Proposition de stage – 2ème année de Master Recherche

Vérification déductive de programmes respectant une discipline de typage à la Rust

Lieu du stage

Équipe Vals
Laboratoire de Recherche en Informatique
Bâtiment 650 "Ada Lovelace"
Université Paris Saclay
91405 Orsay cedex
France
`http://vals.lri.fr`

Encadrement

Jacques-Henri Jourdan
Tél : +33 1 69 15 67 35
Email : `jacques-henri.jourdan@lri.fr`

Claude Marché
Email : `claude.marche@inria.fr`

Contexte

Le domaine de recherche des *méthodes formelles* propose d'utiliser des outils mathématiques et logiques afin d'exclure avec un haut degré de confiance certaines classes de bugs. L'une de ces méthodes, la vérification déductive de programmes, consiste à traduire en un énoncé logique la correction d'un programme préalablement annoté de sa spécification. On utilise ensuite des outils plus ou moins automatisés afin d'obtenir une preuve formelle de cet énoncé, dont la validité sera garantie par la machine.

La vérification déductive de programmes est un domaine en pleine activité. De nombreux outils sont développés aujourd'hui, et plusieurs succès récents témoignent de ces avancées. On peut par exemple citer le compilateur formellement vérifié CompCert, le micro-noyau de système d'exploitation vérifié seL4 et la vérification de l'hyperviseur Hyper-V de Microsoft.

Des défis importants restent cependant à relever pour populariser ces méthodes de preuves de programmes : il est nécessaire de permettre la vérification d'une large variété de programmes, en particulier ceux qui utilisent des pointeurs vers de la mémoire modifiable, tout en automatisant au maximum les outils pour limiter la quantité de travail manuel d'un utilisateur. Ces deux objectifs sont a priori contradictoires : lors de la vérification de programmes utilisant des pointeurs, il est nécessaire de prouver très régulièrement que ceux-ci ne font pas d'alias pour s'assurer que la modification d'une structure de données à travers un pointeur n'en modifie pas aussi une autre que l'on veut garder inchangée. Ces conditions de non-aliasing polluent les preuves et les spécifications et peuvent considérablement augmenter le travail nécessaire pour vérifier un programme. Afin de concilier ces deux objectifs, certains outils comme Why3 [2, 7] ou Cogent[10] proposent d'autoriser les pointeurs dans les programmes à prouver (en Why3, sous la forme de référence mutable), mais de limiter leur utilisation grâce à un système de types qui donne les

garanties de non-aliasing nécessaires à la simplification des spécifications et des obligations de preuves. Malheureusement, les contraintes imposées par un tel système de types peuvent rendre difficile l'écriture de certains programmes. Il y a donc un équilibre à trouver entre l'expressivité du système de types et la simplicité des obligations de preuves.

Pendant ce stage, nous proposons d'explorer un nouveau compromis en développant une logique de programmes pour les programmes bien typés dans un fragment du langage de programmation Rust [1]. Ce nouveau langage de programmation a un système de types conçu dans le but de concilier l'efficacité d'une gestion de la mémoire manuelle et bas-niveau à la C++ tout en garantissant la sûreté à l'exécution comme en OCaml. Ce tour de force a été rendu possible notamment grâce aux garanties de contrôle des alias apportées par le système de types. De plus, sa relative simplicité d'utilisation en pratique ont fait de Rust un langage largement utilisé dans l'industrie malgré sa jeunesse. Il est donc naturel de faire bénéficier la vérification déductive des avancées de ce langage en terme de contrôle fin de l'aliasing afin d'améliorer la flexibilité d'utilisation du langage tout en maintenant un bon niveau d'automatisation.

Description du travail de stage

Le premier objectif du stage sera de développer *sur papier* une logique de programme pour une version idéalisée du langage Rust et de son système de types. On pourra pour cela réutiliser le langage noyau λRust déjà utilisé lors de travaux précédents concernant la preuve formelle de sûreté de typage de Rust dans le projet ERC RustBelt [3, 9, 6]. Cette logique de programme devra être adaptée à l'automatisation des preuves via l'utilisation de démonstrateurs automatiques SMT. Ainsi, il sera nécessaire que cette logique permette un calcul automatique de la plus faible précondition, et sa logique d'assertions devra être du premier ordre et ne pas faire intervenir de notion de possession comme en logique de séparation. Pour arriver à cet objectif, il faudra utiliser les garanties de non-aliasing apportées par le système de types.

Le principal obstacle lors de cette première étape sera le traitement des *emprunts mutables*. Cette fonctionnalité centrale du système de types de Rust permet de transférer temporairement la possession d'une structure de données à une autre fonction (dans une bibliothèque, par exemple) afin notamment d'effectuer des modifications dans cette structure de données. Lorsque cet emprunt se termine, à la fin de sa durée de vie (un instant connu statiquement lors de la preuve du programme), le contenu de la structure de données peut avoir changé, et il faut pouvoir raisonner sur ce changement "indirect" dans la preuve du programme. Afin de lever cette difficulté, nous pourrons nous appuyer sur plusieurs travaux considérant des problèmes proches :

- On pourra considérer une idée récente consistant à utiliser une *variable de prophétie* afin de prédire, au moment de la création de l'emprunt, la valeur de la variable empruntée lors de la fin de la durée de vie de l'emprunt.
- On pourra réutiliser les travaux de Charguéraud et Pottier [5] permettant la traduction fonctionnelle des programmes types avec des capacités et des régions. Il faudra alors interpréter les durées de vies de Rust par des régions.
- On pourra s'inspirer des travaux d'Astrauskas, Müller, Poli et Summers [4], dans le cadre du projet Prusti : il s'agit d'encoder les programmes écrits en Rust dans la logique de l'outil Viper, une version de la logique de séparation.
- On pourra également s'inspirer des travaux expérimentaux d'extension du langage Spark/Ada aux programmes Ada avec pointeurs [8], qui proposent une forme de traduction vers Why3.

Le second objectif sera d'implanter la logique de programme dans un prototype logiciel d'outil de vérification déductive de programmes écrits en Rust. On concentrera les efforts sur un petit fragment du langage, et délèguera à Why3 le calcul final d'obligation de preuve par calcul de

plus faible précondition et l'interaction avec les démonstrateurs SMT. L'utilisation de Why3 en tant que langage intermédiaire fait partie des principaux cas d'utilisation pour lesquels cet outil a été conçu.

Afin d'atteindre ces objectifs, le stagiaire devra passer une partie significative de son temps de travail à l'étude de la bibliographie du domaine. Il est attendu que ce travail débouche sur une synthèse comparative des différentes approches, et la proposition d'un nouveau schéma de traduction de codes à la Rust vers Why3, et d'une preuve papier de correction de cette traduction. Par ailleurs, il est attendu que ce stage comporte une partie pratique de programmation (a priori en OCaml) d'un prototype, afin de valider expérimentalement le schéma de traduction proposé.

Prérequis souhaités

— Une connaissance de niveau Master en logique informatique, en vérification et en théorie des langages de programmation est requise.
— On attend une certaine aisance dans la programmation dans le langage OCaml. Une connaissance du langage Rust serait naturellement un plus.

Références

[1] Langage de programmation Rust. `http://rust-lang.org/`.

[2] Outil de vérification déductive Why3. `http://why3.lri.fr/`.

[3] Projet ERC RustBelt. `http://plv.mpi-sws.org/rustbelt/`.

[4] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. Leveraging rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA) :147, 2019.

[5] Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ICFP*, volume 8, pages 213–224, 2008.

[6] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. Rustbelt meets relaxed memory. In *POPL*. ACM, 2020.

[7] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. A pragmatic type system for deductive verification. Technical report, Université Paris Sud, 2016. `https://hal.archives-ouvertes.fr/hal-01256434v3`.

[8] Georges-Axel Jaloyan, Claire Dross, Maroua Maalej, Yannick Moy, and Andrei Paskevich. Verification of programs with pointers in SPARK. working paper `https://hal.inria.fr/hal-01936105`, November 2018.

[9] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt : Securing the foundations of the rust programming language. In *POPL*. ACM, 2017.

[10] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint : Bringing down the cost of verification. In *ICFP*, pages 89–102. ACM, 2016.