Vers un outil de vérification formelle légere pour OCaml

Thomas Genet, Barbara Kordy, Amaury Vansyngel

Vers un outil de vérification formelle légère pour OCaml

Thomas Genet[1], Barbara Kordy[2], and Amaury Vansyngel[1]

[1]IRISA, Université de Rennes 1, France
[2]IRISA, INSA de Rennes, France

Résumé

Si l'on décrit, par une grammaire, l'ensemble des entrées possibles d'un programme fonctionnel, peut-on connaître la grammaire des sorties de celui-ci ? Il existe des outils en réécriture à même de répondre à cette question, pour certaines fonctions. On peut utiliser ce genre de calcul pour détecter des bugs ou, à l'inverse, pour prouver des propriétés sur ces fonctions. Dans cet article, nous présentons un travail en cours visant à concevoir un outil de vérification formelle légère pour OCaml. Si l'essentiel des résultats théoriques et outils de réécriture existent déjà, leur application à la vérification de programmes OCaml réalistes nécessite de résoudre un certains nombre de problèmes. Nous donnerons l'architecture d'un interprète abstrait pour OCaml, basés sur ces principes et outils, et nous verrons quelles sont les briques manquantes pour finaliser son développement.

1 Une vérification aussi automatique que l'inférence de type

Certains langages de programmation fortement typés (comme Haskell, OCaml, Scala et F#) disposent d'un mécanisme d'inférence de type. Ce mécanisme permet, entre autres, de détecter automatiquement certains types d'erreurs dans les programmes. Pour vérifier plus que le bon typage d'un programme, il est possible de définir des propriétés et, ensuite, de les prouver à l'aide d'un assistant de preuve ou d'un prouveur automatique. Cependant, la formulation de ces propriétés en logique et la construction de la preuve requièrent généralement une certaine expertise.

On s'intéresse ici à une famille restreinte de propriétés : des propriétés "régulières" portant sur les structures manipulées par ces programmes. On décrit, par une grammaire (d'arbres, régulière), l'ensemble de structures données en entrée à une fonction et on cherche à construire la grammaire des structures pouvant être obtenues en sortie (ou une approximation). La grammaire de sortie pourra révéler un bug potentiel dans le programme ou, à l'inverse, montrer que celui-ci vérifie une propriété régulière.

La famille de propriétés démontrables de cette façon est restreinte, même si elle dépasse ce que l'on peut exprimer par typage simple. On peut par exemple distinguer la liste vide de la liste non vide. Une telle distinction est également faisable avec des GADTs (Generalized Algebraic DataTypes) mais, à notre connaissance, pas le type de raisonnement sur des structures récursives que l'on propose ici. Il existe d'autres approches où le système de type est enrichi par des formules logiques et de l'arithmétique [17, 3]. Cependant ces techniques nécessitent généralement d'annoter le programme pour que la vérification de type réussisse. Les propriétés que nous considérons ici sont volontairement plus simples pour éviter au maximum le travail d'annotation. Pour ces propriétés, l'objectif est de proposer une forme de vérification formelle "légère". La vérification est formelle car elle *démontre* que les résultats d'une fonction ont une certaine forme. La vérification est légère car, d'une part, la preuve est automatique, et d'autre part, il n'est pas nécessaire de définir la propriété attendue par une formule logique mais simplement d'observer le résultat d'un calcul abstrait.

Concernant le procédé de calcul lui-même (produire la grammaire de sortie à partir de la grammaire d'entrée), un certain nombre de travaux abordent ce sujet dans la communauté de programmation fonctionnelle [13, 15] comme dans la communauté de réécriture [10, 7]. Dans la communauté de programmation fonctionnelle, l'analyse du flot de contrôle des programmes fonctionnels d'ordre supérieur est une thématique de recherche extrêmement active [2, 14] mais l'objectif est différent puisqu'il ne vise pas à déterminer l'ensemble des résultats. Dans [10, 7], à partir d'un système de réécriture de termes codant une fonction et d'un automate (ou d'une grammaire) reconnaissant les entrées de la fonction, il est possible de produire automatiquement un automate reconnaissant une *sur-approximation* des sorties de cette fonction. On dispose donc d'un outil pouvant interpréter ou évaluer, de façon *abstraite*, une fonction (exprimée par un système de réécriture) non pas sur une entrée, comme le fait un interprète classique, mais sur un ensemble d'entrées. Voyons maintenant ce qui manque pour réaliser, à partir de ces outils de réécriture, un outil de vérification pour OCaml.

2 A quoi pourrait ressembler un interprète abstrait OCaml ?

Imaginons que l'on dispose d'une notation, inspirée des expressions régulières, pour définir des langages réguliers de listes. Notons `[a*]` le langage des listes avec 0 occurrence ou plus du symbole `a`. Notons `[a*,b*]` le langage des listes avec 0 occurrence ou plus de `a`, suivies de 0 occurrence ou plus de `b`. On note `[(a|b)*]` toute liste avec 0 occurrence ou plus de `a` ou `b` (dans n'importe quel ordre). On souhaite définir, par exemple, une fonction de suppression supprimant toutes les occurrences d'un élément dans une liste en OCaml. Voici une première définition (erronée) de cette fonction :

```
let rec delete x l= match l with
  | [] -> []
  | h::t -> if h=x then t else h::(delete x t);;
```

Il est bien sûr possible de réaliser des tests sur cette fonction en utilisant l'interprète OCaml :

```
# delete 2 [1;2;3];;
-:int list= [1; 3]
```

Si l'on disposait d'un interprète *abstrait* OCaml travaillant sur des grammaires, nous pourrions lui poser la question suivante : quel est le langage des listes obtenu en appliquant `delete` à `a` et n'importe quelle liste de `a` et de `b` ?

```
# delete a [(a|b)*];;
-:abst list= [(a|b)*];;
```

La réponse obtenue n'est pas celle attendue. Dans la mesure où toutes les occurrences de `a` auraient dû être supprimées, nous espérions le résultat `[b*]`. Comme l'interprète abstrait donne une sur-approximation de la grammaire de sortie, cela ne démontre pas l'existence d'un bug. Cela met quand même la puce à l'oreille : toutes les occurrences de `a` ne seraient-elles pas supprimées ? Effectivement, il devrait y avoir un appel récursif de `delete` dans la branche `then`. Prenons maintenant cette deuxième version de la fonction :

```
let rec delete x l= match l with
  | [] -> []
  | h::t -> if x=x then (delete x t) else h::(delete x t);;
```

Si on pose à l'interprète la même question, on obtient :

```
# delete a [(a|b)*];;
-:abst list= [];;
```

Cette fois-ci on a la garantie de la présence d'un bug, car la sur-approximation ne contient même pas le langage attendu : `[b*]`. Si on corrige cette dernière erreur (`if x=x...`), on obtient la fonction suivante :

```
let rec delete x l= match l with
  | [] -> []
  | h::t -> if h=x then (delete x t) else h::(delete x t);;
```

Sur cette fonction l'interprète abstrait nous donnera finalement :

```
# delete a [(a|b)*];;
-:abst list= [b*];;
```

Ce résultat prouve deux propriétés : (1) `delete` supprime toutes les occurrences d'un élément dans la liste et (2) `delete` ne supprime *pas* toutes les occurrences des autres éléments dans la liste. Ce ne sont que *quelques* propriétés de `delete` mais celles-ci ont pu être démontrées sans avoir dû, au préalable, les formaliser en logique. De plus, la preuve est automatique. Comme autre exemple, prenons les fonctions suivantes définissant un tri par insertion :

```
let rec insertion e l= match l with
    [] -> [e]
  | h::t -> if e=h then e::h::t else h::(insertion e t)

let rec sort l= match l with
    [] -> []
  | h::t -> insertion h (sort t)
```

Comme pour la fonction `delete`, un test un peu rapide de la fonction avec l'interprète OCaml peut nous donner l'illusion que cette fonction est correcte :

```
# sort [3;2;1];;
-:int list= [1;2;3];;
```

Un interprète OCaml abstrait nous permettrait, lui, de détecter rapidement un problème : la liste en sortie n'est pas nécessairement triée (on suppose que `a` est inférieur ou égal à `b`).

```
# sort [(a|b)*];;
-:abst list= [(a|b)*];;
```

Si l'on corrige l'erreur dans la condition du `if` de la fonction `insertion`, on obtient la fonction :

```
let rec insertion e l= match l with
    [] -> [e]
  | h::t -> if e<=h then e::h::t else h::(insertion e t)

let rec sort l= match l with
    [] -> []
  | h::t -> insertion h (sort t)
```

Avec cette fonction, un interprète abstrait nous donnera un résultat plus proche de nos attentes, comme `[a*,b*]`. Sur ces exemples, la sur-approximation est suffisamment fine pour que l'interprète abstrait révèle une erreur ou, au contraire, prouve la propriété. Cependant, si l'approximation est trop grossière, le langage obtenu ne fournit que peu d'information. Par exemple, si nous analysons de la même façon la fonction de tri par fusion nous obtenons le langage `[(a|b)*]` qui ne garantit pas la propriété attendue. Cependant, nous verrons dans la Section 4 qu'il existe des pistes pour raffiner automatiquement une approximation lorsqu'elle est trop grossière.

3 Construire un interprète abstrait pour OCaml

Comme nous avons vu dans la première section, il existe des travaux qui visent le même objectif que l'interprète abstrait mais pour des systèmes de réécriture. En utilisant Timbuk [8], il est possible d'effectuer les preuves présentées ci-dessus, en traduisant les fonctions OCaml en systèmes de réécriture. Par exemple, la fonction `delete` (corrigée), peut être traduite manuellement dans le système de réécriture $\mathcal{R}$ suivant :

$$
\begin{array}{l}
delete(X, nil) \to nil \\
delete(X, cons(Y, Z)) \to ite(eq(X, Y), delete(X, Z), cons(Y, delete(X, Z))) \\
ite(true, X, Y) \to X \quad ite(false, X, Y) \to Y \\
eq(a, a) \to true \quad eq(a, b) \to false \quad eq(b, a) \to false \quad eq(b, b) \to true
\end{array}
$$

où les constantes a et b, le prédicat d'égalité eq et ite (pour if then else) sont générés en complément de la définition de $delete$ [1]. D'autre part, l'expression `delete a [(a|b)*]` peut être traduite manuellement en un automate d'arbre $\mathcal{A}$ (et non une grammaire) reconnaissant le langage régulier des termes correspondant à cette requête. Enfin, avec Timbuk il est possible de produire automatiquement un autre automate d'arbre, $\mathcal{A}^*$, reconnaissant une sur-approximation de l'ensemble des termes obtenus par réécriture des termes reconnus par $\mathcal{A}$. Si l'on s'intéresse au sous-langage de $\mathcal{A}^*$ représentant des valeurs [2], celui-ci correspond effectivement à `[b*]`. Actuellement, nous complétons un interprète OCaml avec ces fonctionnalités abstraites. Le principe est résumé par la Figure 1.

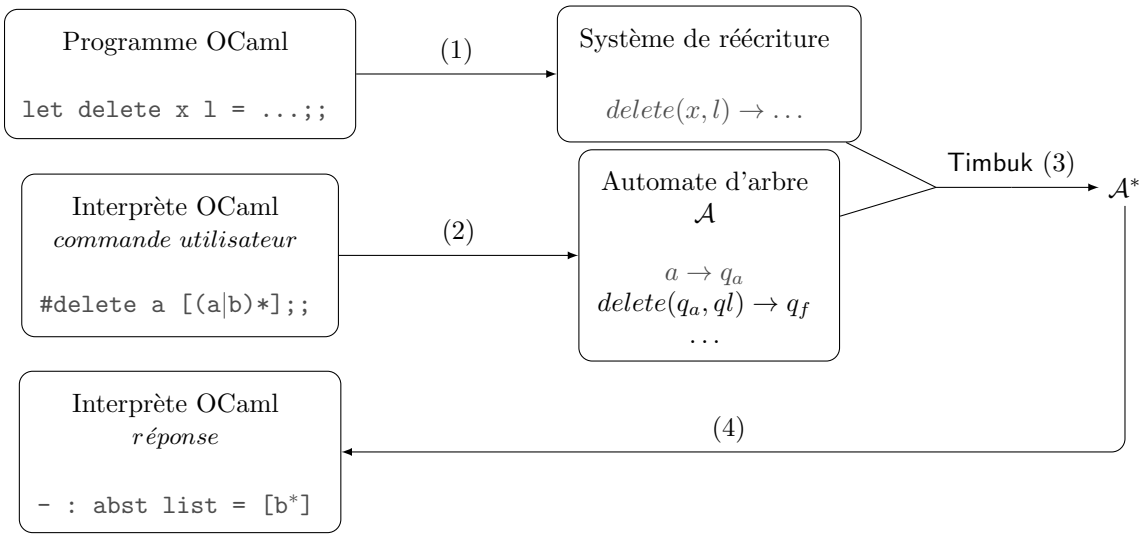

FIGURE 1 – Utilisation de Timbuk pour la réalisation de l'interprète abstrait

(1) La définition d'une fonction OCaml dans l'interprète est complétée par la génération d'un système de réécriture ;

(2) Les requêtes abstraites faites dans l'interprète sont interceptées pour être évaluées séparément. On génère automatiquement un automate d'arbre reconnaissant le langage décrit par l'expression régulière ;

(3) Dans le cas d'une requête classique, on laisse l'interprète OCaml standard faire son travail. Dans le cas d'une requête abstraite, celle-ci est redirigée vers Timbuk pour calculer $\mathcal{A}^*$;

(4) L'automate résultat $\mathcal{A}^*$ est réinterprété sous la forme d'une expression intelligible par l'utilisateur.

Le sous-ensemble du langage OCaml auquel nous nous intéressons actuellement est le suivant : programmes fonctionnels purs (pas d'effets de bord, pas de références, pas de structures mutables, pas d'objets), avec définition de fonctions mutuellement récursives, d'ordre supérieur, avec pattern-matching, sans exceptions, sans types prédéfinis (`int`, `string`, etc.) mais avec des types algébriques. Pour disposer d'un interprète abstrait, sur ce sous-ensemble d'OCaml, il reste quelques obstacles à surmonter. Le premier est de définir formellement une traduction de OCaml vers la réécriture utilisée par (1). Cette traduction doit préserver la sémantique du programme OCaml de départ. D'autres travaux étudient ce type de transformation [16, 4] pour des langages plus riches que ce que nous visons pour l'instant. Par exemple, ces traductions couvrent les types

1. A la place du codage avec ite, on aurait pu utiliser des règles de réécriture conditionnelle mais celles-ci ne sont que partiellement supportées par Timbuk.

2. L'automate $\mathcal{A}^*$ reconnaît tous les calculs intermédiaires jusqu'aux résultats. On peut ne garder que les résultats en calculant un produit entre $\mathcal{A}^*$ et un automate reconnaissant uniquement les valeurs résultats.

prédéfinis, les effets de bords et même la concurrence. En revanche, la traduction attendue en (1) devra produire un système de réécriture suffisamment complet à partir d'un programme OCaml dont le pattern-matching est exhaustif. En effet, la complétude suffisante est nécessaire pour garantir la terminaison de l'interprète abstrait [7]. D'autre part, cette traduction devra prendre en compte les fonctions d'ordre supérieur. Les expérimentations menées avec un codage des fonctions d'ordre supérieur en réécriture, proposé par Jones et Andersen dans [13], sont encourageantes. Si les résultats obtenus par Jones et Andersen ont été dépassés depuis (par [15]), ce n'est pas la faute du codage mais plutôt de la précision de leur algorithme de calcul de grammaires. Notre calcul étant plus précis, nous avons constaté qu'il était possible de traiter les fonctions d'ordre supérieur considérées dans [15] avec ce codage [11]. La question en suspens est de savoir si la terminaison du calcul de $\mathcal{A}^*$ (3), qui est assurée au premier ordre [7], est toujours garantie à l'ordre supérieur avec ce codage. Enfin pour les étapes (2) et (4), comme dans le cas des mots, il existe des formats d'expressions régulières et des algorithmes de passage des expressions vers les automates pour les arbres [5]. Cependant, ces formats d'expressions sont parfois plus difficiles à écrire ou à lire que dans le cas des mots. Par exemple, l'expression régulière (telle qu'elle est définie dans [5]) correspondant à notre notation `[a*]`, est l'expression $nil + cons(a, \Box)^{*,\Box}.{}_{\Box} nil$, où *cons* et *nil* sont les constructeurs de listes. Nous cherchons une solution plus naturelle et facilement utilisable par des programmeurs OCaml pour définir des langages d'éléments de types algébriques usuels.

4 Conclusion et perspectives

Dans cet article, nous donnons quelques pistes pour exploiter des outils de réécriture pour la vérification formelle légère de programmes OCaml. La vérification est *formelle* car elle permet de *prouver* certaines propriétés sur ces programmes. D'autre part, elle est *légère* car la preuve est automatique et la famille de propriétés considérées est restreinte aux propriétés régulières. Il reste un certain nombre de problèmes théoriques à dépasser, comme la prise en compte des fonctions d'ordre supérieur. Il reste également des problèmes plus conceptuels, comme la bonne façon d'interagir avec l'interprète. Car si, dans les exemples présentés dans cet article, les expressions régulières sont lisibles, cela ne sera pas toujours le cas : les expressions de langages pourront être trop complexes. Cependant, dans un interprète abstrait il est également possible de poser des requêtes de la forme : `delete(a,[(a|b)*])=[b*]` où la propriété attendue est donnée comme une post-condition du calcul. A noter que, sous certaines conditions, on pourra bien parler d'une égalité. En effet, la précision des approximations est quantifiable [10] et dans ce cas particulier il est possible de garantir à la fois que `delete(a,[(a|b)*])`$\subseteq$`[b*]` et que `delete(a,[(a|b)*])`$\supseteq$`[b*]`. En donnant ainsi l'ensemble des résultats attendus, si le résultat est `true`, il sera possible d'éviter la phase de lecture et d'analyse de l'expression résultat. Cependant, on a vu que la précision de l'approximation n'était pas toujours garantie : on a mentionné que c'était le cas pour la fonction de tri par fusion. L'autre intérêt de cette forme est de donner le langage attendu pour le résultat et de permettre ainsi un calcul par raffinement d'abstraction automatique (CounterExample Guided Abstraction Refinement : CEGAR), dans le cas où l'approximation se révèle trop grossière. Ceci est utilisé dans l'analyse de programmes fonctionnels [15] et a été expérimenté avec succès sur Timbuk [1]. Parfois, la représentation des valeurs n'est pas seulement trop complexe, elle est inaccessible : c'est le cas des types abstraits qui masquent les détails de leur implantation. Ceci empêche d'exprimer le langage de sortie d'une fonction par une expression régulière ou par un automate. En revanche, ceci n'interdit pas la définition et la vérification de propriétés à l'aide de prédicats, comme dans le cas des preuves réalisées par les assistants de preuve. Dans le cas du type abstrait `Set` d'OCaml, on pourra ainsi exprimer et vérifier des propriétés de la forme :

`not(mem a (remove a add*((a|b),empty)))`, où l'expression régulière sera utilisée pour exprimer de façon abstraite l'ensemble construit dans la requête et le résultat sera booléen et non sous la forme d'une expression régulière.

Sur la route menant à la conception d'un interprète abstrait, il existe d'autres embûches pour lesquelles nous avons déjà quelques éléments de réponse. C'est le cas des stratégies d'évaluation (appel par valeur) et des types prédéfinis (`int`, `string`, etc.). Quand l'exécution du programme repose

sur une stratégie d'évaluation, le calcul de l'automate $\mathcal{A}^*$ doit prendre en compte cette stratégie au risque de produire des résultats trop imprécis. Récemment, nous avons montré que cela était possible pour la stratégie d'appel par valeur [12] utilisée par OCaml. D'autre part, les programmes OCaml ne manipulent pas uniquement des termes mais également des valeurs *prédéfinies* comme des `int`, `string`, etc. Nous savons maintenant que l'algorithme de calcul de l'automate A^* peut être étendu de façon naturelle pour prendre en compte ces valeurs particulières [9]. Les automates obtenus marient des transitions classiques (pour la partie terme/structure) et des éléments d'un domaine abstrait de l'interprétation abstraite [6] (pour les ensembles de valeurs prédéfinies). Les expérimentations menées dans [9] permettent de manipuler des langages réguliers de termes où certaines feuilles peuvent être des intervalles abstrayant des ensembles de valeurs entières prédéfinies. Pour l'instant, ces automates ne permettent pas de définir de relation entre les valeurs associées aux feuilles. Par exemple, on ne peut pas décrire le langage des listes triées. En revanche, ils devraient permettre de définir et de raisonner sur des expressions régulières hybrides plus riches dans l'interprète abstrait. Par exemple, une expression de la forme `[(1;+oo)*]`, pourra représenter les listes de longueur quelconque dont tous les éléments sont des entiers appartenant à l'intervalle $[1; +\infty[$.

Remerciements Les auteurs remercient les relecteurs pour leurs remarques et commentaires.

Références

[1] Y. Boichut, B. Boyer, T. Genet, and A. Legay. Equational Abstraction Refinement for Certified Tree Regular Model Checking. In *ICFEM'12*, volume 7635 of *LNCS*. Springer, 2012.

[2] C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. C-shore : a collapsible approach to higher-order verification. In *ICFP'13*. ACM, 2013.

[3] G. Castagna, K. Nguyen, Z. Xu, H. Im, S. Lenglet, and L. Padovani. Polymorphic functions with set-theoretic types : part 1 : syntax, semantics, and evaluation. In *POPL'14*. ACM, 2014.

[4] F. Chalub and C. Braga. A Modular Rewriting Semantics for CML. *J.UCS*, 10(7) :789–807, 2004.

[5] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://tata.gforge.inria.fr`, 2008.

[6] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[7] T. Genet. Towards Static Analysis of Functional Programs using Tree Automata Completion. In *WRLA'14*, volume 8663 of *LNCS*. Springer, 2014.

[8] T. Genet, Y. Boichut, B. Boyer, V. Murat, and Y. Salmon. Reachability Analysis and Tree Automata Calculations. IRISA / Université de Rennes 1. `http://www.irisa.fr/celtique/genet/timbuk/`.

[9] T. Genet, T. Le Gall, A. Legay, and V. Murat. A Completion Algorithm for Lattice Tree Automata. In *CIAA'13*, volume 7982 of *LNCS*, pages 134–145, 2013.

[10] T. Genet and R. Rusu. Equational tree automata completion. *Journal of Symbolic Computation*, 45 :574–597, 2010.

[11] T. Genet and Y. Salmon. Tree Automata Completion for Static Analysis of Functional Programs. Technical report, INRIA, 2013. `http://hal.archives-ouvertes.fr/hal-00780124/PDF/main.pdf`.

[12] T. Genet and Y. Salmon. Reachability Analysis of Innermost Rewriting. In *RTA'15*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. To be published.

[13] N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. *Theoretical Computer Science*, 375(1-3) :120–136, 2007.

[14] N. Kobayashi. Model Checking Higher-Order Programs. *Journal of the ACM*, 60.3(20), 2013.

[15] L. Ong and S. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL'11*. ACM, 2011.

[16] S. Owens. A Sound Semantics for OCamllight. In *ESOP'08*, volume 4960 of *LNCS*, pages 1–15. Springer, 2008.

[17] N. Vazou, P. Rondon, and R. Jhala. Abstract Refinement Types. In *ESOP'13*, volume 7792 of *LNCS*. Springer, 2013.