PI : Inégalités linéaires et vérification de programmes

Gustave Billon, François Hublet

1er juin 2018

1 Présentation

Nous avons implémenté un vérificateur d'invariants linéaires de programmes pour un sous-langage de C *très proche* de la spécification de l'énoncé et de la syntaxe des exemples fournis (variables entières, `if` et boucles `while`). L'expression *très proche* se justifie par la résolution d'inconsistances observées entre les exemples et la spécification de l'énoncé, ainsi que par l'adoption d'une syntaxe légèrement plus générale quant à la forme des combinaisons linéaires et des invariants.

Nous avons également implémenté les trois extensions proposées, c'est-à-dire la possibilité de vérifier si un point du code est inatteignable, le calcul automatique d'invariants à partir d'invariants au début et à la fin du programme ainsi qu'au début de chaque boucle, ainsi qu'une simplification des invariants.

Voici un exemple de programme vérifié :

```
int x, y, z;
y = x;
z = 0;
if(x > y) {
        while(x > 0) {
                { unsat }
                z = z - 1;
                x = x - 1;
        }
}
else {
        while(x < 0) {
                { y <= 0 && z >= 0 }
                z = z - y;
                x = x + 1;
        }
}
{ z >= 0 }
```

2 Organisation du projet

Le projet, codé en Caml, comporte huit parties principales :

- Un fichier `linlang.ml` où se trouve le corps du programme ;
- Un fichier `lexer.mll` qui définit l'analyse lexicale du programme en entrée ;
- Des fichiers `parser.mly`, `parser.ml`, `parser.mli` qui définissent l'analyse grammaticale du programme en entrée ;
- Un fichier `types.ml` qui définit les types utilisés pour les arbres syntaxiques ;
- Un fichier `simplex.ml` implémentant l'algorithme du simplexe ;
- Des fichiers `fourrierMotzkin.ml` et `fourrierMotzkin.mli` implémentant l'algorithme de Fourrier-Motzkin ;
- Un ensemble de programmes à vérifier pour tester l'algorithme, situés dans le dossier `exemples/` ;
- Un `Makefile` pour la compilation. On compile l'algorithme de vérification par la commande `make linlang` puis on l'exécute avec la commande `./linlang < fichier_à_tester`.

Le corps du programme, situé dans le fichier `linlang.ml`, prend en entrée l'abre syntaxique construit par le lexer et le parser, sous forme d'une liste d'instructions et d'une liste d'invariants. Il est structuré de la façon suivante :

— La méthode `abstract_prog`, définie récursivement avec les fonctions `abstract_block`, `abstract_assignement`, `abstract_if` et `abstract_while`, convertit l'arbre syntaxique du programme compilé à vérifier en une structure de données traitable par l'algorithme du simplexe. C'est lors de cette phase que les invariants non spécifiés dans le programme en entrée sont complétés.
— les méthodes `verify_block`, `verify_assignement`, `verify_if` et `verify_while` vérifient les invariants du programme ainsi transformé. Le cœur de la vérification se trouve dans la méthode `verify_expr`, qui applique l'algorithme du simplexe.

Pour effectuer l'analyse syntaxique du programme en entrée, nous avons utilisé les programmes `Ocamllex` et `Ocamlyacc`.

Le fichier `simplex.ml` comporte deux sous-modules :
— Un module `Fraction` qui implémente les opérations élémentaires sur les rationnels ;
— Un module `LinearOperations` qui regroupe les opérations sur les matrices de fractions.
Le programme est structuré de la façon suivante :
— La fonction `simplex_basis` prend en entier un tableau canonique pour l'algorithme du simplexe : la première ligne représente la première ligne à minimiser et les suivantes les contraintes. La dernière colonne correspond aux constantes. Tout élément de la dernière colonne sauf le premier, qui représente la valeur de la forme linéaire à minimiser, doit être positif. Enfin, les colonnes de la matrice identité de taille k, où k est défini comme dans l'énoncé, doivent se trouver dans le tableau.
— La fonction `simplex` est le point d'entrée du programme. Elle prend en entrée une matrice `a`, un vecteur `b` et des entiers `k` et `l`, qui sont ceux de l'énoncé : `a` et `b` contiennent les coefficients a_{ij} et b_i de l'énoncé, avec les b_i non nécessairement positifs. Elle construit le tableau canonique en appliquant une première fois le simplexe, puis résout le simplexe sur ce tableau.

3 Structures de données

Les principales structures de données utilisées sont les suivantes :
— On a choisi de représenter les combinaisons linéaires à l'aide de fractions, définies dans le module `Fraction`, afin de mener les calculs de façon purement algébrique.
— Les inégalités étant beaucoup sujettes à des manipulations d'algèbre linéaire, notamment pour l'algorithme du simplexe, elles sont représentées par des `Fraction.frac array`. L'algorithme du simplexe est par ailleurs codé de manière impérative, ce qui se prête mieux aux calculs matriciels.
— Les programmes sont représentés par des arbres syntaxiques définis dans le fichier `types.ml`. Il y a deux ensembles de types différents dans ce fichier, le premier pour représenter le programme compilé par `Ocamllex` et `Ocamlyacc`, le second plus adapté à l'algorithme de vérification. Dans les deux cas, un programme (types `pprog` et `prog`) est une liste d'invariants et une liste d'instructions. La principale différence entre les deux formats d'abres syntaxiques se situe dans le type des invariants : la fonction `abstract_prog` prend en entrée des invariants typés de façon récursifs (type `pinv`), et donne en sortie des invariants en forme normale disjonctive représentés par des `Fraction.frac array list list` (type `inv`, le type `extended_inv` comportant un constructeur `Unsat` qui permet de traiter les points du code inatteignables).