage, ce nombre de l'ordre de 10^6.

Heureusement, il est possible d'en fournir une version optimisée (restant de complexité cubique néanmoins). Cette optimisation s'appuie sur les quatre étapes suivants :

1. On commence par séparer les empreintes en deux ensembles : celles qui sont infiniment dissimilaires des autres et celles qui ne le sont pas. Cette phase permet d'éliminer les définitions qui ne partagent absolument rien avec les autres, et sont donc exclues de toute forme de redondance. Cette passe a un coût quadratique en le nombre D d'empreintes.

2. Les dissimilarités inter-empreintes peuvent être précalculées une fois pour toute pour un coût quadratique en le nombre D d'empreintes. Ce précalcul, qui est effectué en parallèle sur plusieurs cœurs, permet de répondre en temps logarithmique la valeur de $d(X, Y)$.
3. On calcule ensuite une surapproximation du partitionnement : à l'aide d'une structure de type *Union-Find*, on regroupe les empreintes qui ont une interdissimilarité finie. On sépare ainsi les empreintes qui ne pourront jamais être dans la même classe car elles sont infiniment dissimilaires deux-à-deux. Ce calcul a un coût en $O(D^2 \cdot logD \cdot \alpha(D))$.
4. Pour finir, l'algorithme de partitionnement naïf est exécuté en parallèle sur toutes les classes du partitionnement sur-approximé. Cette étape a un coût cubique en la taille de chaque partition, ce qui réduit significativement le temps d'exécution de l'algorithme car, sur nos exemples, la cardinalité des classes ne dépasse jamais 10^4.

5 Partitionnement de code étudiant

Motivation Une centaine d'étudiants de troisième année suivent le cours de programmation fonctionnelle. Ils ont deux heures de travaux pratiques par semaine produisant alors plusieurs centaines de réponses. Comment l'enseignant peut-il efficacement analyser ces réponses pour comprendre les difficultés rencontrées par ses étudiants ?

L'utilisation de LearnOCaml [3] est déjà d'une grande aide car la correction automatique des exercices permet de voir les pourcentages de réussite de chaque étudiant. Malheureusement cela ne suffit pas toujours car environ 80% des étudiants obtiennent tous les points ! Une analyse plus poussée vise à déterminer *comment* les étudiants ont répondu. Nous avons donc intégré Asak à LearnOCaml afin de classifier les codes des étudiants. En observant les représentants des classes obtenues et leur taille, l'enseignant peut se faire rapidement une idée de la façon dont les élève ont répondu et ainsi identifier les élèves qui ont fourni une réponse inattendue et qui nécessitent peut-être plus d'attention.

Approche utilisée Partant de l'hypothèse que les tests automatiques ont été bien écrits, nous classifions une première fois les codes selon la note qu'ils ont obtenue. En effet, nous ne voulons jamais identifier deux codes qui n'ont pas eu la même note, même s'ils sont similaires syntaxiquement. Cette première passe nous permet aussi d'avoir pour chaque classe une hypothèse, très forte, d'équivalence sémantique. Cette hypothèse nous permet de raffiner la fonction de calcul d'empreinte afin d'identifier encore plus de codes.

Amélioration de la fonction de calcul d'empreinte Comme nous l'avons vu précédemment, la clé de hachage d'un terme Lambda est déduite des clés de hachage de ses sous-termes *combinées dans l'ordre, de gauche à droite*. Dans cette situation, deux termes équivalents sémantiquement et qui ne diffèrent que par l'ordre de certains sous-termes ne sont pas envoyés vers la même empreinte. Pour résoudre ce problème, nous avons modifié notre fonction de prise d'empreintes pour qu'elle trie les empreintes des sous-termes avant de les combiner. Trier ou non les sous-empreintes est un paramètre global de Asak.

Nous avons déjà soulevé le problème du calcul des empreintes des feuilles de l'arbre en section 3.2. Dans ce contexte d'équivalence sémantique, nous pouvons supposer que dans deux arbres de même forme, les identifiants ne sont que des alias les uns des autres. En effet, s'il existait une différence sémantique entre deux identifiants, les deux codes n'auraient pas la même sémantique. Nous avons donc choisi d'associer la même empreinte à tous les identifiants.

Résultats Les exemples de la figure 4 sont tirés d'un corpus plus gros : celui des réponses des étudiants de troisième année à l'exercice "Implémentez la fonction `rev`". 154 réponses ont obtenu tous les points. Sur ce corpus, Asak produit 9 classes dont voici les cardinaux et une description succinte : 95 fois le premier élément est mis à la fin du reste de la liste renversée récursivement ; 41 fois la réponse introduit une fonction auxiliaire récursive terminale ; 5 fois la réponse utilise une fonction auxiliaire non-locale ; 4 fois le résultat de l'appel récursif est stocké dans une variable locale et ajoute le premier élément à la fin ; 3 fois la réponse utilise `List.rev` ; 3 fois la réponse s'appuie sur `List.fold_left` ; 2 fois la réponse s'appuie sur `List.fold_right` ; 1 fois la réponse utilise un `if-then-else` au lieu du filtrage par motifs.

L'enseignant peut déduire plusieurs choses de ce rapport. D'abord que la plupart des étudiants n'ont pas encore acquis le réflexe d'écrire des fonctions sur les listes dont la récursion est en position terminale. Il peut aussi constater qu'une minorité d'étudiants n'arrive pas encore à se détacher du paradigme impératif ou à utiliser l'analyse de motifs plutôt qu'une expression conditionnelle. Enfin, l'enseignant devra aussi revenir sur son code de correction automatique qui ne capture pas le cas de triche – pourtant grossier – où l'étudiant esquive l'exercice en réutilisant la fonction `List.rev`.

Pour des raisons de protection des données de nos étudiants, nous ne pouvons pas publier le jeu de données qui nous a servi à produire ces résultats. Des résultats similaires sont néanmoins reproductibles avec la version de développement de LearnOCaml [3]. Dans le mode "enseignant", il suffit de faire un clic du milieu sur le nom d'un exercice puis de spécifier la fonction à analyser pour obtenir le partitionnement des copies produit par Asak.

6 Détection de redondance dans les paquets OPAM

Approche utilisée Contrairement à l'environnement bien contrôlé des copies LearnOCaml, chaque paquet OPAM a un ensemble de dépendances et une logique de compilation potentiellement complexe. Pour ces paquets, la phase d'extraction des termes Lambda nécessite donc une configuration spécifique de la partie avant du compilateur. Il serait difficile d'automatiser cette configuration à partir du contenu des paquets. Nous avons préféré créer une version personnalisée du compilateur OCaml [2] installable dans *switch* OPAM. Le compilateur modifié instrumente la compilation standard par un mécanisme de normalisation et d'exportation des termes Lambda. Le dossier contenant le fichier source permet au compilateur de connaître le paquet OPAM en cours d'installation. Chaque terme collecté peut ainsi être étiqueté par le nom du paquet OPAM, par le nom du fichier source et par le chemin du module de la définition dont il est issu.

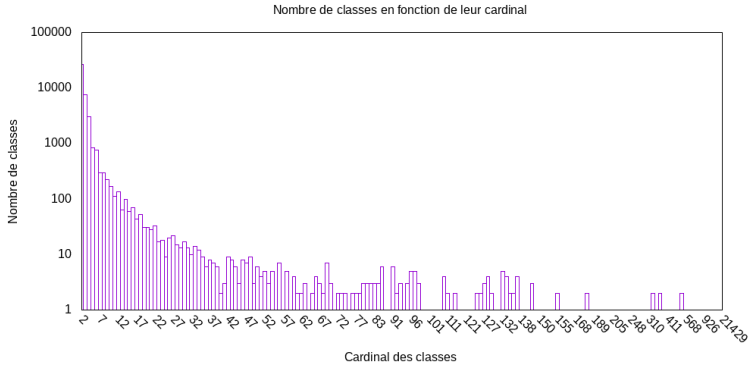
À l'aide d'un serveur de calcul [5], nous avons ensuite parallélisé les (tentatives d')installations de l'ensemble des paquets OPAM. Au bout de 20 heures, nous avons obtenu un corpus de travail dont nous rapportons maintenant l'analyse avec Asak.

Portée de l'analyse Nous avons réussi à compiler 1250 paquets sur les 2428 paquets que compte à ce jour le dépôt OPAM officiel. Tous les paquets n'ont pas pu être installés car certains ne compilent pas avec OCaml 4.08.1 ou sont en conflit avec OCaml 4.08.1 ; ou bien encore parce que certains paquets dépendent de bibliothèques C non-installées.

5. Une machine possédant 40 processeurs Intel(R) Xeon(R) CPU E5-4640 v2 cadencé à 2.20GHz équipé de 756Go de RAM.

Aspects quantitatifs Sur ces 1250 paquets, nous avons extrait 360152 définitions strictement différentes. On a retiré de ce décompte les définitions provenant de deux versions différentes d'un même paquet et qui produisent la même empreinte.

En ne conservant que les empreintes des sous-arbres de plus de 30 nœuds (limite calculable en temps raisonnable de notre implémentation), ces définitions ont été classées en 198841 classes, dont 40935 avec strictement plus d'un élément (c'est-à-dire que nous avons identifié au moins 40935 doublons). Le tout a été calculé en 40 minutes sur la machine décrite précédemment. Voici un diagramme à barre synthétisant la forme des classes obtenues (attention, l'échelle des ordonnées est logarithmique).



Aspects qualitatifs Les classes comprenant plus de 50 éléments sont majoritairement de deux types. Il s'agit d'une part de très petites fonctions (des variables globales ou des fonctions constantes). Comme Asak ne prend pas en compte la valeur des littéraux, ces dernières sont toutes regroupées. L'autre type de redondance est issu du code généré par des outils comme `Coq`, `Menhir`...

Cependant, certaines classes représentent de véritables possibilités de factorisation. Par exemple, nous trouvons une classe contenant 122 éléments comprenant toutes les définitions de la (célèbre) fonction `Option.map`, que l'on trouve sous pas moins de 32 noms différents. Cette classe capture aussi les fonctions similaires concernant les types isomorphes au type `option`. Par exemple, cette classe contient la fonction `map_evar_body` (provenant du module `Evd` de `Coq`) :

```
1 let map_evar_body f =
2   function Evar_empty -> Evar_empty | Evar_defined d -> Evar_defined (f d)
```

Une synthèse complète de l'analyse des paquets OPAM par Asak sera l'objet de travaux futurs. La reproduction des résultats peut s'obtenir en suivant la procédure décrite en ligne à l'adresse https://github.com/nobrakal/asak/blob/master/utils/README.md.

7 Limitations

Nous programmons souvent en utilisant des constructions semblant similaires mais qui sont belles et bien des constructions sémantiquement différentes. Cependant Asak est par définition très sensible à la forme de l'arbre Lambda et ce "presque-sucre syntaxique" fait différer les arbres de codes pourtant très proches. Il en résulte qu'Asak n'est *pas* adapté à la détection de plagiat. Voici deux illustrations.

L'ordre des définitions L'ordre des définitions locales dans une fonction parait anodin mais il influe grandement le calcul d'empreinte puisque celui-ci est effectué récursivement. Il serait alors simple pour quelqu'un désirant fausser les résultats de réorganiser le code afin que sa sémantique reste préservée mais les arbres Lambda produit engendrent des empreintes différentes.

L'expansion des définitions Nous remplaçons souvent mentalement les variables par leur définition. Il s'agit cependant d'une opération complexe qui affecte souvent la sémantique du programme et que le compilateur se risque rarement à faire. Il en résulte que deux codes ayant exactement la même sémantique, l'un ayant une multitude de définitions factorisées et l'autre non engendrent des arbres très différents et donc des empreintes différentes.

8 Travaux connexes

La comparaison de code est un sujet très étudié. Différentes analyses détaillées des techniques existantes ont déjà été faites par Roy et al. [16] et Gautam et al. [9]. On peut grouper les différentes approches en quatre grandes familles qui utilisent chacune plus ou moins la sémantique du langage dans lequel le code est écrit en fonction de contraintes de performance ou de généricité.

Approche textuelle Il s'agit ici de comparer directement les chaînes de caractères composant le code, avec le plus souvent un pré-traitement visant à supprimer les espaces inutiles et les commentaires. Cette approche est la seule indépendante du langage (à l'exception de la phase de pré-traitement). Elle a été mise en œuvre dans des outils comme Duploc [6].

Approche lexicale D'autre outils choisissent de travailler sur la séquence de lexèmes correspondant au code. Ces outils deviennent donc dépendant d'un langage mais permettent de résoudre certains problèmes de l'approche purement textuelle (on est par exemple capable d'identifier deux codes égaux à α-renommage près). Cette méthode est implémentée dans des outils comme CC-Finder [12], CP-Miner [14] et DUP [17].

Approche syntaxique Une autre approche est d'utiliser directement l'arbre de syntaxe abstrait correspondant au code. Elle est donc dépendante du langage et permet de résoudre certains problèmes de l'approche lexicale. Cette méthode a été popularisée avec CloneDr [4] et utilisée plus récemment par Deckard [11].

Approche sémantique Enfin, on peut aussi utiliser les graphes de dépendance [7] du programme. Ces derniers permettent d'introduire beaucoup de sémantique dans la détection de clone mais la technique repose sur la recherche de sous-graphes identiques maximaux, problème qui est NP-complet. Des approximations en temps polynomial permettent néanmoins d'obtenir de bons résultats, comme l'a montré Krinke [13].

9 Conclusion et travaux futurs

Dans cet article, nous avons présenté l'approche suivie par Asak pour la détection de clones de programmes OCaml ainsi que des résultats préliminaires concernant l'application de cet outil

à LearnOCaml et à l'analyse du corpus de paquets OPAM. Pour pallier aux limitations que nous avons explicitées, nous allons continuer à améliorer la prise d'empreintes pour la rendre plus robuste à certaines transformations syntaxiques locales qui préservent la sémantique. Enfin, en intégrant Asak à un outil comme Merlin, nous allons proposer au programmeur un outil pour éviter d'introduire de la redondance dans les programmes OCaml.

Pour finir, les auteurs remercient la Fondation OCaml et ses sponsors pour avoir rendu possible le stage de licence d'Alexandre Moine sans lequel Asak n'aurait pas pu voir le jour.

Références

[1] Asak. https://github.com/nobrakal/asak.

[2] Compilateur OCaml pour la collecte de termes Lambda. https://github.com/nobrakal/ocaml branche 4.08.

[3] LearnOCaml. https://github.com/ocaml-sf/learn-ocaml.

[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, Nov 1998.

[5] Michel Chilowicz, Étienne Duris, and Gilles Roussel. Syntax tree fingerprinting : a foundation for source code similarity detection. Technical report, 2009.

[6] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 109–118, Aug 1999.

[7] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3) :319–349, July 1987.

[8] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In Andrew Kennedy and François Pottier, editors, *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, pages 12–19. ACM, 2006.

[9] Pratiksha Gautam and Hemraj Saini. Various code clone detection techniques and tools : A comprehensive survey. pages 655–667, 08 2016.

[10] Andrew Hunt and David Thomas. *The Pragmatic Programmer : From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[11] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard : Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105, May 2007.

[12] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder : a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7) :654–670, July 2002.

[13] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309, Oct 2001.

[14] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner : finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3) :176–192, March 2006.

[15] Chandan K. Reddy and Bhanukiran Vinzamuri. A survey of partitional and hierarchical clustering algorithms. In Charu C. Aggarwal and Chandan K. Reddy, editors, *Data Clustering : Algorithms and Applications*, pages 87–110. CRC Press, 2013.

[16] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools : A qualitative approach. *Science of Computer Programming*, 74(7) :470 – 495, 2009.

[17] Brenda S. Baker. On finding duplication and near-duplication in large software systems. *Reverse Engineering - Working Conference Proceedings*, 08 2001.