Fouille au code OCaml par analyse de dépendances

Maxence Guesdon[1]

1: Service Expérimentations et Développements
INRIA Paris-Rocquencourt
Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex, France
maxence.guesdon@inria.fr

Résumé

Dans le cadre de la reprise d'un logiciel existant écrit en OCaml, une remise au propre de ce code s'avère nécessaire avant de poursuivre le développement du logiciel. Dans ce but, un outil permettant de trouver les éléments utilisés ou non serait utile pour gagner du temps.

Nous présentons une analyse de code OCaml basée sur la construction d'un graphe de dépendances annoté. Les sommets de ce graphe sont les éléments du langage OCaml (valeurs, modules, types, ...) et les annotations sur les arcs permettent de préciser les dépendances (utilisation, héritage, ...).

Par la suite, un langage de sélection des éléments de ce graphe est défini, permettant de trouver les éléments non référencés (révélateurs de code potentiellement inutile), d'autres éléments comme les champs de types enregistrements jamais consultés.

Cette analyse est implémentée et disponible dans le logiciel libre Oug. Quelques traitements statistiques sur le graphe annoté sont évoqués pour tenter de faciliter la compréhension de l'organisation du code analysé.

1. Introduction

Le travail présenté se situe dans le cadre de la reprise d'un logiciel existant, que nous appellerons L, écrit en OCaml. Une remise en forme du code de L et la fusion de deux branches contenant des évolutions majeures étaient nécessaires.

Pour cette tâche, un outil d'analyse de dépendances dans du code OCaml permettrait un gain de temps appréciable. Nous présentons dans la section 2 les détails du problème, des analyses existantes s'en rapprochant ainsi que la solution envisagée, basée sur la construction d'un graphe de dépendances entre les différents éléments du langage (valeurs, modules, classes, etc.).

La section 3 présente la construction de ce graphe, enrichie au fur et à mesure des tests et des réflexions. En section 4, nous expliquons comment exploiter ce graphe pour effectuer les analyses qui nous intéressent, notamment en définissant un langage de sélection de sommets (éléments) sur le graphe. Enfin, la section 5 donne des informations sur l'implémentation de cette analyse dans un outil en ligne de commande et le résultat de l'utilisation de ce dernier sur le logiciel L.

2. Le besoin

2.1. Le problème

Notre logiciel L contient presque 60.000 lignes de code OCaml écrites par plusieurs personnes successives avec des styles différents (cas classique de CDD à répétition).

Le résultat est :

1. un code peu homogène,
2. des redondances (le même calcul implémenté dans plusieurs fonctions, par exemple), avec comme cause principale la méconnaissance et le manque d'organisation claire de l'existant,
3. du code inutile, qui provoque tout de même des erreurs de compilation lors de changements dans les structures de données et est donc maintenu pour passer l'étape de la compilation,
4. des champs d'enregistrements qui ne sont plus utiles et qui pourtant sont calculés,
5. des constructeurs qui ne sont plus utiles mais sont encore présents comme autant de cas de pattern-matching.

Le code inutile évoqué au point 3 peut être du code mort, c'est-à-dire jamais exécuté, comme la fonction `f` dans l'exemple suivant :

```
let f () = print_string "hello";;
(* fin du programme *)
```

Le code inutile inclut également du code calculant une valeur qui n'est jamais utilisée par la suite, comme la valeur `foo` dans :

```
let f x = x + 1;;
let foo = f 1;;
(* la suite du programme n'utilise pas foo *)
```

Nous souhaitons donc pouvoir trouver les éléments non référencés (donc inutiles), révélateurs de code *potentiellement* inutile. En effet, le code des éléments non référencés est inutile si ce code est purement fonctionnel, mais seulement *potentiellement* inutile s'il contient des effets de bord. Ainsi, dans le code suivant

```
let f x = print_int x ; x + 1;;
let g x = let foo = f x in x + 1;;
```

la valeur `foo` est inutile mais, son calcul ayant un effet de bord, on ne peut simplement supprimer "`let foo = f x in`" sans changer la sémantique du programme.

Les points 1 à 5 représentent un coût supplémentaire en terme de temps de développement et de maintenance. De plus, les points 3, 4 et 5 ont un impact sur les performances.

Une remise au propre du code de L implique d'intervenir sur ces différents points. L'amélioration de l'homogénéité du code et la détection de redondances nécessitent une bonne connaissance de L ; cependant, cette connaissance est plus facile à acquérir sans les entraves que représentent les autres points, que l'on souhaite donc traiter en priorité.

La recherche du code et des champs inutiles étant fastidieuse à la main, on souhaite avoir un outil pour l'automatiser. De plus, un tel outil permettrait de contrôler régulièrement l'apparition de telles anomalies pour les traiter rapidement.

Un autre problème de L est l'existence de deux développements majeurs en parallèle, qu'il faut fusionner. Malheureusement, l'un de ces développements, D_1, introduit de nouvelles structures

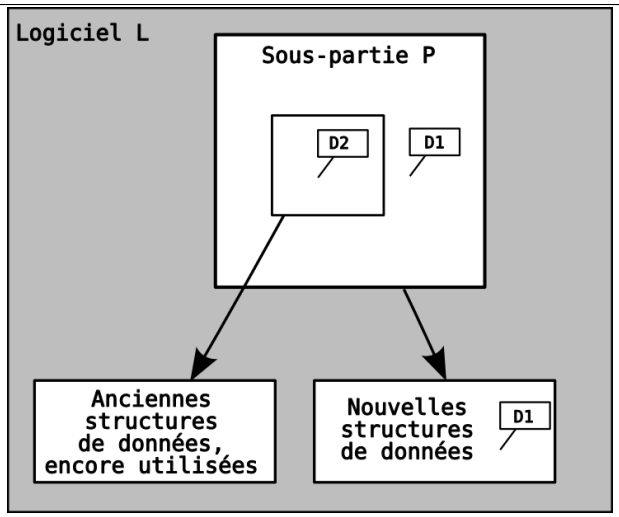

FIG. 1 – Situation du code juste après la fusion des deux développements réalisés en parallèle. On souhaite que le code du développement D_2 utilise maintenant les nouvelles structures de données comme le reste de la partie P du logiciel.

de données utilisées dans une sous-partie P du logiciel à la place d'anciennes, tandis que l'autre développement, D_2, impacte la même sous-partie P. Les anciennes structures de données sont encore utilisées et la plupart des données sont dans une référence globale à l'application, ce qui ne permet pas de s'appuyer sur le typage pour détecter, dans le deuxième développement, le code utilisant les anciennes structures alors qu'il devrait utiliser les nouvelles. La situation est représentée sur la figure 1. On souhaite donc savoir où les modules concernés par D_2 accèdent aux anciennes structures de données, soit directement, soit, plus souvent, par l'intermédiaire de fonctions qui, elles, accèdent à ces anciennces structures. Cette problématique se rapproche de la recherche des champs inutilisés, mais inversée : il s'agit de savoir, pour chaque champ des anciennes structures, quels éléments de D_2 y font référence directement ou indirectement.

2.2. Les analyses existantes

Notre but est de construire une représentation du logiciel L pour en extraire les informations dont nous avons besoin pour faciliter la compréhénsion du code et supprimer les parties inutiles.

Les techniques de suppression de code mort basées sur des analyses locales de vivacité et la propagation de constantes comme dans [1] ne nous conviennent donc pas, d'autant plus qu'elles sont souvent faites sur du code généré et non sur du code source.

Une analyse de dépendances sur du code ML est exposée dans [3] mais vise à ordonner les modules pour décharger le développeur de cette tâche. On y retrouve les problèmes de portée et de masquage que nous évoquons plus loin.

La création d'un graphe de dépendances (pour abstraire la structure d'un programme C) et la possibilité de requêtes sur ce graphe (pour faciliter la maintenance d'un logiciel) sont décrites dans [2].

La solution que nous avons adoptée est proche de cette dernière méthode. Cependant, l'aspect premier ordre du langage OCaml rend difficile la création d'un graphe d'appels, qui nécessiterait une analyse du flot de données. D'autre part, les constructions d'OCaml nécessitent de définir des

dépendances entre éléments qui soient adaptées au langage et à nos besoins.

2.3. L'idée de solution envisagée

Il n'existe aucun outil pour effectuer les analyses et recherches dont nous avons besoin pour du code OCaml. Une façon de faire est d'utiliser la commande UNIX `grep` mais le résultat affiché nécessite d'aller voir le code en question pour étudier dans quel contexte un identifiant recherché est utilisé.

Les fichiers `.annot` produits par l'option `-annot` du compilateur OCaml pourraient représenter une autre piste pour obtenir les informations nécessaires mais ils ne contiennent pas (encore) l'information indiquant où est déclarée un identifiant, et de toutes façons ils ne donnent pas le contexte d'utilisation, par exemple si un champ d'enregistrement est lu ou modifié.

A noter que le compilateur OCaml contient déjà la détection des variables non utilisées, mais il ne s'agit que des variables "locales", c'est-à-dire celles dans une classe ou une autre variable. Ainsi, dans le code suivant :

```
let foo =
  let (y, z) = (3, 4) in
  z;;
let bar gee = 3;;
```

les variables `y` et `gee` sont indiquées comme non utilisées, mais pas `foo` ni `bar`. Cela s'explique facilement puisque ces deux variables sont encore dans la portée lexicale du code qui suit et même accessibles depuis un autre module. En présence d'un fichier `.mli`, le compilateur pourrait détecter que ces variables ne sont ni utilisées dans le module ni présentées dans l'interface, mais cette fonctionnalité n'est pas encore disponible.

L'idée, assez évidente, pour détecter les cas présentés en section 2.1 est donc de construire un graphe de dépendances entre les différents éléments du code du logiciel L. Les types d'éléments qui nous intéressent sont les suivants : modules, valeurs, classes, méthodes, variables d'instance, types, champs de type (champ pour un type enregistrement, constructeur pour un type variant). Nous laissons de côté aujourd'hui les exceptions et les types variants polymorphes.

A partir de ce graphe, nous pourrons ensuite rechercher les éléments qui ne sont pas référencés afin de répondre aux points 3, 4 et 5 évoqués plus haut, ainsi que les éléments utilisant des champs de types donnés, dans le cadre de la problématique de fusion de code exposée précédemment.

Présentons maintenant la construction de ce graphe.

3. Construction du graphe

L'analyse de dépendances dont nous avons besoin nécessite d'analyser le code d'implémentation (les fichiers `.ml`) de notre logiciel L. Nous verrons dans la suite que nous ajouterons un traitement supplémentaire utile pour les bibliothèques, utilisant les fichiers d'interface `.mli`.

Les sommets du graphe représenteront les éléments (modules, valeurs, etc.) tandis que les arcs représenteront les relations de dépendances entre les éléments.

Dans un premier temps, on considère seulement les éléments "exportables", c'est-à-dire nommables dans le fichier `.mli` correspondant au fichier `.ml` analysé. Le code suivant donne quelques exemples d'éléments exportables ou non :

```
let (x, y) = let z = 1 in (z+1,0);; (* x et y sont exportables, z ne l'est pas *)
module P = struct (* P est exportable ... *)
  type t = int  (* ... de même que t ... *)
```

Code	Relations

`let x = 3 ; ; let y = x + 1 ; ;`
`let y = x + 42 ; ;`
`let f x = x + y ; ;`
`module M = struct`
`  let g x = f (f (x + 2))`
`  let h t u = f (x + t) + g u`
`end ; ;`
`type t = One | Two ; ;`
`let rec int_of_t = function`
`  One -> 1`
`| Two -> 1 + int_of_t One ; ;`
	$y \rightarrow x$
	$y_2 \rightarrow x$
	$f \rightarrow y_2$
	$M \rightarrow M.g$, $M.g \rightarrow f$
	$M \rightarrow M.h$, $M.h \rightarrow x$, $M.h \rightarrow f$, $M.h \rightarrow M.g$
	
	$t \rightarrow t.One$, $t \rightarrow t.Two$
	
	$int\_of\_t \rightarrow t.One$
	$int\_of\_t \rightarrow t.Two$,
	$int\_of\_t \rightarrow int\_of\_t$

FIG. 2 – Production d'arcs de dépendance simples.

```
  let compare = Pervasives.compare (* ... et que compare *)
end;;
let f x = (* f est exportable *)
  let module M = Set.create(P) in (* M ne l'est pas *)
  ...;;
module S = Set.Make (* S est exportable *)
  (struct type t = int let compare = Pervasives.compare end)
  (* ce module anonyme n'est pas exportable *)
;;
```

3.1. Graphe simple

Une première approche, naïve, consiste à utiliser des arcs simples pour indiquer que, d'une façon ou d'autre autre, un élément `t` dépend de `u` (relation notée $t \rightarrow u$). La figure 2 donne quelques exemples de code et des relations qui en sont extraites. De plus, si le code de cet exemple est dans un fichier `f.ml`, il faut également ajouter le module résultant `F` et les relations de dépendances vis-à-vis des éléments qu'il contient, en préfixant le nom de ces derniers. On obtient au final : $F \rightarrow F.x$, $F \rightarrow F.y$, $F.y \rightarrow F.x$, $F \rightarrow F.M$, $F.M \rightarrow F.M.g$, etc. On peut remarquer que, même si `F.M.g` contient plusieurs références à `F.f`, un seul arc est ajouté. En effet, il est inutile de garder les références multiples entre deux éléments dans notre cas. D'autre part, la définition d'une seconde valeur `F.y` crée un sommet supplémentaire dans le graphe, distinct du sommet correspondant à la première valeur `F.y`, que nous appelons $F.y_2$ pour le distinguer à l'affichage dans l'exemple. Nous verrons comment ce phénomène de masquage est traité dans l'implémentation.

Ces arcs "simples" ne permettent cependant pas de mener l'analyse que nous souhaitons. En effet, puisque chaque élément (à part les modules de plus haut niveau) est contenu dans un autre élément, chaque élément a donc un prédécesseur. On pourrait imaginer ne pas ajouter les arcs représentant la relation de contenance d'un élément par un autre, mais on ne pourrait alors dire qu'un module est utile car il contient un élément utilisé.

Il nous faut donc préciser les types de dépendances entre éléments.

3.2. Graphe annoté

Nous définissons plusieurs types d'annotations pour les arcs, afin de garder des informations plus précises sur les relations entre éléments :

- `t` $\xrightarrow{contain}$ `u` indique que la définition de l'élément `t` contient la définition de l'élément `u` ; c'est le cas dans l'exemple précédent pour `F` et `F.x`, la définition de `F` contient la définition de `F.x` : `F` $\xrightarrow{contain}$ `F.x`.
- `t` $\xrightarrow{include}$ `u` indique que l'élément `t` inclut l'élément `u` ; c'est le cas pour les inclusions de modules, ainsi que pour l'application de foncteurs : on considèrera que le foncteur et les arguments sont dans le module résultant (il n'est pas apparu le besoin d'avoir une relation spéciale pour lier un module `M` et les arguments passés au foncteur pour créer `M`).
- `c1` $\xrightarrow{inherit}$ `c2` indique une relation entre deux classes : `c1` hérite de `c2`.
- `t` $\xrightarrow{alias}$ `u` indique que l'élément `t` est une sorte d'alias pour l'élément `u`. Cette relation apparaît lors de l'application d'un foncteur. Le module résultat contient des éléments qui sont des aliases des éléments du foncteur appliqué. Par ailleurs, le cas où un module est créé par `"module B = A"` entraîne la création d'un arc "alias" pour conserver cette information : `B` $\xrightarrow{alias}$ `A`.
- `t` $\xrightarrow{use}$ `u` indique que l'élément `t` dépend de `u` d'une autre manière que celles ci-dessus, notamment quand une valeur en utilise une autre dans sa définition.

La figure 3 montre ce que devient l'exemple utilisé pour illustrer la construction du graphe sans annotation, complété par des exemples de manipulations de modules, en supposant que le code se trouve dans un fichier `f.ml`.

On peut remarquer que même si l'application d'un foncteur d'une part et la création d'un module par `include` explicite (`"module E = struct include D end"`) d'autre part entraînent toutes les deux la création d'arcs $\xrightarrow{include}$, un traitement différent est fait pour les éléments des modules inclus. Dans le cas d'un `include` explicite, le nouveau module ne contient aucun élément. En effet, ajouter des éléments au module `E` n'est pas utile puisque ces éléments auront les mêmes dépendances que les éléments du module `D` inclus.

Au contraire, dans le cas d'un module créé par application d'un foncteur, on crée pour ce module résultant autant d'éléments qu'en définit le foncteur, et chacun de ces éléments créés est lié à l'élément original du foncteur par un arc $\xrightarrow{alias}$. Cela est nécessaire car les éléments du module résultant de l'application n'auront pas forcément les mêmes dépendances que les éléments d'un autre module résultant de l'application du même foncteur mais avec un module différent en paramètre. La relation d'alias permettra de voir si un élément d'un foncteur est utilisé, car tant que le foncteur n'est pas appliqué, ses éléments ne peuvent être référencés depuis l'extérieur du foncteur.

De la même façon, la création d'un module `B` par `"module B = A"` n'entraîne pas la création d'éléments dans `B` car il ne s'agit que de la création d'un raccourci de nommage. Il serait toutefois possible d'appliquer le même traitement que pour l'application de foncteur, comme si le module `A` était un foncteur sans argument.

Ces annotations nous permettent donc de trouver :

- les éléments exportables qui ne sont pas utilisés (c'est-à-dire les éléments qui n'ont aucun prédécesseur par certaines relations, comme nous le détaillons en section 4.2),
- les éléments qui, directement ou transitivement, utilisent certains champs de types.

Remarque On ne prend pas en compte dans le graphe les paramètres des fonctions, c'est-à-dire que ces derniers n'ont pas de sommet dans le graphe pour les représenter. L'aspect ordre supérieur d'OCaml n'est donc pas directement abordé, comme le montre l'exemple suivant :

```
let conv = int_of_string;;
let f g x = print_int (g x);;
```

Code	Relations
`let x = 3 ; ; let y = x + 1 ; ;`	F.y $\xrightarrow{use}$ F.x
`let y = x + 42 ; ;`	F.y_2 $\xrightarrow{use}$ F.x
`let f x = x + y ; ;`	F.f $\xrightarrow{use}$ F.y_2
`module M = struct`	
`  let g x = f (f (x + 2))`	F.M $\xrightarrow{contain}$ F.M.g, F.M.g $\xrightarrow{use}$ F.f
`  let h t u = f (x + t) + g u`	F.M $\xrightarrow{contain}$ F.M.h,
	F.M.h $\xrightarrow{use}$ F.x, F.M.h $\xrightarrow{use}$ F.f, F.M.h $\xrightarrow{use}$ F.M.g
`end ; ;`	
`type t = One | Two ; ;`	F.t $\xrightarrow{contain}$ F.t.One, F.t $\xrightarrow{contain}$ F.t.Two
`let rec int_of_t = function`	
`  One -> 1`	F.int_of_t $\xrightarrow{use}$ F.t.One
`| Two -> 1 + int_of_t One ; ;`	F.int_of_t $\xrightarrow{use}$ F.t.Two,
	F.int_of_t $\xrightarrow{use}$ F.int_of_t
`module A =`	
`  struct let x = 1 let y = x end ; ;`	F.A $\xrightarrow{contain}$ F.A.x, F.A $\xrightarrow{contain}$ F.A.y
`module B = A ; ;`	F.B $\xrightarrow{alias}$ F.A
`let g x = x + B.y ; ;`	F.g $\xrightarrow{use}$ F.A.y (car F.B est un alias pour F.A)
`module Fo`	
`  (P : sig val p : int -> int end) =`	
`  struct let f x = P.p x + A.x end ; ;`	F.Fo $\xrightarrow{contain}$ F.Fo.f, F.Fo.f $\xrightarrow{use}$ F.A.x
`module C = struct let p x = x + 1 end ; ;`	F.C $\xrightarrow{contain}$ F.C.p
`module D = Fo(C) ; ;`	F.D $\xrightarrow{include}$ F.C, F.D $\xrightarrow{include}$ F.Fo, F.D $\xrightarrow{contain}$ F.D.f,
	F.D.f $\xrightarrow{alias}$ F.Fo.f, F.D.f $\xrightarrow{use}$ F.C.p, F.D.f $\xrightarrow{use}$ F.A.x
`module E = struct include D end ; ;`	F.E $\xrightarrow{include}$ F.D
`let h = E.f ; ;`	F.h $\xrightarrow{use}$ F.D.f (car F.E.f est en fait F.D.f)

FIG. 3 – Production d'arcs de dépendance annotés.

	Champ d'un enregistrement	Constructeur d'un type variant
"Lecture"	Lecture d'un champ : **v.field** `+ 1`	Filtrage : match x with **Const** → ...
"Création"	Initialisation/affectation : **v.field** ← `1`	Construction : `let x =` **Const** `in ...`

FIG. 4 – Cas de "lecture" et "création" de champs de types.

```
let h s = f conv s;;
```

Le graphe montrera les dépendances `h` $\xrightarrow{use}$ `conv` et `h` $\xrightarrow{use}$ `f` mais pas `f` $\xrightarrow{use}$ `conv`. Cet aspect est évoqué dans la conclusion.

3.3. Trouver les champs de type inutiles

Les annotations définies plus haut ne résolvent pas complètement les points 4 et 5 exposés en section 2.1 concernant l'utilité des champs de types définis dans le code existant.

En effet, OCaml ne permet pas la création d'une valeur de type enregistrement si tous les champs du type ne sont pas définis. Si au moins une valeur de ce type est créée, il existe alors toujours au moins une valeur qui définit tous les champs. En conséquence, il n'est pas possible avec les annotations d'arcs définies plus haut de voir si un champ d'un enregistrement est utilisé et pas seulement initialisé ou modifié.

Par ailleurs, le compilateur OCaml permet de savoir quand un pattern-matching est incomplet. Une bonne pratique de codage consiste à indiquer explicitement tous les constructeurs d'un type variant, même si plusieurs se traitent de la même façon, afin de s'assurer du traitement de tous les cas lors de l'ajout d'un constructeur. Ainsi, de façon inverse aux champs d'un type enregistrement, les constructeurs d'un type variant sont souvent tous filtrés à un moment ou à un autre. Le problème vient cette fois du fait que l'on recherche les constructeurs qui ne sont pas utilisés pour créer une valeur du type en question.

Il nous faut donc distinguer la dépendance par rapport à un champ de type selon qu'on le "crée" ou bien qu'on le "lit". La figure 4 précise le sens de "créer" et "lire" selon le type de champ concerné. La "lecture" d'un champ entraîne la création d'un arc $\xrightarrow{use}$, tandis que la "création" d'un champ provoque l'ajout d'un arc avec une nouvelle annotation, $\xrightarrow{create}$. Pour trouver les champs d'enregistrement inutiles, on cherchera donc ceux qui n'ont aucun prédécesseur par la relation $\xrightarrow{use}$. Le cas où aucune valeur d'un champ enregistrement n'est créée sera détecté en cherchant les champs sans prédécesseur par la relation $\xrightarrow{create}$.

Pour les constructeurs, sous l'hypothèse que le développeur s'est attaché à traiter explicitement chaque constructeur lors des filtrages, on s'intéressera aux constructeurs qui ne sont jamais utilisés pour créer une valeur du type en question. Ce sont les champs de type variant qui n'ont aucun prédécesseur par la relation $\xrightarrow{create}$. Cependant, cette détection n'est pas tout à fait complète, comme le montre l'exemple suivant :

```
type t = One | Two  ;;
let is_two x = x = Two ;;
```

Ici, le constructeur `Two` est en position de "création" et non en position de filtrage. Si on teste le cas de `Two` à l'aide de la fonction `is_two` et que ce constructeur n'est utilisé nulle par ailleurs pour créer une valeur du type `t`, le graphe ne permettra pas de s'en rendre compte.

La figure 5 montre le graphe de dépendances annoté pour l'exemple de la figure 3, avec les arcs $\xrightarrow{create}$.

Le graphe avec ces annotations nous permet donc de détecter les éléments exportables qui nous

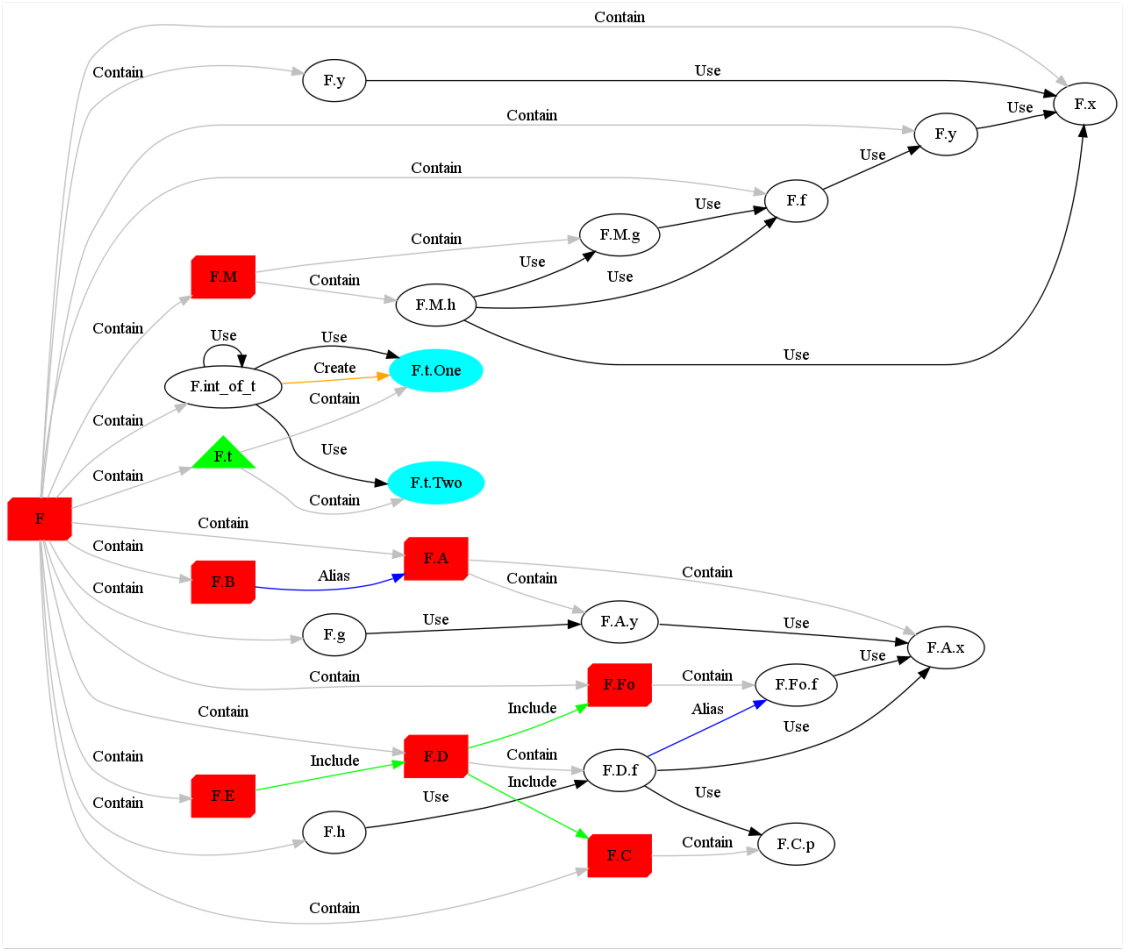

FIG. 5 – Graphe annoté des éléments exportables de l'exemple.

intéressent (modulo l'exception ci-dessus). Cependant, il ne permet pas de détecter d'autres éléments pourtant inutiles, parmi les éléments non "exportables".

3.4. Gestion des éléments non exportables

Considérons le code suivant :

```
let x =
  let module M = struct let y = 1 end in
  let y = Array.create 12 0 in
  1 ; ;
```

Nous avons clairement deux variables inutilisées et inutilisables dans la suite du code : `M.y` et `y`. Le compilateur OCaml ne signale pas le fait que `M.y` est inutilisée. Par ailleurs, le module `M` n'étant pas exportable puisque local à la variable `x`, il n'apparaît pas dans notre graphe. Il faut pourtant détecter ce cas. De même, notre graphe ne prend pas en compte la variable `y`, locale à `x` et non exportable, même si le compilateur OCaml émet un avertissement pour signaler qu'elle est inutilisée.

Ce genre de cas peut être facilement détecté en ajoutant dans notre graphe les sommets correspondant à ces éléments locaux. Pour le dernier exemple ci-dessus, on aura alors les relations suivantes, en utilisant des arcs $\xrightarrow{contain}$ pour indiquer qu'un élément est défini dans un autre élément de façon générale et plus seulement pour les modules, classes et types :

$$\texttt{x} \xrightarrow{contain} \texttt{x.M},\ \texttt{x.M} \xrightarrow{contain} \texttt{x.M.y},\ \texttt{x} \xrightarrow{contain} \texttt{x.y}.$$

Il sera alors facile de détecter que `x.M.y` et `x.y` sont inutilisées, de même que le module `x.M` qui ne contient aucun élément utilisé (ce que n'indique pas le compilateur OCaml).

3.5. Les interfaces entrent en scène

L'analyse du code d'implémentation permet de trouver les éléments inutilisés. Cependant, dans le cas d'une bibliothèque, beaucoup d'éléments sont souvent inutilisés dans la bibliothèque elle-même mais accessibles dans son interface.

Pour éviter de mélanger les éléments non utilisés car seulement offerts par l'interface et ceux correspondant effectivement à du code inutile, il nous faut prendre en compte les fichiers d'interface des modules (fichiers `.mli`) et ajouter une nouvelle annotation, $\xrightarrow{export}$, permettant d'indiquer explicitement les éléments d'interface.

L'ajout de ces annotations est très facile : il suffit de parcourir le fichier d'interface et, pour chaque élément, utiliser son nom complètement qualifié pour ajouter une relation $\xrightarrow{export}$ entre le module ou la classe et l'élément en question. La recherche d'éléments inutilisés est alors modérée par ces arcs $\xrightarrow{export}$.

3.6. A propos des classes

Les exemples ci-dessus n'utilisent pas les classes, mais la construction de sommets dans le graphe pour les classes, méthodes et variables d'instance est intuitive : une classe contient (relation $\xrightarrow{contain}$) des méthodes et variables d'instances, ainsi que des valeurs ou modules locaux comme éléments "non exportables". Les relations d'héritage donnent des arcs $\xrightarrow{inherit}$ qui correspondent plus ou moins à la relation $\xrightarrow{include}$ pour les modules.

Le problème posé par les objets est la résolution dynamique des appels de méthodes. En effet, considérons le morceau de code suivant :

```
class c1 = object method m = 15 end;;
class c2 = object method m = 10 end;;
let obj =
  match Random.int 5 with
    0 -> new c1
  | 1 -> new c2
  | n -> f n (* en supposant f définie et renvoyant un objet du bon type *)
;;
print_int obj#m;;
```

Il n'est pas possible statiquement de déterminer si la méthode `m` invoquée dans l'expression `obj#m` est celle de la classe `c1`, celle de la classe `c2` ou bien celle d'un objet renvoyé par la fonction `f`.

Nous nous "contentons" donc de considérer que toutes les méthodes sont référencées et ne sont jamais inutilisées. Pour cette raison, lorsqu'une classe c_2 hérite d'une classe c_1, nous considérerons que la classe c_1 est utilisée, même si aucune de ses variables d'instances et méthodes n'est utilisée. La relation $\xrightarrow{inherit}$ indiquera donc une utilisation. Cela diffère de la relation $\xrightarrow{include}$ utilisée pour les modules, qui n'induit pas une utilisation du module inclus puisqu'on peut vérifier par le graphe si les éléments du module inclus sont utilisés ou non, et donc si le module lui-même est utilisé.

4. Utilisations du graphe de dépendances

Voyons maintenant comment exploiter le graphe construit dans la section précédente. Nous définissons tout d'abord un langage permettant de sélectionner dans le graphe les sommets (éléments) d'après certaines contraintes. Ensuite, nous utilisons ce langage pour extraire les informations qui nous intéressent (cf. section 2.1).

4.1. Langage de sélection de sommets dans le graphe

Nous souhaitons pouvoir sélectionner les éléments par leur nom et leur type d'une part, par les relations qu'ils ont avec d'autres éléments d'autre part.

Nous définissons donc un langage permettant de construire des filtres à appliquer sur le graphe. Un filtre permet de ne conserver qu'un ensemble de sommets et les arcs entre ces sommets. Les arcs concernant des sommets qui ne sont pas conservés sont retirés.

La notation $< exp >_k$ nous permettra de décrire la sélection des éléments dont le nom correspond à l'expression $< exp >$ et dont le type est l'un de ceux indiqués dans k. Par exemple, $< F.* >_v$ permet d'extraire toutes les valeurs ($k = v$) ayant un nom pleinement qualifié correspondant à l'expression $< F.* >$, c'est-à-dire toutes les valeurs du module `F`. k peut prendre les valeurs suivantes : v (valeur), M (module), C (classe), m (méthode), i (variable d'instance), t (type), f (champ de type).

Les opérations &, |, − et ! seront respectivement les opérations classiques d'intersection, d'union, de différence et de complément sur les ensembles. Ainsi $! < F.* >_v$ donnera tous les éléments analysés sauf les valeurs du module `F`.

Enfin, les contraintes de relations seront exprimées de la façon suivante :

- $f \xrightarrow{d}$: sélectionne les éléments atteignables par la relation de dépendance d, depuis au moins un des éléments sélectionnés par le filtre f.
- $\xrightarrow{d} f$: sélectionne les éléments permettant d'atteindre, par la relation de dépendance d, au moins un des éléments sélectionnés par le filtre f.

La relation de dépendance permet d'indiquer :

- le type d'arc à prendre en compte, par exemple les arcs "Contain" et "Use" dans $\xrightarrow{contain,use}$,

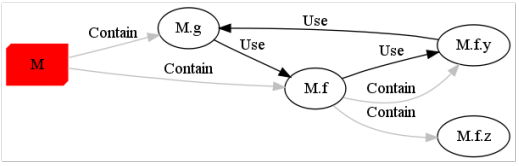

FIG. 6 – Graphe obtenu sur un exemple de fonctions récursives.

- la longeur de chemin, soit par un entier, soit par "+" pour indiquer que la longeur du chemin doit être ≥ 1. Exemple : $\xrightarrow{contain,use\{+\}}$.
- un filtre supplémentaire permettant de sélectionner des éléments par lesquels les chemins intervenant dans la recherche d'éléments ne doivent pas passer. Ainsi, le filtre

$$\xrightarrow{contain,use\{+\}/<F.M.*>} < F.f >_v$$

sélectionne les éléments permettant d'atteindre, par des arcs "Use" ou "Contain", la valeur `F.f`, sans passer par les éléments du module `F.M`.

Dans la suite, par soucis de concision, nous utiliserons la notation $e \rightarrow$ avec e désignant un élément ou un groupe d'éléments du graphe, alors que e devrait désigner un filtre renvoyant l'unique élément ou le groupe d'éléments e.

4.2. Recherche de code inutile

La recherche du code potentiellement inutile, c'est-à-dire les éléments inutiles (car non utilisés et non utilisables), revient à définir les filtres permettant de sélectionner ces éléments dans le graphe construit dans la section 3.

Nous considérons qu'un élément e est inutile si :
- aucun autre élément en dehors des éléments "sous e" ne l'utilise,
- e ne contient que des éléments inutiles.

La notion d'élément "sous e" permet de traiter les cas de récursion, que cela soit pour les définitions de valeurs ou de classes. Considérons le code suivant :

```
let rec f x =
  let z = 1 in
  let y = g (x - 2) in
  y + 1
and g x =
  if x <= 1 then x else f (x + 1) ; ;
```

La figure 6 montre le graphe produit. Les éléments `M.g`, `M.f.z`, `M.f.y` et `M.f` sont "sous `M.f`", c'est-à-dire que le fait que l'un d'eux utilise `M.f` ne permet pas de dire que `M.f` est utilisé, puisque les utilisations forment alors un cycle.

On définit donc les éléments "sous e" de la façon suivante :

$$Sub(e) = e \xrightarrow{export,contain,inherit,include,create\{+\}} |(e \xrightarrow{use,alias\{+\}} \& \xrightarrow{use,alias\{+\}} e).$$

Un élément e est utile si au moins un autre élément du graphe utilise e et ne fait pas partie de $Sub(e)$. Comme chaque élément est contenu dans un autre élément (à part les modules racines), nous ne nous intéresserons pas à la relation $\xrightarrow{contain}$. De plus, la relation $\xrightarrow{include}$ n'induit pas non plus l'utilisation.

Les éléments utilisant e seront donc définis par le filtre suivant :

$$Ref(e) = \xrightarrow{use,create,export,inherit,alias\{+\}} e \ .$$

Cependant, nous devons retirer de cet ensemble les éléments qui sont des aliases de e, car il ne suffit pas que `f` $\xrightarrow{alias}$ `e` pour que e soit utilisé. Il faut qu'un autre élément g utilise cet alias :

$$g \xrightarrow{k,k\neq contain,include,alias} f \xrightarrow{alias} e.$$

On peut généraliser cet exemple au cas où une chaîne d'aliases existe de f vers e, c'est-à-dire que l'on souhaite traiter les cas $f \xrightarrow{alias\{+\}} e$. En d'autres termes, on s'autorise la recherche d'éléments utilisant e par l'intermédiaire d'aliases, mais on considère qu'un alias de e n'utilise pas e. Les aliases de e sont définis ainsi :

$$A(e) = \xrightarrow{alias\{+\}} e.$$

On peut finalement définir $U(e)$, indiquant si e est utile, de la façon suivante :

$$U(e) = card(Ref(e) - A(e) - Sub(e)) > 0.$$

Un élément qui n'est pas utilisé peut tout de même ne pas être inutile, s'il contient lui-même au moins un élément qui n'est pas inutile. C'est souvent le cas d'un module, dont on référence des éléments qu'il contient, mais pas forcément le module lui-même. Il est pourtant utile car l'un de ses éléments est utilisé *en dehors* du module.

Pour chaque élément non utilisé, on fera un calcul supplémentaire pour savoir si tous ses éléments sont inutiles. Cependant, ce test d'utilité de chaque sous-élément doit se faire dans le contexte de l'élément conteneur que l'on teste. En effet, prenons l'exemple suivant :

```
module M = struct
  let f x = x + 1
  let g x = f x + 1
end;;
```

L'élément `M.f` est utile, puisque `M.g` y fait appel. Cependant, lors de l'analyse de `M` pour savoir si ce module contient des éléments utiles, nous devons nous intéresser aux utilisations de ses sous-éléments *en dehors* de `M`. Nous reprenons donc les définitions ci-dessus pour pouvoir les utiliser non pas dans le contexte de l'élément e mais dans celui de son conteneur C. Les éléments "sous e" dans le contexte de C sont alors :

$$Sub(e, C) = C \xrightarrow{export,contain,inherit,include,create\{+\}} |(e \xrightarrow{use,alias\{+\}} \& \xrightarrow{use,alias\{+\}} e)$$

Pour la recherche des éléments référençant e, on ajoute une contrainte, celle de ne pas utiliser l'élément C dans les chemins possibles pour atteindre les éléments en question :

$$Ref(e, C) = \xrightarrow{use,create,export,inherit,alias\{+\}/C} e$$

Ainsi, nous définissons $Cont(e)$ indiquant si l'élément e contient au moins un élément utile (en dehors de e) :

$$Cont(e) = \exists e', e \xrightarrow{contain\{+\}} e' \& \ card(Ref(e', e) - A(e') - Sub(e', e)) > 0$$

Enfin, différents filtres supplémentaires interviennent :
- les méthodes ne sont jamais inutiles (cf. 3.6),
- il est possible et utile de supposer certains éléments comme utiles. Examinons le cas suivant :
  `module P = struct type t = int let compare = Pervasives.compare end ; ;`
  `module MyMap = Map.Make(P) ; ;`
  Si le code du module `Map.Make` n'est pas connu, `P.t` et `P.compare` ne seront jamais référencés.
- pour la même raison, on peut considérer les modules vides comme utiles.

4.3. Recherche des champs de types inutiles

Avec notre graphe des dépendances, la recherche des champs de types non "créés" ou non "lus" est facilitée. Ainsi, les champs de types non créés sont obtenus par le filtre suivant :

$$< * >_f \ \& \ ! < * > \xrightarrow{create} .$$

Ce filtre s'interprète ainsi : l'intersection de l'ensemble des champs de types ($< * >_f$) et de l'ensemble des éléments qu'aucun élément ne crée ($! < * > \xrightarrow{create}$).

De façon symétrique, les champs de types non lus sont obtenus avec le filtre

$$< * >_f \ \& \ ! < * > \xrightarrow{use}$$

où l'on a cette fois l'intersection avec les éléments qu'aucun autre élément n'utilise.

4.4. Réduction du graphe

Si l'on ne s'intéresse qu'aux éléments nommables dans l'interface, on pourra réduire le graphe à ces éléments. Cela permet d'avoir plus rapidement des résultats sur ces éléments et de faciliter l'écriture de filtres.

La réduction du graphe se fait récursivement. Les éléments à réduire sont par exemple les valeurs (qui ne doivent plus contenir de sous-éléments), les modules non nommés (contructions `struct ... end` sans nom), etc., c'est-à-dire tout ce qui ne peut être exporté explicitement dans un fichier `.mli`.

Le principe est le suivant : pour chaque élément du graphe, on descend dans les éléments qu'il contient, que l'on tente de réduire à leur tour. Ensuite, s'il faut réduire l'élément en cours, on supprime chaque arc entre cet élément e et un autre élément f pour le remplacer par un arc de même type entre le conteneur de e et l'élément f.

4.5. Autres filtres utiles

Le langage de filtres permet d'effectuer dans le graphe d'autres recherches que celles évoquées ci-dessus. Par exemple, il peut être utile de rechercher les valeurs qui initialisent un certain champ `M.myrec.myfield` sans l'avoir lu. On utilisera ce filtre sur le graphe réduit :

$$\xrightarrow{create} < M.myrec.myfield > \ \& \ ! \xrightarrow{use,alias\{+\}} < M.myrec.myfield > \ \& \ < * >_v .$$

De même, il est parfois utile pour comprendre un code de voir ce qui lie deux fonctions, c'est-à-dire par l'intermédiaire de quels éléments une fonction `M1.f` utilise une fonction `M2.g` (bien sûr sans tenir compte des possibilités de passage de fonctions en paramètres, cf. la remarque en fin de section 3.2). Le filtre suivant (sur le graphe réduit) permet de sélectionner les éléments situés "entre" ces deux fonctions, et les deux fonctions elles-mêmes :

$$(< M1.f >_v \xrightarrow{use,alias\{+\}} \ \& \ \xrightarrow{use,alias\{+\}} < M2.g >_v) | < M1.f >_v \ | < M2.g >_v .$$

4.6. Utilisation de statistiques

Toujours dans l'optique de faciliter la compréhension d'un gros code existant, on peut imaginer quelques traitements statistiques sur ce graphe.

Nous pouvons par exemple chercher les éléments corrélés, en nous intéressant aux éléments référencés dans le même contexte, comme les valeurs utilisées dans les mêmes valeurs.

Pour ce faire, nous utilisons le graphe réduit, car il s'agit ici de comprendre les liens entre éléments au niveau de l'application en nous affranchissant du style. Si nous utilisions le graphe non réduit, les deux expressions

```
let x = let tmp = f x + 1 in let tmp2 = tmp + 3 in tmp2 * 4;;
let x = (f x + 4) * 4;;
```

ne donneraient pas le même résultat car dans le premier cas la fonction `f` serait utilisée par l'élément `tmp`, alors que dans le deuxième cas elle serait utilisée directement par `x`.

Pour déterminer le dégré de corrélation de deux éléments, nous utilisons le calcul de distance présent dans le test d'indépendance du χ^2 avec deux variables de Bernoulli (une par élément) prenant chacune deux valeurs possibles, "référencé" et "non référencé". Pour chaque élément e différent de e_1 et e_2, nous regardons si au moins un arc existe entre e et e_1 d'une part, et entre e et e_2 d'autre part. On ne s'intéresse pas aux arcs $\xrightarrow{contain}$, $\xrightarrow{include}$ou $\xrightarrow{export}$dans ce cas, car ils ne reflètent pas une utilisation. Selon l'existence des arcs, on incrémente la case correspondante dans le tableau suivant :

	e_1 référencé	e_1 non référencé	
e_2 référencé	$c_{0,0}$	$c_{1,0}$	$c_{*,0} = c_{0,0} + c_{1,0}$
e_2 non référencé	$c_{0,1}$	$c_{1,1}$	$c_{*,1} = c_{0,1} + c_{1,1}$
	$c_{0,*} = c_{0,0} + c_{0,1}$	$c_{1,*} = c_{1,0} + c_{1,1}$	$t = c_{*,0} + c_{*,1}$

Par exemple si un arc $e \rightarrow e_1$ existe mais qu'il n'y a pas d'arc $e \rightarrow e_2$, on incrémente $c_{0,1}$.

Ensuite, nous calculons la distance (par rapport à l'indépendance) du test du χ^2 sur cette matrice et conservons cette valeur :

$$v = \sum_{i=0,1} \sum_{j=0,1} \frac{(c_{i,j} - u_{i,j})^2}{u_{i,j}}$$

avec $u_{i,j} = \frac{c_{i,*}.c_{*,j}}{t}$.

Nous faisons ce calcul pour tous les éléments (ou éventuellement seulement ceux dont nous recherchons les éléments avec lesquels ils sont corrélés).

Pour un élément e, nous pouvons alors classer les éléments avec lesquels l'utilisation de e semble corrélée, selon les distances v calculées pour e et chacun des autres éléments.

Il est possible de restreindre les types d'arcs auxquels on s'intéresse. On peut ainsi découvrir des liens entre des champs de types de données en regardant seulement les arcs $\xrightarrow{create}$, de façon à mettre en évidence la corrélation des affectations à des champs différents.

Nous ne faisons pas un test du χ^2 mais utilisons seulement la formule de distance par rapport à l'indépendance pour classer, pour chaque élément e, les autres éléments par dégré de corrélation avec e. En effet, les conditions requises pour appliquer un test du χ^2 ne sont pas a priori réunies puisque, par exemple sur quatre éléments e_1, e_2, e_3, e_4, cela reviendrait à faire le test d'indépendance de e_1 et e_2 en considérant e_3 et e_4 indépendants, puis à considérer ultérieurement e_1 et e_2 indépendants pour tester l'indépendance de e_3 et e_4.

Cependant, même sans faire le test et en n'utilisant que la distance par rapport à l'indépendance, les résultats obtenus sur L montrent sans ambiguité des corrélations entre éléments dont les utilisations sont effectivement corrélées.

4.7. Contrôles automatiques

Une autre utilisation du graphe est le contrôle régulier du code source selon des contraintes définies par les développeurs.

On peut imaginer par exemple la situation suivante. Un module `M` définit un type `t`. On souhaite que les valeurs de ce type ne soient pas construites ou filtrées par d'autres modules, sauf un module de debogage `Debug`.

OCaml ne permet pas de déclarer une sorte de module "ami" de `M`, l'interface offerte par `M` étant la même pour tous les autres modules.

Une solution est donc de ne pas rendre le type `M.t` abstrait mais d'ajouter un contrôle pour s'assurer que les champs du type `M.t` ne sont lus et créés que dans les modules `M` et `Debug`.

Cela est fait facilement à l'aide du filtre suivant qui donne la liste des éléments accédant aux champs de `M.t` mais n'étant pas définis dans les modules `M` et `Debug` :

$$\xrightarrow{use,create} < M.t.* >_f \ \& \, !(< M.* > | < Debug.* >)$$

Par ailleurs, les analyses de recherche de code et de champs de types inutiles peuvent être relancées régulièrement pour détecter au plus tôt ces anomalies.

5. Implémentation et résultats

5.1. Implémentation

Cette analyse par graphe de dépendances est implémentée dans le logiciel libre Oug[1]. Il s'agit d'un outil en ligne de commande permettant l'analyse des fichiers `.ml` et `.mli` ainsi que le stockage et la lecture de graphes pour effectuer des analyses sur plusieurs parties du code sans reconstruire les graphes à chaque fois. Le langage de filtres est implémenté et permet à l'utilisateur dé spécifier, à l'aide d'une syntaxe adéquate, des filtres pour effectuer des recherches dans le graphe, comme indiqué en section 4. La recherche d'éléments inutiles est disponible directement par une option (ce n'est pas à l'utilisateur de construire et d'appliquer les filtres pour chaque élément), de même que l'analyse de corrélation. Enfin, diverses sorties au format Graphviz sont disponibles.

Une démonstration est accessible à l'adresse `http://yquem.inria.fr/~guesdon/demo.x`.

Nous donnons maintenant quelques détails d'implémentation.

Les éléments (modules, classes, valeurs, types, etc.) sont stockés dans un *tableau des éléments* et ont ainsi un identifiant unique qui est leur indice dans ce tableau. Même si tous les éléments ne peuvent pas forcément être nommés de façon unique, l'utilisation des identifiants évite toute ambiguité. Pour chaque élément, on conserve donc :

- son nom pleinement qualifié, pas forcément unique (par exemple, on peut avoir deux valeurs appelées `Module.f`),
- la localisation de sa définition, quand elle est disponible dans le *typedtree*, ceci afin de pouvoir l'afficher et indiquer sans ambiguité à l'utilisateur la position de l'élément en question,
- le type d'élément : module, classe, valeur, méthode, variable d'instance, type ou champ de type.

Le graphe contient deux tableaux, permettant d'associer à chaque élément la liste des arcs dont il est l'origine d'une part, dont il est la destination d'autre part. Ces listes sont des paires (identifiant d'élément, annotation de dépendance).

L'outil prend en paramètre une liste de fichiers sources OCaml. Ces fichiers doivent être donnés dans leur ordre de dépendance (ceux ne dépendant de rien en premiers), afin qu'un élément du graphe soit construit après les éléments dont il dépend, pour pouvoir créer les arcs de dépendance. Il est possible de construire incrémentalement le graphe en utilisant des options de stockage et de relecture d'un graphe et de l'environnement.

L'outil se lie avec le compilateur OCaml, afin d'en utiliser le *typedtree*. Cela permet d'avoir les "open" déjà résolus (sans avoir à rejouer la mécanique de recherche et d'ouverture de fichiers

[1] `http://pauillac.inria.fr/~guesdon/oug.en.html`

signatures) et les contraintes de types nécessaires à la gestion d'un environnement utilisé lors de l'analyse. En effet, les noms "résolus" dans le *typedtree* ne sont pas les noms pleinement qualifiés. Les sommets du graphe sont créés en parcourant les *typedtree* des différents modules à analyser, l'environnement permettant d'une part de résoudre les identifiants rencontrés (en gérant la portée lexicale) et d'autre part d'associer un nom pleinement qualifié de module, classe, valeur, etc., à l'identifiant du sommet créé dans le graphe. En particulier un élément est toujours créé avant les éléments qu'il contient (relation $\xrightarrow{contain}$) et a donc toujours un identifiant inférieur à ces derniers. Par ailleurs, l'utilisation du *typedtree* permet de garantir que le programme est correctement typé et qu'il n'y a pas d'identifiant non lié. Enfin, comme indiqué dans les perspectives, les indications de types pourraient servir pour des analyses ultérieures.

Lors de l'analyse d'un fichier `.mli`, il peut y avoir plusieurs éléments possibles correspondant à un élément présent dans ce fichier. Par exemple :

Fichier .mli	Fichier .ml
`val x : int`	`let x = 3 ; ;` `let x = 4 ; ;`

On considèrera, comme le compilateur, que c'est l'élément créé en dernier qui est exporté, c'est-à-dire celui ayant l'identifiant le plus élevé (ici le `x` de `let x = 4`).

5.2. Résultats

Sur le code du logiciel L, une première utilisation de Oug a permis de détecter et supprimer plusieurs dizaines d'éléments inutiles, incluant des champs d'enregistrements et des constructeurs. Le code est ainsi passé de 59.600 lignes à 57.300, soit une diminution de 3.85% de la taille du code. Certains éléments ont cependant été conservés en attendant de savoir si certaines fonctionnalités seraient à nouveau intégrées ou non. De plus, du code de debogage actuellement inutilisé n'a pas été supprimé non plus (commenter ou supprimer, telle est la question).

Lors des évolutions d'architecture prévues pour le logiciel, il est probable que Oug permettra de détecter et supprimer encore d'autres parties devenues inutiles.

6. Conclusion et perspectives

Nous avons défini la construction d'un graphe de dépendances pour du code OCaml et différentes façons d'exploiter ce graphe, notamment dans un contexte de reprise d'un logiciel existant. Cette analyse est implémentée dans un logiciel libre, Oug.

Les modules récursifs ne sont pas encore complètement traités par l'outil. La difficulté consiste à créer des sommets référençant des sommets inconnus au moment de l'analyse. Deux passes d'analyse seront nécessaires. Le problème ne se pose pas pour les classes car leurs éléments ne sont pas référencés depuis l'extérieur des classes (en l'absence d'analyse de flots de données).

Une piste pour enrichir le graphe serait la gestion des paramètres et l'ajout de liens entre ces paramètres et les autres éléments. A première vue, cela entraînerait la nécessité de créer un sommet dans le graphe pour chaque expression du programme analysé. On se trouverait alors au bord de l'analyse de flots de données.

Une autre piste de réflexion est l'ajout d'autres traitements statistiques pour faire ressortir des informations pertinentes pour la compréhension du programme à reprendre. Les analyses statistiques sont utiles sur des données incomplètes ou contenant des erreurs. Cela correspond pour notre cas à l'analyse d'un gros logiciel dont on cherche à comprendre l'organisation générale malgré un manque

d'homogénéité dans sa structure. Pour cela, l'ajout d'un type d'arc reflétant le type des valeurs dans le graphe serait sans doute utile.

Enfin, le graphe pourrait être exploité pour détecter des *code smells* [5], en définissant certains critères pour identifier automatiquement des parties douteuses du code.

Références

[1] Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman, *Compilateurs - Principes, techniques et outils, 2ème édition*, Pearson Education, 2007.

[2] Paul Anderson, Thomas Reps, Tim Teitelbaum, *Design and implementation of a fine-grained software inspection tool*, IEEE Transactions on software engineering, Volume 29, Issue 8 (Aug. 2003)

[3] Matthias Blume, *Dependency analysis for Standard ML*, ACM Transactions on Programming Languages and Systems (TOPLAS) Volume 21, Issue 4 (July 1999)

[4] Xavier Leroy et al., *The Objective Caml system release 3.11*, INRIA, 2008

[5] Eva van Emben, Leon Moonen, *Java quality assurance by detecting code smells*, Ninth Working Conference on Reverse Engineering, 2002.