Machine à registres non limités

Projet CAML

L3 Informatique 2017–2018

Samuele Giraudo, Carine Pivoteau et Stéphane Vialette

Problématique. Le but de ce projet est de programmer un simulateur pour une machine à registres non limités et de compiler des programmes écrits sur un jeux d'instructions étendus pour pouvoir les exécuter sur ces machines. Des explications supplémentaires ainsi que des programmes de test sont disponibles sur la page `https://tinyurl.com/yd7nkmu4` consacrée au projet.

1 Première partie

1.1 Jeu d'instructions URM

1.1.1 Machines à registres non limités

Nous allons formaliser une notion d'ordinateur très simple, une *machine à registres non limités*[1] (ou « URM » pour « Unlimited Register Machine »). Il s'agit d'un modèle simplifié et abstrait du fonctionnement des appareils mécaniques de calcul.

Ces machines ont une mémoire constituée d'un nombre fini de registres $R_0, \ldots, R_k$, chaque registre pouvant contenir un entier naturel. S'il n'y a pas de limite supérieure aux nombres de registres qu'une URM peut utiliser, la valeur d'un registre non initialisé est indéfinie (elle peut donc varier d'une implémentation à une autre). Une URM dispose également d'un *programme* qui est une suite finie constituée de quater types de *commandes*, et d'un *pointeur d'instruction* qui indique la prochaine commande du programme à exécuter. Une URM s'arrête uniquement si le pointeur d'instruction ne pointe pas vers une commande du programme.

1.1.2 Un peu de vocabulaire

La *longueur* d'un programme est le nombre d'instruction URM qui le composent. L'*état* (ou « *state* ») d'une URM à un instant donné consiste en la valeur du pointeur d'instruction et les valeurs des registres utilisés par le programme. L'*avancement du*

1. Consulter `https://tinyurl.com/y773jg3x` ou `https://tinyurl.com/y9dxon7b` pour plus de détails.

calcul (ou « *stage of computation* »ou plus simplement « *stage* ») est le nombre d'instructions qui ont été exécutées depuis le lancement du programme. Le *résultat* (ou « *output* ») d'un programme est la valeur des registres après exécution du programme. Le *nombre de registres* d'un programme est le nombre de registres manipulé dans le programme. Si P est un programme pour une URM, il est d'usage de noter $\rho(P)$ l'index maximum d'un registre utilisé par le programme P (et donc $\rho(P) = 0$ si le programme ne contient aucune commande).

1.1.3 Commandes autorisées

Les commandes valides qui peuvent apparaître dans un programme d'une URM sont les suivantes :

— `COPY` i j

Copie la valeur du registre R_j dans le registre R_i (R_j conserve sa valeur initiale). Incrémente ensuite (de 1) le pointeur d'instruction.

Le programme s'arrête en erreur si $i = j$ ou si le registre R_j n'a pas été initialisé.

— `JUMP` i j ℓ

Si les valeurs des registres R_i et R_j sont égales, ceci place le pointeur d'instruction sur la ligne numéro ℓ (la première ligne du programme est la ligne numéro 0) ; sinon, ceci incrémente (de 1) le pointeur d'instruction.

Le cas $i = j$ est valide. Le programme s'arrête en erreur si le registre R_i n'a pas été initialisé ou si le registre R_j n'a pas été initialisé.

— `SUCC` i

Incrémente (de 1) le contenu du registre R_i et incrémenter ensuite (de 1) le pointeur d'instruction.

Le programme s'arrête en erreur si le registre R_i n'a pas été initialisé.

— `ZERO` i

Écrit la valeur 0 dans le registre R_i. Incrémente ensuite (de 1) le pointeur d'instruction.

Aucune commande URM n'est sensible à la casse ; les commandes `COPY`, `copy` et `CoPy` sont donc équivalentes. Dans tous les cas (sauf — éventuellement — pour la commande `JUMP` i j ℓ), la prochaine commande à exécuter est la suivante (incrémentation de 1 implicite du pointeur d'instruction). On considère toujours que la numérotation des commandes est implicite et débute à 0. De plus, un registre R_i est simplement désigné par son index i.

Un exemple devrait clarifier les choses. Considérons la machine qui calcule l'addition de deux registres (les registres R_1 et R_2) et place le résultat dans le registre R_0. Le registre R_3 est utilisé pour effectuer des calculs intermédiaires :

```
0 :  ZERO 0
1 :  ZERO 3
2 :  JUMP 1 3 6
3 :  SUCC 0
4 :  SUCC 3
5 :  JUMP 3 3 2
6 :  ZERO 3
7 :  JUMP 2 3 11
8 :  SUCC 0
9 :  SUCC 3
10 :  JUMP 3 3 7
```

Suivre pas à pas ce programme et remarquer qu'une commande `JUMP i i ℓ` place toujours le pointeur d'instruction sur la ligne ℓ (nous comparons en effet la valeur d'un registre avec elle même dans ce cas !). De plus, et le nom *machine à registres non limités* vient — en partie — de là, il n'est pas nécessaire de déclarer l'utilisation d'un registre avant de l'utiliser (nous commençons par exemple le programme directement avec la commande `ZERO 0`, c'est-à-dire mettre à 0 le registre d'index 0).

1.2 Implantation CAML

Il est impératif d'utiliser dans cette première partie les types CAML suivants :

```
(* line number *)
type line = int

(* register index *)
type regidx = int

(* register value *)
type regval = int

(* register *)
type reg = Reg of regidx * regval

(* URM instruction *)
type urmcmd =
            |Copy of regidx * regidx
            |Jump of regidx * regidx * line
            |Succ of regidx
            |Zero of regidx

(* instruction pointer *)
type instptr = InstPtr of (line * urmcmd) list * (line * urmcmd) list

(* URM *)
type urm = {instptr : instptr; regs : reg list}
```

Les cinq premiers types (`line`, `regidx`, `regval`, `reg` et `urmcmd`) ne demandent pas d'explications supplémentaires. Arrêtons-nous en revanche un moment sur les types `instptr` et `urm`.

— Le type `instptr` permet de gérer le pointeur d'instruction. Il est composé de deux listes. La première liste contient (en ordre inverse) les instructions qui se trouvent avant l'instruction courante, tandis que la seconde liste contient (dans l'ordre) les instructions à partir de l'instruction courante.

  Déplacer le pointeur d'instruction revient à déplacer des instructions d'une liste vers l'autre dans `instptr`.

— Le type enregistrement `urm` représente une machine URM. Il permet d'accéder à l'instruction courante (via `instptr`) et contient tous les registres (index et valeur de chaque registre).

En reprenant l'exemple de l'addition des registres R_1 et R_2 avec le résultat dans le registre R_0, nous obtenons la session CAML suivante :

```
# let add_program = [Zero 0; Zero 3; Jump (1, 3, 6); Succ 0; Succ 3;
  Jump (3, 3, 2); Zero 3; Jump (2, 3, 11); Succ 0; Succ 3; Jump (3, 3, 7)];;
val add_program : urmcmd list =
  [Zero 0; Zero 3; Jump (1, 3, 6); Succ 0; Succ 3; Jump (3, 3, 2);
   Zero 3; Jump (2, 3, 11); Succ 0; Succ 3; Jump (3, 3, 7)]

# let m = urm_mk add_program [Reg (1, 2); Reg (2, 3)];;
val m : urm =
  {instptr =
    instptr ([],
     [(0, Zero 0); (1, Zero 3); (2, Jump (1, 3, 6));
      (3, Succ 0); (4, Succ 3); (5, Jump (3, 3, 2));
      (6, Zero 3); (7, Jump (2, 3, 11)); (8, Succ 0);
      (9, Succ 3); (10, Jump (3, 3, 7))]);
   regs = [Reg (1, 2); Reg (2, 3)]}
# urm_run m;;
- : reg list = [Reg (0, 5); Reg (1, 2); Reg (2, 3); Reg (3, 3)]
```

Il est également possible de lire un programme URM depuis un fichier. La fonction de lecture est fournie. Pour l'utiliser, il est nécessaire de lier le module `Str` au moyen de `#load "str.cma";;` en mode interprété.

```
exception Syntax_error

let rec string_of_file f =
    try
        let str = input_line f in
        str ^ " " ^ (string_of_file f)
    with
        | End_of_file -> ""

let rec program_of_lex lex =
```

```
    match lex with
        |[] -> []
        |"zero" :: arg_1 :: tail ->
            (Zero (int_of_string arg_1)) :: (program_of_lex tail)
        |"succ" :: arg_1 :: tail ->
            (Succ (int_of_string arg_1)) :: (program_of_lex tail)
        |"copy" :: arg_1 :: arg_2 :: tail ->
            (Copy ((int_of_string arg_1), (int_of_string arg_2)))
                :: (program_of_lex tail)
        |"jump" :: arg_1 :: arg_2 :: arg_3 :: tail ->
            (Jump ((int_of_string arg_1), (int_of_string arg_2),
                (int_of_string arg_3)))
                :: (program_of_lex tail)
        |_ -> raise Syntax_error

let program_of_string str =
    let lex = Str.split (Str.regexp "[ \t\n(),]+") str in
    List.iter (fun s -> print_string s; print_newline ()) lex;
    program_of_lex lex
```

1.3 Travail demandé pour la première partie du projet

Voici le détail des différentes tâches à effectuer dans la première partir du projet, avec un sous-ensemble des fonctions nécessaires pour y parvenir.

1.3.1 Pointeur sur instruction

Écrire les fonctions permettant de gérer le pointeur d'instructions. Il y aura besoin, entre autres, de fonctions pour :

— créer un pointeur d'instructions à partir d'une liste de commandes ;
— vérifier la validité du pointeur ;
— déplacer le pointeur d'instructions vers le haut ou vers le bas ;
— obtenir l'instruction courante (sous forme d'instruction `urmcmd` ou de `string`).

Voici quelques exemples d'en-têtes de fonctions :

```
(* Create an instruction pointer for an URM program. *)
(* val instptr_mk : urmcmd list -> instptr = <fun> *)
let instptr_mk ...

(* Move the instruction pointer up. Do nothing if this is not possible. *)
(* val instptr_move_up : instptr -> instptr = <fun> *)
let instptr_move_up ...

(* Get the current command from the instruction pointer. *)
(* Fail if the command pointer is not set on a valid command. *)
(* val instptr_get : instptr -> line * urmcmd = <fun> *)
let instptr_get = ...
```

```
(* Get the current instruction as a string.
 * Returns "null" is the instruction pointer is not valid. *)
(* val instptr_string : instptr -> string = <fun> *)
let instptr_string instptr ...
```

1.3.2 Registres

Écrire les fonctions de gestion des registres, et notamment :

— leur manipulation (accès, affichage, comparaison, *etc.*) ;

— l'accès et la mise à jour d'un registre donné dans une liste de registres.

Voici quelques exemples d'en-tête de fonctions :

```
(* Returns the index of a register. *)
(* val reg_idx : reg -> regidx = <fun> *)
let reg_idx ...

(* Compares two register Ri and Rj.
 * It returns an integer less than, equal to, or greater than zero if
 * the first register index is respectively less than, equal to, or
 * greater than the second register index. *)
(* val reg_compar : reg -> reg -> int = <fun> *)
let reg_compar ...

(* Returns the register value of a register from its index. Fails if
 * there is not register with the sought register index. *)
(* val get_regval : reg list -> regidx -> regval = <fun> *)
let regs_get ...
```

1.3.3 URM

Écrire les fonctions de gestion d'une machine URM :

— deux fonctions qui exécutent un programme `urm` (avec ou sans trace) ;

— une fonction qui fabrique un programme `urm` à partir d'une liste d'instruction et des valeurs initiales des registres initiaux.

```
(* Runs an URM.
 * Returns all registers when the program halts. *)
(* val urm_run : urm -> reg list = <fun> *)
let urm_run ...

(* Runs an URM in trace mode.
 * Returns all registers when the program halts. *)
(* val urm_run_trace : urm -> regval = <fun> *)
let urm_run_trace ...

(* Makes an URM. *)
```

```
(* val urm_mk : urmcmd list -> reg list -> urm = <fun> *)
let urm_mk ...
```

Dans le mode « trace », l'instruction courante ainsi que tous les registres sont affichées avant d'exécuter l'instruction courante. En reprenant l'exemple de l'addition des registres R_1 et R_2 avec le résultat dans le registre R_0, nous obtenons la session CAML suivante :

```
# urm_run_trace m;;
0: Zero 0
(1,2),(2,3)

1: Zero 3
(0,0),(1,2),(2,3)

6: Jump 1 3 6
(0,0),(1,2),(2,3),(3,0)

3: Succ 0
(0,0),(1,2),(2,3),(3,0)

4: Succ 3
(0,1),(1,2),(2,3),(3,0)

2: Jump 3 3 2
(0,1),(1,2),(2,3),(3,1)

6: Jump 1 3 6
(0,1),(1,2),(2,3),(3,1)

3: Succ 0
(0,1),(1,2),(2,3),(3,1)

4: Succ 3
(0,2),(1,2),(2,3),(3,1)

2: Jump 3 3 2
(0,2),(1,2),(2,3),(3,2)

6: Jump 1 3 6
(0,2),(1,2),(2,3),(3,2)

6: Zero 3
(0,2),(1,2),(2,3),(3,2)

11: Jump 2 3 11
(0,2),(1,2),(2,3),(3,0)

8: Succ 0
(0,2),(1,2),(2,3),(3,0)
```

```
9: Succ 3
(0,3),(1,2),(2,3),(3,0)

7: Jump 3 3 7
(0,3),(1,2),(2,3),(3,1)

11: Jump 2 3 11
(0,3),(1,2),(2,3),(3,1)

8: Succ 0
(0,3),(1,2),(2,3),(3,1)

9: Succ 3
(0,4),(1,2),(2,3),(3,1)

7: Jump 3 3 7
(0,4),(1,2),(2,3),(3,2)

11: Jump 2 3 11
(0,4),(1,2),(2,3),(3,2)

8: Succ 0
(0,4),(1,2),(2,3),(3,2)

9: Succ 3
(0,5),(1,2),(2,3),(3,2)

7: Jump 3 3 7
(0,5),(1,2),(2,3),(3,3)

11: Jump 2 3 11
(0,5),(1,2),(2,3),(3,3)

- : reg list = [Reg (0, 5); Reg (1, 2); Reg (2, 3); Reg (3, 3)]
```