Travaux Pratiques URM

Pierre KOEBELIN

30 avril 2018

Résumé

L'ensemble du code du projet est disponible sur GitHub à l'adresse `https://github.com/Derriick/Unlimited-Register-Machine`.
L'objectif de ce projet est l'implémentation d'un simulateur pour les URMs. Une machine à registres exécute des instructions, assigne des registres et lit un programme. Les instructions sont représentées par le type :

```
type instruction =
| Reset of int
| Incr of int
| Set of int*int
| Jump of int*int*int
```

Les registres sont stockés dans des tableaux que nous initialisons à une taille fixe `max_registers`. Comme dans le cours, nous utilisons le registre 0 pour désigner l'instruction en cours. Pour chaque programme, le résultat après son exécution est stocké dans le registre 1 par défaut. Il reste cependant possible de récupérer des résultats dans d'autres registres. Un programme est alors un tableau d'instructions.
Quelques outils ont également été implémentés pour manipuler des URM.

Table des matières

1 Exercices

Code L'ensemble de l'implémentation concernant les exercices du TP noté est disponible dans le fichier `src/mgr.ml`. Il se sert du code réalisé lors du 1^{er} TP, situé dans le fichier `src/urm.ml` ; il constitue la base d'une *Unlimited Register Machine*.

1.1 Exercice 1

Chaque programme est représenté sous la forme d'un tableau d'instructions. Il en est de même pour l'ensemble des exercices. Ces programmes serviront ensuite à réaliser l'ensemble des tests.

1.1.1 `succ`

Le but de ce programme d'incrémenter la valeur se trouvant dans le 1^{er} registre. Pour cela, j'utilise simplement l'instruction `Incr(1)`.

```
0 Incr(1)
```

1.1.2 `sum`

Le but de ce programme de calculer la somme des valeurs contenues dans les 1^{er} et 2^{eme} registres. Pour cela, j'utilise la suite d'instructions suivante :

```
0 Reset(3)
1 Jump(2,3,5)
2 Incr(1)
3 Incr(3)
4 Jump(0,0,1)
```

1.1.3 `constant`

Ce programme ne doit rien faire par définition. Cependant, l'implémentation de l'URM m'empêche d'utiliser un programme vide. Je copie dont le contenu du 1^{er} registre dans lui-même avec l'instruction `Set(1, 1)`.

```
0 Set(1,1)
```

1.1.4 `bigger`

L'objectif de ce programme est de renvoyer `1` si la valeur contenue dans le 1^{er} registre est strictement supérieure à celle du 2^{nd}, et `0` sinon. J'utilise donc la suite d'instructions suivantes :

```
0 Set(1,3)
1 Set(2,4)
2 Jump(1,4,10)
3 Jump(2,3,7)
4 Incr(1)
5 Incr(2)
6 Jump(0,0,2)
7 Reset(1)
8 Incr(1)
9 Jump(0,0,11)
10 Reset(1)
```

1.2 Exercice 2

Les fonctions `string_of_prog` et `debut_program` sont utilisées lors des tests afin d'avoir un rapide aperçu du programme exécuté, ainsi que des registres utilisés avant et après l'exécution.

1.2.1 `string_of_prog`

Cette fonction prend en entrée un programme dont elle affiche tout simplement le code. Pour cela, on parcours l'ensemble du programme, en écrivant chaque instruction précédée de son index.

1.2.2 `debut_program`

Celle-ci quant à elle affiche les valeurs dans les registres, exécute le programme passé en paramètre, puis réaffiche leur contenu. Pour ce faire, elle prend en entrée un tableau de registres, un programme et renvoie la valeur contenue dans le 1^{er} registre après l'exécution.

1.3 Exercice 3 : `compose1`

La fonction `compose1` prend en entrée 2 programmes et renvoie un seul étant la concaténation des 2. Cependant, les instructions `Jump` du 2^{nd} programme sont modifiées afin de pointer vers la bonne instruction. J'utilise donc la fonction `Array.map` pour parcourir l'ensemble du programme, et modifier chaque instruction `Jump` en ajoutant à son index la longueur du $1er$ programme.

1.4 Exercice 4 : `translate`

Cette fonction permet de transformer un programme utilisant normalement avec des registres entre `1` et `n` pour qu'il puisse prendre en paramètre registres de a_1 à a_n, puis de stocker le résultat dans un registre `k`. Cela permet de ne pas modifier les registres de a_1 à a_n, pour qu'ils puissent être réutilisés par d'autre programmes. Elle renvoie ainsi un programme réalisant l'ensemble de ces actions.

1.5 Exercice 5 : `compose2`

Cette fonction permet d'exécuter chaque programme G_i d'un vecteur `G` devant chaque utiliser les mêmes valeurs dans les registres, puis un programme `F` utilisant le résultat de chaque G_i. Pour ce faire, j'utilise la fonction `translate` qui me permet d'exécuter chaque programme de `G` avec à chaque fois les mêmes valeurs en entrée et de stocker les résultats à un endroit du banc de registres ne rentrant pas en conflit lors de l'exécution, ensuite lu par le programme `F`.

1.6 Exercice 6 : `expr_to_prog`

Une expression est du type :

```
type expression =
    | S of expression * expression
    | I of int
```

À partir des programmes `sum` et `constant`, la fonction `expr_to_prog` produit un programme correspondant à l'expression en paramètre. J'utilise la fonction `translate` afin de réaliser l'ensemble des opérations de l'expression pour ne pas modifier le contenu des registre en entrée et placer le résultat de chaque `sum` ou `constant` au bon endroit pour passer à la suite.

1.7 Exercice 7 : `if_then_else`

En fonction du résultat d'un programme `progC` (`C` pour condition), on veut exécuter le code de `progT` (`T` pour `true`) ou `progF` (`F` pour `false`). On utilise alors la fonction `if_then_else` dans laquelle j'exécute `progC`. Si le résultat est nul, j'exécute `progF`, et `progT` dans le cas contraire. Pour créer un tel programme, cette fonction prend en paramètres les 3 programmes, et en fait une composition en ajouter des instructions `Jump` entre chacun pour exécuter le code qui nous intéresse. J'attribue la valeur `0` grâce à l'instruction `Reset` pour faire la comparaison. J'emploie également la fonction `translate` pour que chaque programme est accès aux valeurs dont il a besoin.

2 Tests

L'ensemble des codes de test sont disponibles dans le fichier `src/app.ml` du projet. Il utilise les fonctions de `src/exec.ml` afin d'exécuter les programmes et afficher leur résultat.

2.1 Exercice 1

On exécute tout simplement les programmes `succ`, `sum`, `constant` et `bigger`.

2.2 Exercice 2

On exécute des programmes composés des 4 fonctions de l'exercice 1. On vérifie par exemple la transitivité de la fonction `compose1` (3^{eme} et 4^{eme} tests), ainsi que la combinaison renvoie le bon résultat.

2.3 Exercice 3

Les fonctions `string_of_prog` et `debug_program` sont utilisées pour chaque test des autres exercices.

2.4 Exercices 4 et 5

On vérifie la fonction `compose2` à partir des programmes de l'exercice 1, ainsi qu'avec des compositions de ceux-ci, obtenus à l'aide ce `compose1`.

2.5 Exercice 6

On crée quelques expressions dont vérifie ensuite le résultat après être passées par la fonction `expr_to_prog`. On change également l'ordre des registres dans l'expression afin de s'assurer que cela n'influence pas le résultat final.

2.6 Exercice 7

On teste quelques cas en sachant à l'avance quel programme entre `progT` et `progF` doit être exécuté, ainsi que son résultat. On utilise ensuite le programme `bigger` en tant que `progC`.