Vérification de programmes linéaires

Gustave Billon, François Hublet

École polytechnique, Palaiseau

22 mai 2018

Module Fraction

- Représentation des rationnels.
- Définition des opérations algébriques.
- On n'accepte que les entiers inférieurs à $\frac{\texttt{Int32.max\_int}}{100}$.
- Pour la sortie du simplexe :
  - `Infty` signifie que la forme linéaire n'est pas minorée ;
  - `Void` signifie que les contraintes ne sont pas satisfiables

Algorithme du simplexe

Prend en entrée une matrice a qui représente $\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0l} \\ \vdots & & & \vdots \\ a_{k0} & a_{k1} & \cdots & a_{kl} \end{bmatrix}$

et un vecteur b qui représente $\begin{bmatrix} b_0 \\ \vdots \\ b_k \end{bmatrix}$ et résout le problème :

Minimiser $-a_{00}z_0 - a_{01}z_1 - \cdots - a_{0l}z_l + b_0$ *sous les contraintes :*

$$\begin{cases} a_{10}z_0 + a_{11}z_1 + \cdots + a_{1l}z_l & \leq b_1 \\ a_{21}z_0 + a_{22}z_1 + \cdots + a_{2l}z_l & \leq b_2 \\ \vdots & \\ a_{k0}z_0 + a_{k1}z_1 + \cdots + a_{kl}z_l & \leq b_k \end{cases}$$

et $z_i \geq 0$.

On introduit des variables auxiliaires $y_1, \ldots, y_k$ et le problème devient :

Minimiser

$$-a_{00}z_0 - a_{01}z_1 - \cdots - a_{0l}z_l + b_0$$

sous les contraintes :

$$\begin{cases} y_1 + a_{10}z_0 + a_{11}z_1 + \cdots + a_{1l}z_l &= b_1 \\ y_2 + a_{21}z_0 + a_{22}z_1 + \cdots + a_{2l}z_l &= b_2 \\ \vdots & \\ y_k + a_{k0}z_0 + a_{k1}z_1 + \cdots + a_{kl}z_l &= b_k \end{cases}$$

et $y_i \geq 0$, $z_i \geq 0$.

Si les b_i sont positifs, on considère le tableau :

$$\begin{bmatrix} 0 & 0 & \cdots & 0 & a_{00} & a_{01} & \cdots & a_{0l} & b_0 \\ 1 & 0 & \cdots & 0 & a_{10} & a_{11} & \cdots & a_{1l} & b_1 \\ 0 & 1 & \cdots & 0 & a_{20} & a_{21} & \cdots & a_{2l} & b_2 \\ \vdots & & & & & & & & \vdots \\ \vdots & & & & & & & & \vdots \\ \vdots & & & & & & & & \vdots \\ 0 & 0 & \cdots & 1 & a_{k0} & a_{k1} & \cdots & a_{kl} & b_k \end{bmatrix} \begin{bmatrix} -y_1 \\ -y_2 \\ \vdots \\ -y_k \\ -z_0 \\ \vdots \\ -z_l \\ 1 \end{bmatrix} = \begin{bmatrix} \text{forme à minimiser} \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

et on résout le problème par pivots successifs.

En pratique, on veut par exemple minimiser $-4z_0 + 2z_1 + 1$ avec

$$\begin{cases} -4z_0 + 6z_1 & \leq 6 \\ 4z_0 - 3z_1 & \leq -1 \end{cases}$$

et les z_i non nécessairement positifs.
On considère alors le problème :

Minimiser $-4z_0 + 4z'_0 + 2z_1 - 2z'_1 + 1$ *avec*

$$\begin{cases} y_1 - 4z_0 + 4z'_0 + 6z_1 - 6z'_1 & = 6 \\ y_2 + 4z_0 - 4z'_0 - 3z_1 + 3z'_1 & = -1 \end{cases}$$

et les z_i, z'_i, y_i *positifs.*

On commence par résoudre :

Minimiser $t_1 + t_2$ *avec*

$$\begin{cases} t_1 + y_1 - 4z_0 + 4z_0' + 6z_1 - 6z_1' &= 6 \\ t_2 - y_2 + 4z_0 + 4z_0' + 3z_1 - 3z_1' &= 1 \end{cases}$$

et z_i, z_i', y_i, t_i *positifs.*

Le tableau est alors

$$\begin{bmatrix} 0 & 0 & 1 & -1 & -8 & 8 & 9 & -9 & 7 \\ 1 & 0 & 1 & 0 & -4 & 4 & 6 & -6 & 6 \\ 0 & 1 & 0 & -1 & -4 & 4 & 3 & -3 & 1 \end{bmatrix}.$$

Après application du simplexe, on obtient :

$$\begin{bmatrix} -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & \boxed{0} \\ 1 & -1 & 1 & 1 & 0 & 0 & 3 & -3 & 5 \\ 0 & 1 & 0 & -1/4 & -1 & 1 & 3/4 & -3/4 & 1/4 \end{bmatrix}.$$

On applique alors le simplexe avec le tableau :

$$\begin{bmatrix} 0 & -1 & 0 & 0 & 1 & -1 & 2 \\ 1 & 1 & 0 & 0 & 3 & -3 & 5 \\ 0 & -1/4 & -1 & 1 & 3/4 & -3/4 & 1/4 \end{bmatrix}$$

et on obtient :

$$\begin{bmatrix} -1/3 & -4/3 & 0 & 0 & 0 & 0 & 1 \,/\, 3 \\ 1/4 & 1/2 & 1 & -1 & 0 & 0 & 1 \\ 1/3 & 1/3 & 0 & 0 & 1 & -1 & 5/3 \end{bmatrix}.$$

Grammaire proposée

$$\texttt{prog} ::= \begin{cases} \textbf{int } x_0, \ldots, x_n; \\ \texttt{block}; \end{cases}$$

$$\texttt{block} ::= \begin{cases} \langle \texttt{inv}_0 \rangle \, \texttt{instr}_0; \\ \ldots \\ \langle \texttt{inv}_n \rangle \, \texttt{instr}_n; \langle \texttt{inv}_{n+1} \rangle \end{cases}$$

$$\begin{aligned} \texttt{instr} ::= {} & x_j = \texttt{expr} \\ & \mid \textbf{if } (\texttt{expr} \geq 0) \, \texttt{block}_0 \textbf{ else } \texttt{block}_1 \\ & \mid \textbf{while } (\texttt{expr} \geq 0) \, \texttt{block} \end{aligned}$$

$$\texttt{inv} ::= \texttt{expr}_0 \geq 0 \wedge \cdots \wedge \texttt{expr}_n \geq 0$$

$$\texttt{expr} ::= c + \sum_{i=0}^{n} \alpha_i \star x_i$$

Limites de la grammaire proposée

- Un seul test de comparaison $\geq$: $a = b$ s'écrit $a \geq b \wedge b \geq a$.
- Pour les extensions :
  - Pas de construction `unsat` ;
  - Obligation d'écrire des invariants vides explicitement ;
  - Pas de connecteur $\vee$, pourtant nécessaire pour la génération automatique des invariants en sortie d'un **if**.
- Pas consistant avec l'exemple du sujet qui utilise $<$ et $=$.

$\Rightarrow$ D'où le choix de définir une grammaire plus générale.

Grammaire adoptée

$$\texttt{prog} ::= \begin{cases} \textbf{int } x_0, \ldots, x_n; \\ \texttt{block}; \end{cases}$$

$$\texttt{block} ::= \begin{cases} [\langle \texttt{xinv}_0 \rangle]\texttt{instr}_0; \ldots \\ [\langle \texttt{xinv}_n \rangle]\texttt{instr}_n; [\langle \texttt{xinv}_{n+1} \rangle] \end{cases}$$

$$\begin{aligned} \texttt{instr} ::= {} & x_j = \texttt{expr} \\ & \mid \textbf{if}\,(\texttt{inv})\,\texttt{block}_0\ \textbf{else}\ \texttt{block}_1 \\ & \mid \textbf{while}\,(\texttt{inv})\,\texttt{block} \end{aligned}$$

$$\texttt{xinv} ::= \texttt{inv} \mid \texttt{unsat}$$

$$\begin{aligned} \texttt{inv} ::= {} & \texttt{expr}_0 \{\geq, \leq, >, <, =\} \texttt{expr}_1 \\ & \mid \texttt{inv}_0\, \{\vee, \wedge\}\, \texttt{inv}_1 \\ & \mid \neg \texttt{inv} \\ & \mid (\texttt{inv}) \end{aligned}$$

$$\texttt{expr} ::= c + \sum_{i=0}^{n} \alpha_i \star x_i$$

Implémentation

- `ocamllex` pour l'analyse lexicale (`.mll` → `.ml`).
- `ocamlyacc` pour l'analyse syntaxique (`.mly` → `.ml`).
- On conserve les numéros de ligne pour le débogage.
- Variables représentées par leur nom : `type pvar = string`.
- Invariants représentés sous une forme abstraite :

```
type pinv = Naught of int | PUnsat of int
| Expr of int * pineq | Not of int * pinv
| And of int * pinv list | Or of int * pinv list .
```

Production de code intermédiaire

- Une fois qu'on a lu le code et qu'on l'a transformé en arbre syntaxique, on veut le mettre dans une forme qui nous permette de le vérifier plus facilement.
- Une fonction de conversion par structure à convertir.
- On produit du code intermédiaire et (extension) on en profite pour compléter les invariants laissés vides.
- Variables représentées par un entier : `type var = int`.
- On conserve toujours les numéros de ligne.
- Donc invariants sous forme normale disjonctive :
  `type inv = int * (Fraction.frac array) list list`.

Complétion des invariants

On utilise les règles suivantes :

$$\langle \mathtt{inv}\,[x_j \leftarrow \mathtt{expr}]\rangle\; x_j = \mathtt{expr}\;\langle \mathtt{inv}\rangle$$
$$\langle \mathtt{inv}_0\rangle\; \textbf{if}\,(\mathtt{inv}_1)\,\mathtt{block}_0\; \textbf{else}\; \mathtt{block}_1\; \langle \ell\,(\mathtt{block}_0) \vee \ell\,(\mathtt{block}_1)\rangle$$
$$\langle \mathtt{inv}_0\rangle\; \textbf{while}\,(\mathtt{inv}_1)\,\mathtt{block}_0\; \langle \neg \mathtt{inv}_1 \wedge (\ell\,(\mathtt{block}) \vee \ell\,(\mathtt{block}_1))\rangle$$

où ℓ renvoie le dernier invariant d'un bloc.
Dans

$$\langle \mathtt{inv}_0\rangle\; \textbf{if}\,(\mathtt{inv}_1)\,\mathtt{block}_0\; \textbf{else}\; \mathtt{block}_1$$

on complète le premier invariant du **if** par $\mathtt{inv}_0 \wedge \mathtt{inv}_1$ et le permier invariant du **else** par $\mathtt{inv}_0 \wedge \neg \mathtt{inv}_1$.
Un `unsat` se propage dans un bloc et tous ses sous-blocs.

Simplification des invariants

Au fur et à mesure de leur complétion, les invariants sont simplifiés. On utilise l'algorithme du simplexe et les formules

$$(e_0 \wedge e_1 \wedge \cdots \wedge e_n \Leftrightarrow e_1 \wedge \cdots \wedge e_n) \Leftrightarrow (e_1 \wedge \cdots \wedge e_n \Rightarrow e_0)$$

et

$$(c_0 \vee c_1 \vee \cdots \vee c_n \Leftrightarrow c_1 \vee \cdots \vee c_n) \Leftrightarrow (c_0 \Rightarrow c_1 \vee \cdots \vee c_n) \Leftrightarrow \bigvee_{i=1}^{n} (c_0 \Rightarrow c_i)$$

qui nous donnent deux règles d'élimination d'une inégalité (ici e_0) ou d'une clause conjonctive (ici c_0).

Vérification des invariants

On applique l'algorithme proposé dans le sujet, en s'appuyant sur l'algorithme du simplexe et celui de Fou~~r~~rier-Motzkin.
Pour raccourcir les fonctions :

- `extended_verify_inv: xinv -> xinv -> bool;`
- `verify_inv: inv -> inv -> bool;`
- `verify_expr: expr list -> expr -> bool.`

Le cas de `unsat` est traité à part grâce à :

- `sat_inv: inv -> bool;`
- `sat_conj: expr list -> bool.`

Un peu de stylistique pour terminer

Les inégalités étant stockées sous forme de tableaux de fractions, le nombre de variables (longueur du tableau) est une donnée importante.
En style fonctionnel pur sans modules ni objets, `var_count` doit être passé à chaque fonction.
Solution :
`let verifier (var_count : int) = object(s) ... end.`
Dans `linlang.ml`, le typage des fonctions est explicité afin de faciliter le débogage.