Preuves de correction automatiques des programmes

Pour tenter d'éliminer les bogues de leurs programmes, les développeurs testent leur code sur des exemples. Mais pour être assuré qu'un programme est correct, il faut écrire une preuve qui assure son bon fonctionnement. C'est un processus long et fastidieux que l'on aimerait pouvoir déléguer à un ordinateur.

Dans ce TIPE on essaye de transformer des propriétés sur des programmes en preuves de correction.

Positionnement thématique (étape 1) :

- Informatique (Informatique théorique)
- Mathématiques (Autres)
- Informatique (Informatique pratique)

Mots-clés (étape 2)

Mots-clés (en français)

- Démonstration automatique de théorème
- Programmation fonctionnelle
- Système de réécriture
- Arbre de preuve
- Induction structurelle

Mots-clés (en anglais)

- Automated theorem proving
- Functional programming
- Rewriting system
- Gentzen tree
- Structural induction

Bibliographie commentée

Une partie de l'étude de la correction d'un programme consiste à déterminer si ce programme termine sur toutes ses entrées et s'il vérifie des propriétés appelées post-conditions décrivant les effets attendus à l'issue de son exécution.

L'analyse de la correction des programmes rencontre un problème fondamental : il n'existe pas de programme vérifiant la terminaison d'un autre programme. Ce problème, appelé le problème de l'arrêt, est alors dit indécidable [1]. Ce résultat permet de déduire que d'autres problèmes sont indécidables grâce aux réductions. Un résultat important est le théorème de Rice qui énonce que pour toute propriété dite non triviale de la sémantique d'un programme, le problème de tester cette propriété est indécidable [2].

On peut, pour contourner ce problème, restreindre notre langage de programmation pour ne contenir que des fonctions pour lesquelles il est possible de prouver la terminaison. Un tel langage est dit total [3]. Pour prouver qu'un algorithme termine, il suffit de trouver un objet, appelé variant, qui décroît strictement au cours de la progression de l'algorithme et qui est à valeurs dans un ensemble dit bien ordonné.

Quant à la vérification des post-conditions. Il faut se munir de règles formelles de déduction pour que l'ordinateur puisse raisonner. Ces règles sont appelés règles d'inférence. Et un ensemble de telles règles est appelé un système de preuve.

Le typage est un exemple de propriété sur les variables d'un programme. C'est une propriété importante car souvent nécessaire pour la compilation du code. Dans de nombreux langages l'utilisateur doit donc explicitement donner le type de chaque variable. Il est alors nécessaire, à minima, de vérifier que l'utilisation des variables est cohérente avec leur type. Le système de types de Hindley-Milner donne un système de preuves qui permet de vérifier cette cohérence. Et même mieux que cela, il existe un algorithme pouvant déduire les types automatiquement [4]. Ainsi, dans certains langages de programmation, les types sont déterminés automatiquement.

Les triplets de Hoare, eux, formalisent les effets des instructions d'un algorithme sur les préconditions [5]. Ce système de preuves décrit, pour chaque opération élémentaire d'un langage de programmation impératif, comment démontrer que sous certaines hypothèses une opération donnée produit certains effets. Dans un langage de programmation fonctionnel, les boucles sont remplacées par des appels récursifs. Il faut donc d'autres règles d'inférence, dont une qui décrit les effets d'un appel récursif.

En mathématiques, on utilise le théorème de récurrence qui permet de prouver une propriété en examinant deux façons qui suffisent pour construire tous les entiers : le constructeur zéro et le constructeur successeur. Et ce théorème marche même si cette propriété fait intervenir un objet défini par récurrence. Ce type de raisonnement s'étend aux objets dits inductifs, qui sont construits avec des objets de base et des règles de construction. On possède alors un théorème qui généralise le principe de récurrence, appelé théorème d'induction structurelle [6]. De plus les objets inductifs produisent naturellement un ordre bien fondé, on peut donc aussi les utiliser comme variants pour prouver la terminaison.

Une fois le système de preuves défini, on a les outils permettant de construire des preuves. Mais leur recherche automatique est souvent très difficile. Sans compter qu'il se pourrait même qu'il existe une post-condition vraie, mais ne pouvant pas être prouvée dans le système de preuves choisi.

Problématique retenue

Comment générer des preuves de correction automatiquement dans un langage de programmation total et purement fonctionnel ?

Objectifs du TIPE du candidat

- Définir un langage de programmation jouet.
- Construire un parseur du langage pour produire la structure en mémoire des programmes à manipuler.
- Réussir à prouver des égalités sémantiques par réécritures successives.
- Réussir à trouver et prouver un variant pour assurer la terminaison.
- Réussir à prouver des propriétés, portant sur des fonctions récursives, par induction structurelle.

Références bibliographiques (étape 1)

[1] A. M. Turing, « On Computable Numbers, with an Application to the Entscheidungsproblem ». doi: 10.3217/jucs-010-07-0751.

[2] H. G. Rice, « Classes of Recursively Enumerable Sets and Their Decision Problems ». [En ligne]. Disponible sur: https://archive.wikiwix.com/cache/?url=https%3A%2F%2Fwww.jstor.org%2Fsici%3Fsici%3D0002-9947(195303)74%253A2%253C358%253ACORESA%253E2.0.CO%253B2-N

[3] D. A. Turner, « Total Functional Programming ». 28 juillet 2004. doi: 10.3217/jucs-010-07-0751.

[4] Robin Milner, « A Theory of Type Polymorphism in Programming ». doi: 10.1016/0022-0000(78)90014-4.

[5] C. A. R. Hoare, « An axiomatic basis for computer programming ». doi: 10.1145/363235.363259.

[6] Gérard Huet et J.M. Hullot, « Proofs by induction in equational theories with constructors ». [En ligne]. Disponible sur: https://hal.inria.fr/inria-00076533/file/RR-0028.pdf

DOT

[1] : De février à juin 2024 - recherche d'un sujet de TIPE puis planification de celui-ci.

[2] : De juillet à septembre 2024 - implémentation d'un analyseur léxicographique puis d'un parseur pour OCaml ; permettant de transformer le code source d'un fichier en arbre de syntaxe.
[3] : Octobre 2024 - création d'identificateurs uniques pour lier les variables dans l'arbre de syntaxe.
[4] : De novembre à décembre 2024 - implémentation des règles de réécritures et de l'algorithme de preuves ; permettant de prouver des propriétés non récursives.
[5] : De février à mars 2025 - implémentation de l'algorithme d'inférence de types de Hindley-Milner.
[6] : D'avril à mai 2025 - réécriture majeure du code pour utiliser les identificateurs de Brujin.