Programmes linéaires et principe de transposition

Éric Schost

Eric.Schost@polytechnique.fr

Le principe de transposition est un ensemble de *règles de transformation* pour les programmes calculant des applications linéaires. Si P est un programme qui calcule l'application $x \mapsto Ax$, où A est une matrice $n \times m$, alors ces règles de transformation permettent d'obtenir un programme P^t qui calcule l'application $y \mapsto A^t y$, où A^t est la matrice transposée de A. En outre, le nombre d'instructions du programme modifié est essentiellement égal au nombre d'instructions du programme initial.

Quelle est la motivation ? Si A est une matrice complètement quelconque, il faut mn opérations pour la multiplier par un vecteur, que ce soit à droite ou à gauche, donc on ne va pas gagner grand-chose. Par contre, si A est structurée, il est concevable que l'on dispose d'un algorithme plus rapide que la version naïve pour la multiplier par un vecteur. En utilisant le principe de transposition, on obtient aussitôt un algorithme de complexité analogue pour multiplier A^t par un vecteur, sans se fatiguer.

Le but de ce projet est d'implanter ce principe dans un modèle particulier, celui des programmes linéaires sur des machines à allocation de registres. Pour un présentation (plus formelle) du principe de transposition, on renvoie à [1], dont on fournira des photocopies sur demande. On fait plus bas référence à un rapport de recherche INRIA, qui sera lui aussi fourni.

1 Machines à registres et programmes linéaires

Commençons par définir les machines RAM à instructions linéaires. On fixe un entier M ; il représente la mémoire disponible, de sorte que l'on peut accéder à M registres $R_1, \ldots, R_M$. On y stocke des *nombres*, c'est-à-dire des éléments d'un corps K (un anneau ferait tout aussi bien l'affaire).

Un *programme linéaire* est la donnée des objets suivants :

- un sous-ensemble $R_{i_1}, \ldots, R_{i_m}$ des registres que l'on appelle *entrée* ;
- un sous-ensemble $R_{o_1}, \ldots, R_{o_n}$ des registres que l'on appelle *sortie* ;
- une suite d'*instructions* portant sur ces registres. Les instructions sont choisies parmi :
  - $R_i = \pm R_j \pm R_k$, avec $1 \le i, j, k \le M$,
  - $R_i = \lambda R_j$, avec $1 \le i, j \le M$ et λ dans K.

  Noter qu'il n'y a ni boucle, ni test.

Le fonctionnement d'un tel programme est le suivant : à l'initialisation, les registres d'entrée reçoivent des valeurs $x_1, \ldots, x_m$ dans K et les autres sont mis à zéro. On effectue ensuite toutes les instructions, puis le programme renvoie les valeurs des registres de sortie. Noter que l'on calcule bel et bien une fonction linéaire des x_i.

La première partie du projet demande d'implanter une machine à registres virtuelle. Cette machine lira des fichiers contenant des programmes linéaires, et effectuera les calculs correspondants. On est libre de choisir pour K le corps des rationnels, un corps fini $\mathbb{Z}/p\mathbb{Z}$, ... Bien évidemment, le mieux sera d'écrire du code *polymorphe*, qui permettra d'utiliser au choix l'un des domaines ci-dessus.

2 Transposition

Venons-en au principe de transposition. La deuxième partie du projet consiste à implanter les deux versions du principe de transposition présentées ci-dessous.

2.1 Cas d'un jeu d'instructions réduit

Pour commencer, on traite le cas d'une machine avec un jeu d'instructions limité par rapport au cas général. On ne s'autorise que les instructions du type :

- $R_i = R_i \pm R_j$, avec $1 \leq i, j \leq M$;
- $R_i = \lambda R_i$, avec $1 \leq i \leq M$, $\lambda \in K$.

Pour "transposer" un programme comportant des instructions de ce type, on interprète chacune de ces intructions comme une application linéaire $K^M \to K^M$.

Pour motiver ce procédé, considérons un exemple. Avec $M = 3$, l'instruction $R_1 = R_1 + R_3$ s'interprète comme l'application linéaire dont la matrice est

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} :$$

R_2 et R_3 n'ont pas changé et R_1 devient $R_1 + R_3$. La transposée de cette application linéaire a pour matrice

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} ;$$

on en déduit que l'on peut la traduire par l'instruction $R_3 = R_3 + R_1$.

De manière générale, la transposée de l'instruction $R_i = R_i + R_j$ est l'instruction $R_j = R_j + R_i$. Par écriture matricielle, on voit que l'instruction $R_i = \lambda R_i$ est inchangée par transposition, et que $R_i = R_i - R_j$ se transpose en $R_j = R_j - R_i$.

On peut alors définir le programme transposé P^t d'un programme P :

- les entrées de P^t sont les sorties de P ;
- les sorties de P^t sont les entrées de P ;
- les instructions de P^t sont les transposées des instructions de P, *prises en sens inverse.*

La dernière condition, le retournement de la liste des instructions, traduit l'égalité $(AB)^t = B^t A^t$, pour des matrices A, B. Cette définition montre que si P calcule une application linéaire $K^m \to K^n$, alors P^t calcule bien l'application transposée $K^n \to K^m$.

Exemple. Considérons le programme

```
R4:=R4+R2;
R3:=3*R3;
R4:=R4+R3;
R2:=2*R2;
R3:=R3+R2;
R3:=R3+R1;
```

dont les entrées sont R_1, R_2, R_3 et les sorties R_3, R_4. Ce programme calcule l'application linéaire dont la matrice est

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 3 \end{bmatrix}.$$

Le programme transposé s'écrit

```
R1:=R1+R3;
R2:=R2+R3;
R2:=2*R2;
R3:=R3+R4;
R3:=3*R3;
R2:=R2+R4;
```

et on vérifie qu'il calcule l'application linéaire dont la matrice est

$$\begin{bmatrix} 1 & 0 \\ 2 & 1 \\ 3 & 3 \end{bmatrix}.$$

2.2 Cas général

Le cas du jeu d'instructions général se ramène au cas du jeu d'instructions réduit de manière immédiate. Ainsi, l'instruction $R_1 = R_2 + R_3$ est équivalente à la suite d'instructions

```
R1=0
R1=R1+R2
R1=R1+R3
```

Il est donc possible de le transposer sous la forme

```
R3=R3+R1
R2=R2+R1
R1=0
```

De même, on réécrit aisément l'instruction $R_i = \lambda R_j$ en utilisant le jeu d'instructions réduit, ce qui permet de la transposer, ...

3 Optimisations

Notre principe de transposition n'est pas complètement satisfaisant : au vu des règles de réécriture ci-dessus, il semble que l'on puisse perdre jusqu'à un facteur 3 (en termes de nombre de lignes). Dans cette partie, on demande d'optimiser le code produit, afin de regagner tant que possible les lignes perdues. On utilisera pour ce faire les règles suivantes.

Suppression des zéros. La transformation pour passer du cas général au jeu d'instructions réduit introduit des lignes du type $R_i = 0$. On peut supprimer une telle ligne, à condition de réécrire les instructions suivantes en conséquence.

Suppressions des recopies. Après avoir supprimé les zéros, il se peut qu'il reste des lignes du type $R_i = \pm R_j$. On peut également supprimer ce type de ligne, à condition de réécrire de manière judicieuse les instructions qui suivent.

On peut en fait montrer qu'il est possible d'obtenir un code transposé qui fait exactement le même nombre de lignes que l'original, si par exemple l'application linéaire que l'on calcule est inversible.

4 Exemple : le produit transposé

Pour finir, on propose d'implanter un exemple, relatif au calcul avec des polynômes en une variable. Cet exemple est tiré de [2], qui en décrit largement la motivation et les applications.

Fixons une fois pour toutes un polynôme f dans $K[X]$, de degré n. Pour m quelconque, on peut considérer l'application de *multiplication par* f, qui à un polynôme g de degré inférieur à m associe le produit fg, de degré inférieur à $m+n$. Il s'agit bien d'une application linéaire de K^m dans K^{m+n}.

L'opération transposée porte le nom de *produit médian* ; savez-vous dire pourquoi ? Indice : regarder l'écriture matricielle de ces opérations.

Dans cette partie, on demande d'écrire une fonction qui prend en entrée le polynôme f, un entier m et qui produit un programme linéaire calculant l'application de multiplication par f décrite ci-dessus. Transposez alors le code obtenu, et vérifiez que le produit médian porte bien son nom.

Question subsidiaire. Par défaut, on propose d'implanter un algorithme "naïf" de multiplication des polynômes, de complexité quadratique. Il est cependant possible de faire beaucoup mieux : les méthodes à base de FFT descendent quasiment jusqu'à une complexité linéaire. La FFT est bien trop difficile à implanter ; une solution intermédiaire est l'algorithme de Karatsuba (décrit dans [1]). Il sera très instructif d'implanter cette approche.

References

[1] P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic Complexity Theory.* Springer, 1997.

[2] G. Hanrot, M. Quercia, and P. Zimmermann. The middle product algorithm , I. Speeding up the division and square root of power series. Technical report, RR INRIA 4664, 2002.