Vérification par preuve formelle de propriétés fonctionnelles d'algorithme de classification

Léo Andrès

Vérification par preuve formelle de propriétés fonctionnelles d'algorithmes de classification

Rapport de travaux de recherche par

Léo Andrès

26 août 2019

Magistère 2 Informatique

Sous la direction de

Jean-Christophe Filliâtre

Résumé

Ce document est un rapport de mes travaux en tant qu'ingénieur d'étude sur le sujet *Vérification par preuve formelle de propriétés fonctionnelles d'algorithmes de classification*, effectué dans le cadre de ma formation en Magistère 2 d'Informatique à l'Université Paris-Sud. J'y présente la preuve formelle par vérification déductive de propriétés des algorithmes de Parcoursup' au moyen de Why3. J'ai été encadré par Jean-Christophe Filliâtre.

Table des matières

1. Donald KNUTH

1 Le LRI, l'équipe VALS, les méthodes formelles

Le LRI [2] est un laboratoire de l'Université Paris-Sud. C'est une UMR [3] du CNRS. Le LRI est composé de plusieurs équipes dont notamment l'équipe VALS [4], qui mène de nombreux travaux dans le domaine des méthodes formelles.

1.1 « Software is hard. » [5]

Du fait de leur complexité, les programmes sont rarement exempts de bugs. Une question majeure et récurrente en informatique est de trouver des moyens permettant de prévenir l'introduction de bugs et de garantir leur absence. Pour cela, plusieurs méthodes sont possibles.

1.2 Approche *a priori*

Une solution partielle à ce problème a par exemple été de chercher à concevoir des langages de programmation qui permettent d'éviter des classes d'erreurs courantes. C'est par exemple ce qui a été fait dans des langages comme OCaml avec le typage statique. Cependant, cette approche ne permet de vérifier que des propriétés génériques (e.g. *le programme est bien typé*) et non pas spécifiques à notre programme (e.g. *le programme calcule les mille premiers nombres premiers*). En effet, dans une telle approche, on ne connaît pas le programme à l'avance et encore moins les propriétés spécifiques qu'il est censé vérifier. Or, prouver une propriété qui n'a pas même été énoncée est impossible.

1.3 Les tests

1.3.1 Les tests manuels

On peut bien évidemment tester notre programme à la main, en l'exécutant et vérifiant qu'il se comporte comme prévu. Cette approche comporte de nombreux inconvénients : dans certains cas il est impossible ou extrêmement coûteux de tester le programme en conditions réelles, une intervention humaine est nécessaire à chaque fois que l'on veut vérifier le programme. Il n'est pas non plus garanti que l'on aura testé l'ensemble des cas possibles d'utilisation du programme. Enfin, si l'on détecte un problème, il n'est pas forcément aisé de retrouver l'origine de celui-ci. De plus, se comporter *comme prévu* est plutôt vague et sujet à l'interprétation.

2. Laboratoire de Recherche en Informatique.
3. Unité Mixte de Recherche.
4. Vérification, Algorithmes, Langages et Systèmes.
5. Donald KNUTH

1.3.2 Les tests unitaires

Pour pallier à cela, il est possible de réaliser des *tests unitaires* : pour chaque fonction de notre programme, on écrit à la main un ensemble de tests automatisés, qui appellent la fonction sur une entrée donnée et comparent le résultat avec le résultat attendu. Cette façon de faire permet de résoudre la majorité des problèmes liés aux tests manuels, à l'exception de l'exhaustivité. Une fonction peut potentiellement avoir un nombre très important ou infini d'entrées valides : il n'est alors pas envisageable ou possible d'écrire un test pour chacune.

1.4 La vérification

Si l'on veut vérifier une propriété de manière exhaustive, et donc, sur une infinité de cas, il ne nous reste plus qu'une possibilité : la démonstration mathématique. On pourrait faire une preuve sur le papier, mais bien que pouvant fournir une bonne intuition quant à la raison pour laquelle le programme est correct ou sur la marche à suivre pour la preuve, il est possible et même courant de se tromper en procédant ainsi. On préfère pour cela, et pour d'autres raisons, une preuve mécanisée.

Il existe plusieurs méthodes permettant de vérifier une propriété d'un programme. On peut notamment citer le *model checking*, l'*interprétation abstraite* et la *vérification déductive*.

1.4.1 La vérification déductive

« Deductive program verification is the art of turning the correctness of a program into a mathematical statement and then proving it. »[Fil11]

Étant donnée une spécification d'un programme, il faut donc commencer par la transcrire. Cela nécessite d'avoir une logique permettant d'énoncer de façon formelle les propriétés que l'on souhaite vérifier.

L'énoncé permet alors de générer des VC [6], qu'il faut prouver. Pour cela, deux méthodes sont possibles :

- une preuve interactive, au moyen d'assistants de preuve tels que Coq, HOL ou Isabelle ;
- une preuve automatique, où l'on délègue à des prouveurs automatiques de théorèmes tels qu'Alt-Ergo, Z3 ou CVC4.

La vérification déductive peut mener à la génération d'un important nombre de VC. On préfère éviter d'avoir à toutes les traiter interactivement et donc, on

6. Verification Conditions, ou conditions de vérification en français.

essaie autant que faire se peut de passer par une preuve automatique. Cela présente plusieurs avantages : l'effort à fournir est réduit et l'on peut plus facilement modifier le programme ou bien l'énoncé que l'on cherche à vérifier.

Dans certains cas, les prouveurs échouent. Il faut alors tenter de les aider en ajoutant des assertions ou des invariants, ou, en dernier recours, passer à une preuve interactive.

2 Parcoursup

2.1 Généralités

Parcoursup est une plateforme gouvernementale française utilisée par les étudiants pour candidater à des formations de l'enseignement supérieur. Le principe général est le suivant :

- les étudiants candidatent à plusieurs formations ;
- chaque formation classe l'ensemble des étudiants selon les critères de son choix [7] ;
- la plateforme Parcoursup apporte des modifications aux classements pédagogiques [8], selon des critères et des algorithmes publics et notifie les candidats acceptés, qui doivent alors faire un choix parmi les formations où ils sont acceptés [9] ;
- lorsqu'un candidat choisit une formation parmi celles où il a été accepté, cela libère des places dans les autres, qui sont alors proposées aux candidats suivants sur l'ordre d'appel.

Plus de détails sont donnés dans [Gim19].

2.2 Algorithme avec taux de boursiers

Pour passer du classement pédagogique à l'ordre d'appel, il existe plusieurs cas, selon les formations. Chaque cas correspond à un algorithme particulier. Le cas le plus simple est celui avec un taux de boursiers.

Pour chaque formation sélective, le recteur de l'académie fixe un taux minimum de boursiers à qui l'on doit proposer une place dans la formation.

Initialement, l'ordre d'appel est vide. On dit que le taux est contraignant s'il reste au moins un boursier qui n'a pas été ajouté à l'ordre d'appel et si le fait

7. On parlera de *classement pédagogique*.
8. Le nouveau classement est appelé l'*ordre d'appel*.
9. Tout en gardant la possibilité de choisir une autre formation où ils sont sur liste d'attente pour le moment.

d'ajouter un candidat non-boursier à l'ordre d'appel rendrait le taux de boursiers dans l'ordre d'appel inférieur au taux minimum.

Pour remplir l'ordre d'appel, on procède comme suit :

— si le taux n'est pas contraignant, alors, on prend le meilleur candidat, qu'il soit boursier ou non ;

— si le taux est contraignant, alors, on prend le meilleur candidat boursier.

2.3 Propriétés de l'algorithme avec taux de boursiers

On souhaite vérifier cinq propriétés de l'algorithme. Elles ont été publiées avec la description des différents algorithmes[Gim19]. On a introduit une propriété P_0, qui est une propriété implicite du document.

P_0 L'ordre d'appel est une permutation du classement pédagogique. Ce qui signifie que l'on n'a pas ajouté ou perdu de candidat en modifiant le classement.

P_1 Pour tout préfixe de l'ordre d'appel, soit le taux de boursiers est respecté, soit il n'y a plus aucun boursier après. Ce qui signifie que l'on a toujours respecté le taux si c'était possible.

P_2 Un candidat boursier avec le rang r dans le classement pédagogique aura au moins le rang r dans l'ordre d'appel. Ce qui signifie qu'un boursier n'est jamais doublé par personne, et donc, que son rang ne peut que devenir meilleur ou rester le même.

P_3 Un candidat non-boursier avec le rang r dans le classement pédagogique aura au mieux le rang r dans l'ordre d'appel et au pire le rang $r \times (1 + q_b/(100 - q_b)$ avec q_b le taux de boursiers. Ce qui signifie qu'un candidat non-boursier ne double jamais personne, et donc, que son rang ne peut que rester le même ou être moins bon ; cela signifie aussi qu'il ne peut pas perdre plus de places que ce qu'impose le taux minimum de boursiers.

P_4 Parmi toutes les permutations possibles du classement pédagogique qui respectent P_1, l'ordre d'appel est celle qui minimise le nombre d'inversions — distance de Kendall-tau. Cela signifie que pour faire respecter le taux de boursiers, on a modifié le classement pédagogique le moins possible.

P_5 L'ordre d'appel est le minimum selon l'ordre lexicographique induit par les classements parmi toutes les permutations possibles du classement pédagogique qui respectent P_1. Cela signifie que pour faire respecter le taux de boursiers, on a favorisé les meilleurs candidats autant que faire se peut.

3 Preuves et implémentation de référence

On a utilisé Why3, un outil pour la vérification déductive de programmes [FP13]. L'algorithme avec taux de boursiers a été implémenté en WhyML et c'est sur cette implémentation qu'ont été réalisées les preuves.

3.1 Propriétés génériques

Avant toute chose, on souhaite vérifier un ensemble de propriétés que tout programme se doit de respecter.

3.1.1 Typage

Notre programme, tout comme sa spécification, est bien typé, sans quoi il serait rejeté par le *type checker* de Why3. L'intérêt du typage est présenté dans [Mil78]. En comparaison d'un langage comme OCaml, Why3 impose quelques restrictions supplémentaires comme le contrôle statique des alias — voir [FGP16a] —, mais cet algorithme ne s'y heurte pas, contrairement au second algorithme avec taux de boursiers et de résidents.

3.1.2 Terminaison

On dit qu'un programme termine sur une entrée s'il finit par s'arrêter et qu'il ne tourne pas indéfiniment. On souhaite montrer la terminaison pour l'ensemble des entrées du programme. Seules quelques constructions du langage peuvent produire un programme qui ne termine pas : boucles `while` et fonctions récursives. Lorsque l'on utilise une telle construction, il suffit généralement d'exhiber un terme dont la valeur décroit et de montrer que l'on va finir par s'arrêter. Par exemple, dans la boucle `while` qui ajoute un candidat à l'ordre d'appel, on répète le corps de la boucle tant que la taille de l'ordre d'appel est strictement inférieure au nombre initial de candidats dans le classement pédagogique. Il suffit de fournir le *variant* :

```
nb initial de candidats − taille de l'ordre d'appel
```

pour prouver la terminaison de la boucle. On a montré que l'ensemble du programme termine.

3.1.3 Sûreté

Une autre propriété que l'on souhaite montrer est la sûreté. On veut par exemple montrer que lorsque l'on accède au $i^{\text{ème}}$ élément d'un tableau, $0 \leq i < n$

avec n la taille du tableau. Un tel accès ayant généralement lieu au sein d'une boucle, il suffit de maintenir un invariant sur l'indice auquel on accède et de le prouver. On a montré la sûreté de l'ensemble du programme.

3.2 Preuves des propriétés

Les propriétés P_0, P_1, P_2 et P_5 on été prouvées. Pour P_3, seule la partie indiquant qu'un candidat de rang r dans le classement pédagogique aura au mieux le rang r dans l'ordre d'appel a été prouvé. P_4 et la seconde partie de P_3 ont été énoncées mais ne sont pas prouvées.

Les diverses preuves menées ont impliqué des techniques variées et intéressantes comme utilisation de code *ghost* — voir [FGP16b] — et de *lemma functions*, ou encore de méthodes évitant d'avoir à prouver l'absence d'*overflow* — voir [CFP] — etc. On ne détaillera pas les preuves ici par manque de place, mais quelques points sont intéressants à souligner.

Tout d'abord, on avait au départ décidé de maintenir divers invariants à l'intérieur même des fonctions de l'algorithme. Cependant, en procédant ainsi, on créait de la redondance : un même invariant pouvait demander d'être ajouté en précondition et en postcondition de nombreuses fonctions, ce qui était fastidieux au vu du nombre d'invariants. On a alors décidé de créer un type contenant la majorité des variables utilisées par l'algorithme et de donner des invariants à ce type directement. Ainsi, on doit toujours prouver leur préservation, mais sans avoir à les spécifier plusieurs fois. La définition du type et de quelques-uns des invariants est donnée en annexe A.1.

D'autre part, pour faciliter les preuves, plusieurs modifications ont été apportées à la bibliothèque standard de Why3. On peut notamment mentionner la réécriture des modules de file et de permutation, ainsi que l'ajout de divers lemmes, sur les séquences notamment.

3.3 Production de code exécutable

Il est possible de générer du code dans un autre langage à partir de l'implémentation WhyML, on appelle cela l'*extraction*, ce mécanisme est brièvement décrit dans [Fil+18]. On a extrait notre code vers OCaml. En compilant le code OCaml extrait, on obtient ainsi une implémentation de référence, que l'on peut exécuter avec des données anonymisées et comparer ainsi les résultats obtenus.

4 Perspectives

4.1 Autres algorithmes

Seul le premier algorithme a été traité, il est celui qui couvre le plus de cas parmi les formations. Le deuxième, avec taux de boursiers et taux de résidents, est en réalité très similaire, il a été partiellement implémenté et diverses preuves ont été débutées. Il est à noter que les preuves du premier algorithme ont été réalisées de telle sorte qu'elles devraient facilement s'adapter au deuxième algorithme. Le troisième algorithme, qui gère les internats, est différent et bien plus compliqué, mais il ne concerne qu'un petit nombre de formations.

4.2 Implémentation originale

On souhaite pouvoir vérifier l'implémentation originale en Java. Pour cela, un outils appelé `jml2why3` est développé par Benedikt Becker. Il devrait permettre de produire du code WhyML à partir du code Java original. On espère alors pouvoir transposer les preuves de notre implémentation WhyML au code WhyML généré.

5 Bilan, remerciements

Plusieurs cours suivis durant mon cursus ont été utiles lors de ce stage. Notamment le cours de logique (L3), de programmation fonctionnelle (L2/L3), d'informatique théorique (L3), de génie logiciel (L3) et de compilation (M1). Le fait d'avoir suivi un cursus renforcé en mathématiques en L2 et d'avoir déjà réalisé des stages dans le domaine de la preuve m'a aussi facilité la tâche. Cependant, ce stage a aussi été l'occasion d'acquérir de nouvelles connaissances, autour bien évidemment de la vérification déductive, de Why3, des solveurs SMT etc.

Ce stage a été effectué à la suite de travaux encadrés de recherche lors du second semestre, sur un sujet différent mais toujours sous la direction de Jean-Christophe. Je ne peux que lui réitérer mes remerciements pour le temps qu'il m'a consacré et toutes les choses si variées qu'il a pu me transmettre. Je tiens aussi à remercier les différents membres de l'équipe vals, j'ai pris un grand plaisir à passer du temps en leur compagnie, à la fois agréable et stimulante sur le plan scientifique. Merci à tous et particulièrement, merci Jean-Christophe.

A Annexes

A.1 Type et exemple d'invariant

```
type ordre_appel_valide = {
  ordre_appel: Q.t voeu;
  boursiers: Q.t voeu;
  non_boursiers: Q.t voeu;
  taux_b: int;
  nb_voeux: int;
  nb_b_total: int;
  mutable nb_b_appeles: int;
} invariant { 0 <= taux_b <= 100 }
  invariant { nb_voeux >= 0 }
  invariant { boursiers_seulement boursiers }
  invariant { sorted boursiers }
  invariant { length boursiers = 0 <-> nombre_boursiers
   ↪  ordre_appel = nb_b_total }
  invariant { forall i. 0 <= i <= length ordre_appel ->
    (nombre_boursiers ordre_appel[..i] = nb_b_total -> length
     ↪  boursiers = 0) }
  invariant { forall i. 0 <= i <= length ordre_appel ->
    let nb_b_local = nombre_boursiers ordre_appel[..i] in
    taux_ok taux_b nb_b_local i \/ nb_b_local = nb_b_total }
  invariant { forall i. 0 <= i < length ordre_appel ->
    let candidat = ordre_appel[i] in
    est_boursier candidat ->
    boursiers <> empty ->
    voeu_lt candidat boursiers[0]
  }
  invariant { forall i1 i2. 0 <= i1 < i2 < length ordre_appel
   ↪  ->
    let c1 = ordre_appel[i1] in
    let c2 = ordre_appel[i2] in
    (est_boursier c1 && (not est_boursier c2)) ->
    let nb_b_local = nombre_boursiers ordre_appel[..i1] in
    voeu_lt c1 c2 \/ not (taux_ok taux_b nb_b_local (i1 + 1))
  }
```

A.2 Séminaires

Durant ce stage, j'ai eu l'occasion d'assister à plusieurs séminaires. On en détaille quelques-uns ici.

A.2.1 Toward a Coq verified SQL's compiler

Ce séminaire a été donné par Véronique Benzaken et présentait un travail également réalisé par Évelyne Contejean et Chantal Keller. Tout d'abord, il faut préciser que c'est dans le cadre de ce projet que j'avais effectué un stage en 2017.

SQL est un langage déclaratif, c'est-à-dire que l'on écrit ce que l'on veut obtenir, mais pas comment l'obtenir. Le compilateur doit donc générer un *plan d'exécution*, qui lui, contient les algorithmes qui seront utilisés pour obtenir le résultat demandé. De plus, les bases de données stockant un important volume de données, elles sont forcées d'utiliser le disque, ralentissant énormément leur exécution. Ainsi, les optimisations sont omniprésentes dans tout compilateur SQL.

Pour ces raisons, la vérification d'un compilateur SQL est difficile. Ce séminaire présentait le travail effectué pour permettre d'arriver à un compilateur vérifié en Coq. Notamment sur la définition d'une sémantique, le lien avec l'algèbre relationnelle et les implémentations en Coq.

A.2.2 When Everyday Combinatorics Meets Formal Verification

Ce séminaire a été donné par Alain Giorgetti. Il y décrivait son travail sur diverses structures utilisées en combinatoire et son usage des méthodes formelles. Particulièrement, le fait qu'au départ il ne travaillait pas dans le domaines des méthodes formelles mais qu'avec le temps, il a été convaincu par leur importance. Notamment parce que la formalisation de théorèmes peut mener à une meilleure compréhension du domaine étudié. Plusieurs exemples ont été donnés dans le cadre d'une formalisation avec Why3.

A.2.3 Scallina : on the Intersection of Scala and Gallina

Ce séminaire a été donné par Youssef El Bakouny. Il y décrivait l'extraction de programmes Coq (dans le langage Gallina donc) vers Scala. Du fait de son système de type, Scala lève des difficultés supplémentaires par rapport à OCaml ou Haskell pour lesquels l'extraction est déjà possible. En effet, Scala permet de typer des expressions qui ne sont pas typables en OCaml ou Haskell. De plus, lors de l'extraction, on souhaite généralement obtenir un programme lisible. Une implémentation (Scallina) et les méthodes employées pour résoudres ces différences difficultés ont été détaillés durant le séminaire.

A.2.4 Ghost Code in Action : Automated Verification of a Symbolic Interpreter using Why3

Ce séminaire a été donné par Benedikt Becker. Il y présentait la réalisation en Why3 d'un interpréteur symbolique pour les scripts POSIX Shell. Le but du projet étant la vérification des scripts d'installation de paquets Debian. La différence entre exécution symbolique et exécution concrète a été rappelée pour ensuite montrer comment il est possible de prouver la correction de l'interpréteur par deux théorèmes. À savoir : l'ensemble des exécutions concrètes est représenté par l'exécution symbolique ; aucun état inutile n'est généré dans l'exécution symbolique. Cela a été réalisé en ayant comme objectif une vérification automatique avec Why3, menant à des techniques de preuve originales, dont un important usage de code *ghost*.

Références

[Fil11] Jean-Christophe Filliâtre. « Deductive Program Verification ». Thèse d'habilitation. Université Paris-Sud, déc. 2011. URL : http://www.lri.fr/~filliatr/hdr/memoire.pdf (cf. p. 3).

[Gim19] Hugo Gimbert. *Document de présentation des algorithmes de Parcoursup*. 2019. URL : https://framagit.org/parcoursup/algorithmes-de-parcoursup/blob/master/doc/presentation_algorithmes_parcoursup_2019.pdf (cf. p. 4, 5).

[FP13] Jean-Christophe Filliâtre et Andrei Paskevich. « Why3 — Where Programs Meet Provers ». In : *Proceedings of the 22nd European Symposium on Programming*. Sous la dir. de Matthias Felleisen et Philippa Gardner. T. 7792. Lecture Notes in Computer Science. Springer, mar. 2013, p. 125-128 (cf. p. 6).

[Mil78] Robin Milner. « A theory of type polymorphism in programming ». In : *Journal of Computer and System Sciences* 17 (1978), p. 348-375 (cf. p. 6).

[FGP16a] Jean-Christophe Filliâtre, Léon Gondelman et Andrei Paskevich. *A Pragmatic Type System for Deductive Verification*. Research Report. https://hal.archives-ouvertes.fr/hal-01256434v3. Université Paris Sud, 2016 (cf. p. 6).

[FGP16b] Jean-Christophe Filliâtre, Léon Gondelman et Andrei Paskevich. « The Spirit of Ghost Code ». In : *Formal Methods in System Design* 48.3 (2016), p. 152-174. ISSN : 1572-8102. DOI : 10.1007/s10703-016-0243-x (cf. p. 7).

[CFP] Martin Clochard, Jean-Christophe Filliâtre et Andrei Paskevich. « How to avoid proving the absence of integer overflows ». In : p. 94-109 (cf. p. 7).

[Fil+18] Jean-Christophe Filliâtre et al. *A Toolchain to Produce Correct-by-Construction OCaml Programs*. Rapp. tech. artifact : https://www.lri.fr/~mpereira/correct_ocaml.ova. 2018 (cf. p. 7).