Rapport du TP de CAL

Adrien Blassiau

19 mai 2019

1 Introduction

Ce petit projet contient 5 fichiers : `urm.ml` et `urm.mli` qui sont les fichiers avec les fonctions associées aux différentes questions du projet, `test.ml` qui est le fichier comportant les tests unitaires, `Makefile`, le Makefile du projet et enfin `README.md` un README avec des instructions sur le projet.
On rappelle que pour lancer les tests unitaires, il suffit de taper la commande suivante :

```
> make clean && make all && ./run
```

Toutes les fonction du projet ont été testées dans le fichier `test.ml`. Si le test est un succès, on affiche OK à côté du nom du test. Sinon, un message d'erreur apparaît.
Le projet est aussi disponible sur mon github, ici.

2 Présentation de l'implémentation

2.1 Programme de base

Pour tester nos différents fonctions de manipulation d'URM, il nous faut quelques programmes de base. Ceux demandés dans la question 1 du projet ont été implémentés (en rouge), et j'en ai rajouté quelques autres pour étendre les tests. Dans le tableau suivant, je détaille les différents programmes construits et je donne leur ρ, c'est à dire leur plus grand registre utilisé (j'utilise la définition de ρ du cours et du TP1 ...).

	description	ρ
prog_succ	Renvoie le successeur du premier registre	0
prog_somme_2_args	Renvoie la somme des deux premiers registres	2
prog_somme_3_args	Renvoie la somme des trois premiers registres	4
prog_somme_modify	Renvoie la somme des deux premiers registres (utilisé pour tester normalize)	2
prog_constant	Renvoie l'entier passé en paramètre (obsolète)	1
prog_constant_v2	Renvoie l'entier passé en paramètre	0
prog_bigger	Renvoie 1 si le premier registre est plus grand que le deuxième, 0 sinon	3
prog_moins_1	Renvoie le contenu du premier registre - 1	2
prog_division_2	Renvoie le contenu du premier registre divisé par deux	2
prog_test_0	Renvoie O si le premier registre contient 0, 1 sinon	1
prog_somme_dizaine_unite	Renvoie la somme des chiffres des dizaine et unité	4

2.2 Fonction d'affichage et débug

La fonction `string_of_prog` est utilisée pour afficher une chaîne de caractères lisible d'un programme. Elle est ausi utilisée dans les tests afin d'observer que les méthodes ont bien été respectées, notamment pour la composition totale. On reviendra sur un exemple détaillé utilisant cette fonction plus tard dans ce rapport.
Notre fonction `debug_program` utilise la fonction `print_registers` qui affiche les registres utilisés avant et après l'exécution du programme P.

2.3 Concaténation de programme

Une fonction de concaténation a été implémentée. Elle se nomme `compose`. Son fonctionnement est simple, il faut juste bien penser à décaler les indices des instructions de saut comme expliqué dans l'énoncé. Par exemple, voici une utilisation de `compose` :

```
let prog = compose prog_succ prog_moins_1 in run_function prog [4] false = 4
```

On utilise ici la fonction `compose` sur le programme qui donne le successeur et celui qui donne le prédécesseur. Au final, on arrive avec une concaténation de deux programmes, ce qui revient à rajouter puis enlever 1 à l'argument d'entrée, d'où le résultat 4. Le programme correspondant à `compose prog_succ prog_moins_1` a donc cette forme :

0 : Incr 0
1 : Jump 0, 1, 6
2 : Incr 1
3 : Jump 0, 1, 6
4 : Incr 2
5 : Jump 0, 0, 2
6 : Set 2, 0

En effet, sachant :

```
let prog_succ = [| Incr 0 |]

let prog_moins_1 =
  [| Jump (0, 1, 5);
     Incr 1;
     Jump (0, 1, 5);
     Incr 2;
     Jump (0, 0, 1);
     Set (2, 0) |]
```

On voit bien que la concaténation entraîne un décalage de 1 dans les indices de saut des différents Jump de `prog_moins_1`, d'où le résultat obtenu (en rouge les indices modifiés).

2.4 Construction d'un programme translaté

La construction d'un programme translaté suit une procédure très particulière, expliquée dans l'énoncé et qui a été implémentée sous la fonction `translate`. Prenons une translation de `prog_somme_dizaine_unite` obtenu via une translation par la fonction `translate` afin d'illustrer ce mécanisme :

```
1 let prog = translate prog_somme_dizaine_unite [7;8;9] 2 in string_of_prog prog;
```

On obtient le résultat suivant, avec les éléments modifiés ou ajoutés qui ont été coloriés :

0 : Set 7, 0
1 : Set 8, 1
2 : Set 9, 2
3 : Reset 3
4 : Reset 4
5 : Incr 4
6 : Incr 4
7 : Incr 4
8 : Incr 4
9 : Incr 4
10 : Incr 4
11 : Incr 4
12 : Incr 4
13 : Incr 4
14 : Incr 4
15 : Jump 0, 1, 24
16 : Incr 1
17 : Incr 2
18 : Jump 2, 4, 21
19 : Jump 0, 1, 24
20 : Jump 0, 0, 15
21 : Reset 2
22 : Incr 3
23 : Jump 0, 0, 15
24 : Reset 0
25 : Jump 0, 3, 30
26 : Incr 0
27 : Incr 2
28 : Jump 0, 3, 30
29 : Jump 0, 0, 26
30 : Set 2, 0
31 : Set 0, 2

— La première règle indique de transférer le contenu des registres $a_1, ... a_n$ dans les registres 1 à n. Ici, on veut donc déplacer le contenu des registres 7,8 et 9 dans les registres 0, 1 et 2. C'est ce que l'on fait lignes 1, 2 et 3.

— La deuxième règle demande d'initialiser à 0 les registres de n+1 à $\rho(P)$, où P est le programme. On sait que $\rho(prog\_somme\_dizaine\_unite) = 4$ et que $n = 2$. On obtient donc les lignes 3 et 4 du programme.

— La troisième règle correspond juste à un appel de `translate_jump`, c'est-à-dire un décalage des indices de saut des différents Jump. On les a marqué en vert.

— La dernière étape demande de transférer le contenu du premier registre, donc du résultat du programme, dans le registre passé en paramètre. Ici, on passe 2 en paramètre donc on obtient logiquement la ligne 31 de notre programme.

2.5 Construction d'une composition de programme

La construction d'un programme suivant une composition générale suit une procédure très particulière, expliquée dans l'énoncé et qui a été implémentée sous la fonction `general_compose`. Prenons la composition suivante :

```
let new_prog = general_compose prog_somme_2_args [prog_somme_2_args;prog_bigger] 2 in
run_function new_prog [5;2] false = 8
```

On obtient le résultat suivant, avec les lignes d'intérêts en couleur :

0 : Set 0, 4
1 : Set 1, 5
2 : Set 4, 0
3 : Set 5, 1
4 : Reset 2
5 : Jump 1, 2, 10
6 : Incr 0
7 : Incr 2
8 : Jump 1, 2, 10
9 : Jump 1, 1, 6
10 : Set 0, 6
11 : Set 4, 0
12 : Set 5, 1
13 : Reset 2
14 : Reset 3
15 : Jump 1, 3, 22
16 : Jump 0, 2, 25
17 : Incr 2
18 : Incr 3
19 : Jump 1, 3, 22
20 : Jump 0, 2, 25
21 : Jump 1, 1, 17
22 : Reset 0
23 : Incr 0
24 : Jump 1, 1, 26
25 : Reset 0
26 : Set 0, 7
27 : Set 6, 0
28 : Set 7, 1
29 : Reset 2
30 : Jump 1, 2, 35
31 : Incr 0
32 : Incr 2
33 : Jump 1, 2, 35
34 : Jump 1, 1, 31
35 : Set 0, 0

— la première règle indique de calculer le registre **N libre** avec n le nombre de programmes, k le nombre d'arguments, $\rho(\mathrm{F})$ le dernier indice occupé par la fonction qui compose et les $\rho(G_i)$ les derniers indices occupés par les fonctions arguments. On a N = max(n,k,ρ(F),max($\rho(G_i)$)). Sachant que n = 2, k = 2, `F=prog_somme_2_args` donc $\rho(\mathrm{F})$=2 et max($\rho(G_i)$)=max(2,3)=3, on trouve **N=3**.

— la deuxième règle indique de sauvegarder les données se trouvant dans les registres 1 à k dans les registres N+1 à N+k. Ici, on commence au registre 0, donc on sauvegarde les registres 0 à k-1 dans les registres N+1 à N+k. Finalement, on sauvegarde les registres 0 à 1 dans les registres 4 à 5, ce que l'on retrouve bien lignes 0 et 1.

— la troisième règle correspond à la translation de chaque programme argument, donc à un appel à la fonction de translation avec pour arguments le $i^{\text{ème}}$ programme, la position des registres de N+1 à N+k et le registre de sortie en N+k+i, où i est l'indice du $i^{\text{ème}}$ programme.
Pour le premier programme argument `prog_somme_2_args`, on a les registres d'entrée qui sont 4 et 5 (car N=3 et k=2) et le registre de sortie qui est 6 (car N=3, k=2 et i=1). On obtient donc le programme en bleu.
Même principe pour le programme `prog_bigger`, en vert : on a les registres d'entrée qui sont 4 et 5 (car N=3 et k=2) et le registre de sortie qui est 7 (car N=3, k=2 et i=2).

— la quatrième règle correspond là encore à une translation, donc à un appel à la fonction de translation avec pour arguments la position des registres de N+k+1 à N+k+n et le registre de sortie en 0. On obtient donc le programme en orange.

2.6 Construction d'un petit langage d'expressions et utilisation d'un embranchement conditionnel

On construit notre petit langage d'expression enrichi de cette manière, à partir d'un type somme :

```
type expression =
  | Entier of int
  | Somme of expression * expression
  | If_then_else of expression * expression * expression
```

Voyons un exemple de l'utilisation de prog_of_expr combiné avec le langage d'expression et notamment if_then_else :

```
let prog_expr_test =
prog_of_expr (If_then_else (Entier 1,
                            Somme (Entier 11, (If_then_else ((Entier 0),
                                                             (Entier 11),
                                                             (Entier 42)))),
                            Somme (Entier 1,Entier 2))) in
debug_program prog_expr_test [] true = 53
```

On rappelle que le test est vrai si le résultat testé est non nul. Ainsi, on a l'exécution suivante (en rouge) :

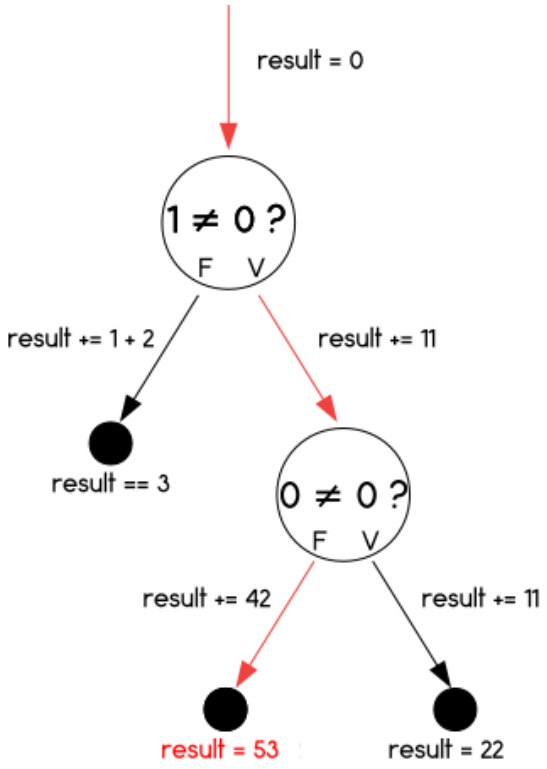

FIGURE 1 – Chemin d'exécution du programme

On obtient bien le résultat voulu. Les autres chemins conditionnels sont testés dans le fichier de tests.