Machine à registres non limités

Projet CAML

L3 Informatique

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 Deuxième partie

1.1 Jeu d'instructions URM

1.1.1 Machines à registres non limités

Le jeu d'instructions URM (`COPY`, `JUMP`, `SUCC` et `ZERO`) ne rend pas l'écriture de programmes très facile. Nous allons considérer dans cette deuxième partie un jeu d'instructions EURM (pour « extented URM ») de plus haut niveau.

Deux points importants avant de commencer :

— nous continuerons d'utiliser le simulateur URM. De fait, un programme EURM sera d'abord compilé (*i.e.*, transformé) en un programme URM équivalent afin d'être exécuté sur votre simulateur conçu dans la 1re partie ;

— le jeu d'instructions EURM ne permet finalement que de simplifier l'écriture des programmes. Il n'est pas plus puissant puisque tout programme EURM peut être réécrit en un programme URM équivalent (mais en général beaucoup plus long).

1.1.2 Commandes autorisées

Le jeu d'instructions EURM comprend les commandes suivantes :

— `Add i j`

Additionne les valeurs des registres R_i et R_j et place le résultat dans le registre R_i. (le registre R_j conserve sa valeur initiale). Incrémente ensuite (de 1) le pointeur d'instruction.

Les registres R_i et R_j doivent avoir été initialisés.

— `Comment` *string*

Introduit un commentaire dans le programme.

— `Copy` i j

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

Le registre R_j doit avoir été initialisé.

— `Dec` i

Décrémente (de 1) la valeur du registre R_i. Incrémente ensuite (de 1) le pointeur d'instruction.

Le registre R_i doit avoir été initialisé.

— `Eq?` i j *label-string*

Si les valeurs des registres R_i et R_j sont égales, le pointeur d'instruction est placé sur la ligne déclarant le label *label-string*. Sinon, incrémente (de 1) le pointeur d'instruction.

Les registres R_i et R_j doit avoir été initialisés. Le cas $i = j$ est valide.

— `GEq?` i j *label-string*

Si la valeur du registre R_i est supérieure ou égale à la valeur du registre R_j, le pointeur d'instruction est placé sur la ligne déclarant le label *label-string*. Sinon, incrémente (de 1) le pointeur d'instruction.

Les registres R_i et R_j doit avoir été initialisés. Le cas $i = j$ est valide.

— `GT?` i j *label-string*

Si la valeur du registre R_i est supérieure strictement à la valeur du registre R_j, le pointeur d'instruction est placé sur la ligne déclarant le label *label-string*. Sinon, incrémente (de 1) le pointeur d'instruction.

Les registres R_i et R_j doit avoir été initialisés. Le cas $i = j$ est valide.

— `Goto` *label-string*

Le pointeur d'instruction est placé sur la ligne déclarant le label *label-string*.

— `Inc` i

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

Le registre R_i doit avoir été initialisé.

— `Label` *label-string*

Déclare le label *label-string*.

— `Mult` i j

Multiplie les valeurs des registres R_i et R_j et place le résultat dans le registre R_i. (le registre R_j conserve sa valeur initiale). Incrémente ensuite (de 1) le pointeur d'instruction.

Les registres R_i et R_j doivent avoir été initialisés.

— `Quit`

Termine l'exécution du programme.

— `Sub i j`

Calcule la différence des valeurs des registres R_i et R_j (c'est à dire $R_i - R_j$) et place le résultat dans le registre R_i. (R_j conserve sa valeur initiale). Incrémente ensuite (de 1) le pointeur d'instruction.

Les registres R_i et R_j doivent avoir été initialisés.

— `Zero i`

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

— `Zero? i` *label-string*>

Si la valeur du registre R_i est 0, le pointeur d'instruction est placé sur la ligne déclarant le label *label-string*. Sinon, incrémenter (de 1) le pointeur d'instruction.

Le registre R_i doit avoir été initialisé.

Aucune commande EURM n'est sensible à la casse ; les commandes `COPY`, `copy` et `CoPy` sont donc équivalentes.

Un exemple devrait clarifier les choses. Considérons le programme qui calcule la factorielle (plus précisément, ce programme place dans le registre R_1 la valeur $n!$, où n est la valeur initiale du registre R_1).

```
 0 :  COMMENT compute |R1|! and place the result in register R1.
 1 :  ZERO? 1 R1=0
 2 :  GOTO R1>0
 3 :  COMMENT R1=0, we are done as 0! = 1.
 4 :  LABEL R1=0
 5 :  INC 1
 6 :  GOTO done
 7 :  COMMENT R1>0, use n! = 1 x 2 x ... x n
 8 :  LABEL R1>0
 9 :  COPY 2 1
10 :  ZERO 1
11 :  INC 1
10 :  ZERO 3
11 :  INC 3
12 :  COMMENT main loop
13 :  LABEL loop
14 :  MULT 1 3
15 :  EQ? 2 3 done
16 :  INC 3
17 :  GOTO loop
18 :  COMMENT that's all folks.
19 :  LABEL done
20 :  QUIT
```

Suivre pas à pas ce programme et remarquer que le registre R_1 prend successivement les valeurs $1, 1\times 2, 1\times 2\times 3$, *etc.* Remarquez également que la dernière commande `QUIT` n'est pas indispensable (pourquoi ?).

1.2 Implantation CAML

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

```
(* label *)
type label = string

(* EURM instruction *)
type eurmcmd =
    |Add of regidx * regidx
    |Comment of string
    |Copy of regidx * regidx
    |Dec of regidx
    |EqPredicate of regidx * regidx * label
    |GEqPredicate of regidx * regidx * label
    |GTPredicate of regidx * regidx * label
    |Goto of label
    |Inc of regidx
    |Label of label
    |LEqPredicate of regidx * regidx * label
    |LTPredicate of regidx * regidx * label
    |Mult of regidx * regidx
    |Quit
    |Sub of regidx * regidx
    |Zero of regidx
    |ZeroPredicate of regidx * label
```

Pour éviter les problèmes de typage explicite [1], nous vous conseillons fortement de renommer le type `urmcmd` de la première partie [2]. Par exemple :

```
(* URM instruction *)
type urmcmd =
    |URMCopy of regidx * regidx
```

1. `Copy i j` est-il de type `urmcmd` ou `eurmcmd` ?
2. Il est conseillé de créer un nouveau fichier qui va contenir le code de cette 2e partie et contiendra la modification suggérée.

```
  |URMJump of regidx * regidx * line
  |URMSsucc of regidx
  |URMZero of regidx
```

1.3 Compilation

Vous êtes libres d'utiliser l'algorithme de votre choix pour la compilation d'un programme EURM vers un programme URM. Néanmoins, nous insistons sur le fait que le but n'est pas d'obtenir la compilation la plus efficace possible mais d'écrire des fonctions simples que l'on chaîne pour produire le compilateur.

Nous vous proposons la stratégie suivante : le programme EURM est réécrit plusieurs fois en un programme EURM utilisant des jeux d'instructions EURM de plus en plus réduits (jusqu'à pourvoir se traduire en URM). Par exemple, il est facile de voir qu'une instruction `Mult i j` peut être réécrite en utilisant les instructions `Copy`, `Zero`, `Label`, `Add`, `Inc` et `Goto`.

Plus précisément, nous proposons l'algorithme suivant.

— **Prétraitement.**

Les commentaires (`Comment`) sont supprimés. Pour éviter les conflits, les labels (`Label` *string-label*) sont réécrit en une suite de labels `Label 1`, `Label 2`, ...

— **Étape 1.**

Les instructions EURM `Dec`, `GEqPredicate`, `LEqPredicate`, `LTPredicate`, `Mult` et `ZeroPredicate` sont réécrites. En d'autres termes, **Étape 1** produit un programme EURM équivalent qui n'utilise pas les instructions `Dec`, `GEqPredicate`, `LEqPredicate`, `LTPredicate`, `Mult` et `ZeroPredicate`.

— **Étape 2.**

Les instructions EURM `Add`, `GTPredicate` et `Sub` sont réécrites. En d'autres termes, **Étape 2** produit un programme EURM équivalent qui n'utilise pas les instructions `Dec`, `GEqPredicate`, `LEqPredicate`, `LTPredicate`, `Mult` (**Étape 1**), ni les instructions `ZeroPredicate`, `Add`, `GTPredicate` et `Sub`.

— **Étape 3.**

Les instructions EURM `Goto` sont réécrites. En d'autres termes, **Étape 3.** produit un programme EURM équivalent qui n'utilise pas les instructions `Dec`, `GEqPredicate`, `LEqPredicate`, `LTPredicate`, `Mult` (**Étape 1**), ni les instructions `ZeroPredicate`. `Add`, `GTPredicate` et `Sub` (**Étape 2**), ni l'instruction `Goto`.

— **Étape 4.**

Les instructions EURM `Inc`, `EqPredicate`, `Label` et `Zero` sont réécrites. Le programme sur un jeu d'instructions URM est ensuite produit.

Notez que le programme URM est produit en sortie de **Étape 4**. En d'autres termes, les étapes **Prétraitement**, **Étape 1**, **Étape 2** et **Étape 3** produisent un jeu d'instructions EURM, tandis que l'étape **Étape 4** produit un jeu d'instructions URM.

L'état courant. Une des principales difficultés que vous rencontrerez sera la gestion des labels et des registres. En effet, pour transformer certaines instructions, on a besoin d'ajouter de nouveaux labels (resp. registres). Pour cela, il faut être capable d'en générer qui soient différents des labels (resp. registres) déjà existants. Une technique simple consiste à déterminer et maintenir le numéro maximum d'un label (resp. registre) existant et de toujours générer un numéro plus grand.

Il sera donc nécessaire de maintenir un *état courant* permettant de savoir quels sont les prochains numéros à engendrer. En supposant que cet état est de type `state`, nous obtenons assez naturellement les typages suivants :

```
val compile_preprocess : eurmcmd list -> eurmcmd list = <fun>
val compile_stage1 : eurmcmd list -> state -> eurmcmd list * state = <fun>
val compile_stage2 : eurmcmd list -> state -> eurmcmd list * state = <fun>
val compile_stage3 : eurmcmd list -> state -> eurmcmd list * state = <fun>
val compile_stage4 : eurmcmd list -> state -> urmcmd list * state = <fun>
```

Vous écrirez en particulier la fonction suivante permettant de compiler (*i.e.*, transformer) un programme EURM en un programme URM équivalent.

```
val urm_from_eurm : eurmcmd list -> urmcmd list = <fun>
```

N'oubliez pas non plus d'augmenter la fonction `program_of_lex` réalisant le parser (donnée dans l'énoncé de la partie 1 du projet) de sorte à prendre les nouvelles instructions en compte.

Voici (page suivante) un exemple d'utilisation reprenant la fonction calculant la factorielle (plus précisément, ce programme place dans le registre R_1 la valeur $n!$, où n est la valeur initiale du registre R_1).

```
# eurm_factorial;;
- : eurmcmd list =
[Comment "Compute r1! and place the result in r1"; ZeroPredicate (1, "r1=0");
 Goto "r1>0"; Comment "r1 holds 0"; Label "r1=0"; Inc 1; Goto "done";
 Comment "r1 holds a positive integer"; Label "r1>0"; Copy (2, 1); Zero 1;
 Inc 1; Zero 3; Inc 3; Comment "main loop"; Label "loop"; Mult (1, 3);
 EqPredicate (2, 3, "done"); Inc 3; Goto "loop"; Label "done"; Quit]
# let prog = urm_from_eurm eurm_factorial;;
val prog : urmcmd list =
  [URMZero 4; URMJump (1, 4, 4); URMZero 8; URMJump (8, 8, 7); URMSucc 1;
   URMZero 9; URMJump (9, 9, 29); URMCopy (2, 1); URMZero 1; URMSucc 1;
   URMZero 3; URMSucc 3; URMCopy (5, 1); URMZero 1; URMZero 6;
   URMJump (3, 6, 25); URMZero 7; URMJump (5, 7, 22); URMSucc 1; URMSucc 7;
   URMZero 10; URMJump (10, 10, 17); URMSucc 6; URMZero 11;
   URMJump (11, 11, 15); URMJump (2, 3, 29); URMSucc 3; URMZero 12;
   URMJump (12, 12, 12); URMZero 13; URMJump (13, 13, 38)]
# let m = urm_mk prog [Reg (1, 5)];;
val m : urm =
  {instptr =
    InstPtr ([],
     [(0, URMZero 4); (1, URMJump (1, 4, 4)); (2, URMZero 8);
      (3, URMJump (8, 8, 7)); (4, URMSucc 1); (5, URMZero 9);
      (6, URMJump (9, 9, 29)); (7, URMCopy (2, 1)); (8, URMZero 1);
      (9, URMSucc 1); (10, URMZero 3); (11, URMSucc 3); (12, URMCopy (5, 1));
      (13, URMZero 1); (14, URMZero 6); (15, URMJump (3, 6, 25));
      (16, URMZero 7); (17, URMJump (5, 7, 22)); (18, URMSucc 1);
      (19, URMSucc 7); (20, URMZero 10); (21, URMJump (10, 10, 17));
      (22, URMSucc 6); (23, URMZero 11); (24, URMJump (11, 11, 15));
      (25, URMJump (2, 3, 29)); (26, URMSucc 3); (27, URMZero 12);
      (28, URMJump (12, 12, 12)); (29, URMZero 13);
      (30, URMJump (13, 13, 38))]);
   regs = [Reg (1, 5)]}
# urm_run m;;
- : reg list =
[Reg (1, 120); Reg (2, 5); Reg (3, 5); Reg (4, 0); Reg (5, 24); Reg (6, 5);
 Reg (7, 24); Reg (8, 0); Reg (10, 0); Reg (11, 0); Reg (12, 0); Reg (13, 0)]
```