Lazi : Prouver

Emmanuel Chantréau

1er septembre 2016

Table des matières

1 Introduction

Ce texte présente le système de production de preuves en Lazi. Il sert également, du point de vue informatique, à la vérification des types (la partie analyse sémantique des compilateurs).

2 Unifier la production de preuve et la vérification des types

Nous avons vue que les systèmes de types en informatique peuvent être vue simplement d'un point de vue mathématique par des propriétés. La partie "vérification des types" des compilateurs sert à vérifier que les éléments du langage possèdent certaines propriétés. Il n'y a que très peu de modifications à apporter pour mathématiser la vérification des types :

2.1 Mathématiser les déclarations

Historiquement une déclaration de fonction comme "char f(int x) ;" implique que si on utilise f avec autre chose qu'un argument de type "int" alors il y a erreur. Puis on a voulue surcharger les fonctions pour permettre différents types d'arguments.

En mathématique nous ne pouvons pas interdire, et donc le polymorphisme va de soit. On peut traduire "char f(int x) ;" par "si x est de type int alors f x est de type char". Comment alors signaler une erreur si par exemple le programmeur a écrit "f 2.15" ? Dans ce cas le système de vérification des types n'arrivera pas à trouver le type de "f 2.15" et donc bloquera sur cette expression (sauf si la surcharge est prévue et dans ce cas le type de "f 2.15" sera calculable).

Remarquons que la preuve de "f x est de type char" est plus forte que le service offert par les compilateurs classiques qui en général ne trouvent rien à redire si f boucle indéfiniment (et donc ne retourne pas une chose de type char).

2.2 Prouver les déclarations

La phase de vérification des types en Lazi consiste à lire le code source comme une preuve. Par exemple une déclaration de type est vu comme une déduction. Grâce aux extensions Lazi le programmeur n'a en général pas à fournir de preuve par lui-même car c'est l'extension qui fait ce travail (comme le font les compilateurs lors de l'analyse sémantique).

Mais que se passe-t-il si dans l'exemple précédant ni l'extension ni le programmeur ne peuvent prouver que f ne boucle pas ? On peut alors ajouter des hypothèses et dans ce cas la preuve de la validité des types est remplacée par une preuve d'une règle où les conditions sont les hypothèses ajoutées (comme celle que f ne boucle pas) et la conclusion la validité des types.

3 Un code source et deux lectures

En vue de calculer des expressions grâce à lazi-compute on peut définir des fonctions, c'est le cas par exemple dans les fichiers lazi de définition des fondations mathématiques. Dans ce cas les noms définis grâce à la notation "`$Def`" sont interprétés comme des noms définis par lazi-compute et les déclarations de types sont ignorées.

On peut utiliser un même fichier source lazi en tant que preuve : dans ce cas les notations "`$Def`" sont interprétées comme des définitions de noms par l'extension Lazi de définition de noms, et les déclarations de types ou autres notations de déductions sont interprétés comme telles. Cette lecture se fait d'abord par une phase de traduction (lazi-translate) puis par l'exécution d'un programme (en lazi) de vérification de preuve.

4 Un système de cache

Vérifier une preuve ou la sémantique d'un code source peut être coûteux en temps de calcul. C'est pourquoi les compilateurs ont en général un système de cache pour éviter de tout refaire à chaque fois. Les mathématiciens procèdent de même : ils s'appuient sur des théorèmes déjà démontrés sans refaire la vérification eux-même.

Il est ici aussi possible d'utiliser un système de cache : il suffit d'écrire dans un fichier le partie de la preuve déjà vérifiée. Pour qu'il soit effectif il faut toutefois prévoir une gestion des dépendances entre les noms définis et les déductions.

On peut alors avoir toute une bibliothèque d'assertions sur lesquelles s'appuyer et qui ne nécessitent pas d'être d'être redémontrées.