Vérifier des fonctions d'ordre supérieur à l'aide d'automates d'arbre

Thomas Genet, Timothée Haudebourg, Thomas Jensen

Vérifier des fonctions d'ordre supérieur à l'aide d'automates d'arbre*

Thomas Genet Timothée Haudebourg Thomas Jensen

Univ Rennes, Inria, CNRS, IRISA

14 mai 2018

Les fonctions d'ordre supérieur font partie intégrante des langages de programmation modernes comme Haskell, Caml, mais aussi Java, Scala ou même JavaScript. Là où leur utilité n'est plus à démontrer, leur utilisation pose problème dès lors qu'il s'agit de prouver la correction des programmes qui les utilisent. Un ingénieur soucieux de prouver la correction de son travail pourra se tourner vers des assistants de preuves interactifs (ex. Coq [1] ou Isabelle/HOL [8]), ou écrire une spécification dans un formalisme adapté à la preuve semi-automatique (ex. Liquid Types [9] ou Bounded Refinement Types [11, 10]). Cependant ces approches requièrent une expertise des méthodes formelles utilisées, ainsi que du temps que ce soit pour l'annotation du programme ou sa preuve. D'autres approches visent à vérifier le programme de manière complètement automatique : la preuve est réalisée sans annotation ni lemme intermédiaire. Ces approches, comme le model-checking d'ordre supérieur [7, 4, 5, 6], sont accessibles à un public plus important, mais sont applicables à un ensemble de propriétés plus restreint.

Dans ce travail [2], nous poursuivons cette ligne de recherche en proposant une approche basée sur les systèmes de réécriture de termes (TRS). Les TRS permettent une représentation formelle simple des programme fonctionnels. Dans ce formalisme, la fonction *filtrer* est définie par le TRS $\mathcal{R}$ suivant :

$$\begin{aligned} @(@(\mathit{filtrer}, \underline{p}), cons(\underline{x}, \underline{l})) \rightarrow\ & \textbf{si } @(p, x) \\ & \textbf{alors } cons(\underline{x}, @(@(\mathit{filtrer}, \underline{p}), \underline{l})) \\ & \textbf{sinon } @(@(\mathit{filtrer}, \underline{p}), \underline{l}) \\ @(@(\mathit{filtrer}, \underline{p}), vide) \rightarrow\ & vide \end{aligned}$$

On remarquera que *filtrer* est une fonction d'ordre supérieur : elle prend en premier paramètre p qui est lui même une fonction (prédicat). Le symbole @ est utilisé ici pour représenter l'application. Ainsi $@(\underline{p}, \underline{x})$ signifie « x appliqué à p ». Les termes

*Cet article est un résumé long de l'article *Verifying Higher-Order Functions with Tree Automata* [2] accepté à la conférence FoSSaCS 2018

soulignés représentent les variables. Un TRS est donc un ensemble de règles régissant l'évaluation des expressions/termes d'un programme. Évaluer une expression, un terme, reviens à le réécrire en appliquant les règles définies dans $\mathcal{R}$. Par exemple $@(@(\mathit{filtrer}, \mathit{pair}), \mathit{cons}(0, \mathit{cons}(s(0), \mathit{vide})))$ se réécrit (en plusieurs étapes) en $\mathit{cons}(0, \mathit{vide})$. Ainsi vérifier qu'une fonction est correcte consiste à vérifier que l'ensemble des termes atteignables, en appliquant $\mathcal{R}$ à l'ensemble des entrées, est lui-même « correct ». Soit $\mathcal{L}$ le *langage* (l'ensemble) des termes initiaux. On définit un ensemble Err de termes « interdits », en théorie inatteignable si la fonction considérée est correcte. L'objectif ne notre méthode est de vérifier qu'aucun terme de Err n'est effectivement atteignable en réécrivant un terme $\mathcal{L}$ avec $\mathcal{R}$. Autrement dit que $\mathcal{R}^*(\mathcal{L}) \cap Err = \emptyset$, où $\mathcal{R}^*(\mathcal{L})$ est l'ensemble des termes atteignables à partir de $\mathcal{L}$ avec $\mathcal{R}$.

Dans notre exemple, $\mathcal{L}$ est l'ensemble des termes de la forme $@(@(\mathit{filtrer}, \mathit{pair}), l)$ où l est une liste quelconque d'entier naturels et Err est l'ensemble des liste contenant au moins un entier impair (on souhaite vérifier que seuls les entiers pairs sont conservés).

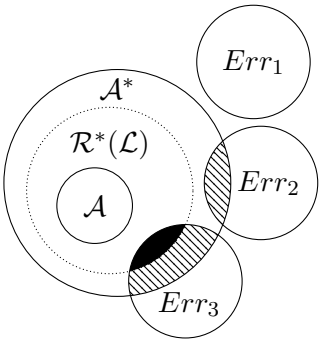

FIGURE 1 – Vérification à l'aide d'automates d'arbres. L'automate $\mathcal{A}$ représente le langage initial $\mathcal{L}$. $\mathcal{A}^*$ est calculé à partir de $\mathcal{A}$ comme une sur-approximation de $\mathcal{R}^*(\mathcal{L})$ (qui n'est jamais explicitement calculé). La fonction n'est vérifiée que si aucun terme de Err n'est dans $\mathcal{A}^*$.

En pratique les langages $\mathcal{L}$ et Err sont représentés à l'aide d'*automates d'arbres*. Si l'on nomme $\mathcal{A}$ l'automate représentant $\mathcal{L}$, notre méthode consiste ensuite à « compléter » [3] $\mathcal{A}$ à l'aide de $\mathcal{R}$ afin d'obtenir une sur-approximation, la plus précise possible, de $\mathcal{R}^*(\mathcal{L})$ en un automate $\mathcal{A}^*$. Il est ensuite possible de tester l'intersection de $\mathcal{A}^*$ avec Err. Si celle-ci est vide, la fonction est prouvée correcte. Sinon il est *possible* d'avoir $\mathcal{R}^*(\mathcal{L}) \cap Err \neq \emptyset$ (cas Err_2 ou Err_3 de la figure 1). Dans cet article, nous avons défini un mécanisme automatique de calcul de $\mathcal{A}^*$ pour une classe de programmes fonctionnels relativement large. Il s'agit des programmes d'ordre supérieur terminants, complets et ne construisant pas de « tours d'applications partielles », c'est à dire des structures arbitrairement grandes contenant des fonctions partiellement appliquées. L'implantation de cette technique a permis de prouver automatiquement des propriétés non triviales sur des programmes d'ordre supérieur classiques (`http://people.irisa.fr/Thomas.Genet/timbuk/funExperiments`).

Références

[1] Coq. The coq proof assistant reference manual : Version 8.6. 2016.

[2] Thomas Genet, Timothée Haudebourg, and Thomas Jensen. Verifying Higher-Order Functions with Tree Automata. In *21st International Conference on Foundations of Software Science and Computation Structures*, 2018.

[3] Thomas Genet and Vlad Rusu. Equational approximations for tree automata completion. *Journal of Symbolic Computation*, 45(5) :574–597, 2010.

[4] Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 416–428, 2009.

[5] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. Predicate abstraction and CEGAR for higher-order model checking. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 222–233, 2011.

[6] Yuma Matsumoto, Naoki Kobayashi, and Hiroshi Unno. Automata-based abstraction for automated verification of higher-order tree-processing programs. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, pages 295–312, 2015.

[7] C.-H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 81–90. IEEE, 2006.

[8] Lawrence C Paulson et al. The isabelle reference manual. Technical report, University of Cambridge, Computer Laboratory, 1993.

[9] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169, 2008.

[10] Niki Vazou, Alexander Bakst, and Ranjit Jhala. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 48–61, 2015.

[11] Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. Abstract refinement types. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 209–228, 2013.