Preuve d'un micro-compilateur pour le lambda-calcul non typé

Thomas Laure

Mai 2021

Introduction

J'ai choisi ce projet pour l'occasion d'apprendre un peu plus à utiliser Coq, et comprendre un peu mieux le fonctionnement des assistants de preuve.

J'ai trouvé ce projet très intéressant pour cela, bien que j'ai passé beaucoup de temps à chercher comment faire des choses stupides, et à fouiller la bibliothèque Arith.

J'appréhende un peu mieux le fonctionnement de Coq, et j'ai appris qu'il faut raisonner de la manière la plus inductive possible pour que les preuves passent correctement.

Malheureusement, j'ai été rattrapé à la fin du projet par la surcharge de travail de la fin mai, et j'ai dû renoncer à le terminer. Les deux dernières preuves n'ont pas pu être faites.

1 Indices de De Bruijn et manipulation de λ-termes

On commence par définir inductivement les λ-termes avec indices de De Bruijn, et le prédicat $C[n](t)$. Ensuite, la preuve que $C[n](t) \Rightarrow C[n+1](t)$ se fait simplement avec les lemmes de Arith.Lt.

Pour la substitution, il faut prendre garde au fait qu'il faut décaler les indices des varaibles libres du terme par lequel on substitue d'autant qu'on a rencontré de λ-abstractions sur le chemin. Pour cela, on définit une fonction shift qui décale les indices des variables supérieures à une limite l de p crans. (et qu'on appliquera avec $l = 0$).

On fait la substitution en 3 étapes : une fonction qui prend en argument un entier i, un entier j et deux termes u et v ainsi qu'un décalage p, et qui renvoie u shifté de p crans si $i = j$ et v sinon (correspond à la substitution de $v = varj$ par u). Ensuite, la fonction qui substitue dans t avec un décalage p, et enfin celle qui appelle la précédente avec un déacalage initial nul.

Pour la substitution parallèle, on applique la même stratégie, avec une liste et donc une première fonction qui renvoie le bon terme dans la liste.

De nombreux lemmes sont démontrés pour les substitutions. Autant que possible, j'ai adopté pour stratégie de décomposer mes preuves. Ainsi, les propriétés sont généralement prouvées d'abord avec la première fonction pour les variables, puis avec la seconde fonction pour le terme, et enfin avec la fonction de substitution principale.

Particulièrement pour la dernière preuve de la partie, on commence par faire un certain nombre de lemmes intermédiaires, qui permettent de prouver que la fonction de substitution parallèle dans une variable renvoie bien v dans un cas, le premier terme de la liste si $i = j =$, et un autre terme de la liste sinon.

2 Sémantique des λ-termes

Partie qui contient beaucoup moins de difficultés. Pour définir la réduction en un nombre quelconque d'étapes, on commence par définir la réduction en nombre n d'étapes, et pour prouver la clôture au contexte, on prouve d'abord celle de cette relation.

3 La machine abstraite de Krivine

On remarque en observant la fonction τ de la partie suivante que le code vide pourrait nous poser quelques soucis. On cherche donc à l'évacuer.

Or, on remarque que la fonction de compilation permet de garantir que :

1. Le code qui suit un Grab n'est jamais vide.
2. Les codes sous un Push et après un Push ne sont jamais vides.
3. Un Access est toujours la dernière instruction d'un code.

De plus, on s'aperçoit aisément que les 4 transitions de la sémantique permettent de conserver cet invariant en y ajoutant que tout code situé dans un couple de l'environnement ou de la pile n'est jamais vide.

Dès lors, on peut définir formellement le code cette manière. Grab est suivi d'un code. Push prend en argument deux codes, et Access ne prend en argument qu'un entier.

Ensuite, on définit la fonction sémantique qui renvoie un type option, et None quand il n'existe pas de transition.

J'ai défini les couples avec des constructeurs et sans utiliser de liste, car j'ai remarqué que Coq peinait à faire du calcul avec les couples, notamment un bug rencontré plusieurs fois était la difficulté de faire des calculs lorsque l'on prend deux composantes d'un couple pour former un triplet avec une troisième, ce qui bloquait les preuves de corrections.

4 Compilation

La fonction compile ne présente aucune difficulté.

5 Correction de la compilation

On définit τ par étapes. D'abord pour du code, puis pour un environnement (qui renverra alors une liste de termes par laquelle la substitution parallèle va s'effectuer, et enfin pour une pile. Pour celles-ci, on utilise un accumulateur du fait de l'associativité gauche de l'application des $\lambda-$termes.

On définit la corrections des états en procédant à nouveau par étapes.

Enfin, on fait la preuve qu'une transition depuis un état correct est soit inexistante, soit vers un état correct.

Les preuves qu'une transition entre états corrects correspond à la $\beta-$réduction et la correction de la compilation ont dû être abandonnées par manque de temps.