Machines Virtuelles – MV6
CM 2

Peter Habermehl (merci aux auteurs des transparents)

Université Paris Cité
UFR Informatique
Institut de Recherche d'Informatique Fondamentale

Peter.Habermehl@irif.fr

30 janvier 2025

Rappel
de la semaine dernière

Exemple d'une machine à a-pile

Un état de la machine est constitué de:

- une pile S
- deux registres:

  A: accumulateur

  PC: program counter

  pointeur vers la prochaine instruction
- un programme, cet à-dire tableau d'instructions C

Jeu d'instructions constituant C:

Push	empile le contenu de A sur S
Pop	dépile la tête de S et l'écrit dans A
Consti n	remplace le contenu de A par n
Addi	dépile un mot n de S, remplace A par $A + n$
Ori	dépile un mot n de S, remplace A par 0 si $A = n = 0$, par 1 sinon
Eqi	dépile un mot n de S, remplace A par 1 si $n = A$, 0 sinon

Implémentation en OCaml

Un type pour les instructions et l'état

```
1  type instr = Push | Pop | Consti of int | Addi | Ori | Eqi
2
3  type state = {
4      mutable acc: int;   (* accumulateur *)
5      code: instr array;  (* programme    *)
6      mutable pc: int;    (* indice prochaine instruction *)
7      stack: int array;   (* pile *)
8      mutable sp: int;    (* indice du sommet de la pile *)
9    }
```

Une fonction d'initialisation

```
1  (* creation etat initial d'un programme sur des entrees *)
2  let init (p : instr array) (args : int list) : state =
3    {
4      acc =(
5        match args with [] -> 42    (* valeur quelconque *)
6                      | h::tl -> h (* valeur en entree  *)
7      );
8      code = p;
9      pc = 0;
10     stack =
11      begin
12       let args = List.rev args (* respecter ordre arguments c
13       in Array.init (List.length args + 50)
14                     (fun i -> if i < List.length args-1
15                                then List.nth args i
16                                else 42 (* valeur quelconque *)
17      end;
18     sp = max (-1) (List.length args-2);
19   }
```

Un array pour une pile ?

- dans la suite on devrait ajouter l'instruction Acc n accédant aux cases internes à la pile.
- donc plus efficace utiliser un tableau plutôt qu'une liste,
  - on alloue un espace suffisant
  - on garde un pointeur sp vers la tête
- quelques exemples:

$$1 :: 2 :: 3 :: \texttt{nil} \longrightarrow (\texttt{sp} = 2, \boxed{3}\boxed{2}\boxed{1}\boxed{-}\boxed{-})$$

$$3 :: \texttt{nil} \longrightarrow (\texttt{sp} = 0, \boxed{3}\boxed{-}\boxed{-}\boxed{-}\boxed{-})$$

$$\texttt{nil} \longrightarrow (\texttt{sp} = -1, \boxed{-}\boxed{-}\boxed{-}\boxed{-}\boxed{-})$$

- et un mémento:
  - ajouter un élément = incrémenter sp
  - supprimer un élément = décrementer sp

La fonction exec à petits pas

```
(* execution a petits pas *)
let exec_small (m : state) : state =
    begin match m.code.(m.pc) with
      | Push ->
        m.sp <- m.sp+1;
        m.stack.(m.sp) <- m.acc
      | Pop ->
        m.acc <- m.stack.(m.sp);
        m.sp <- m.sp-1
      | Consti n ->
        m.acc <- n
      | Addi ->
        m.acc <- m.stack.(m.sp) + m.acc;
        m.sp <- m.sp-1
      | Ori ->
        m.acc <- if m.stack.(m.sp)=0 && m.acc=0 then 0 else 1;
        m.sp <- m.sp-1
      | Eqi ->
        m.acc <- if m.stack.(m.sp)=m.acc then 1 else 0;
        m.sp <- m.sp-1
    end;
    m.pc <- m.pc+1; m
```

La fonction exec à grands pas

```
1  (* execution a grand pas *)
2  let rec exec_big (m : state) : state =
3    if m.pc < Array.length m.code
4    then exec_big (exec_small m)
5    else m
6
7  (* execution a grand pas, version equivalente *)
8  let exec_big_loop (m : state) : state =
9    while m.pc < Array.length m.code do
10     let _ = exec_small m in ()
11   done; m
```

Code octet
assembleur / disassembleur

Code octet (bytecode)

- Pour l'instant, un programme est une valeur OCaml, (e.g. [| Consti 2; Push; Addi |])

- Mais un programme est en mémoire, sous forme d'un fichier binaire qui encode la liste des instructions.

- Cette representation binaire doit être:
  - compacte
  - efficace

  et prend le nom de code octet.

- En général, un code octet est un code binaire proche d'un code natif mais à exécuter par une machine virtuelle.
- Sa proximité avec le code natif réduit le travail de la MV (la couche d'interprétation).

Code octet (bytecode)

Un codage des instructions

- une instruction par octet:

  bit 0-2 *opcode* de l'instruction
  bit 3-7 vide, sauf si l'instruction est Consti n, auquel cas c'est n

- les opcodes sont:

`Push` $\mapsto$ 000	`Addi` $\mapsto$ 001	`Eqi` $\mapsto$ 010
`Ori` $\mapsto$ 011	`Consti` $\mapsto$ 100	`Pop` $\mapsto$ 101

Limitations:

- On ne peut coder que les constantes entre 0 et 31 (!)
- Il n'y a plus beaucoup de place pour étendre la table d'instructions.

Exemple

code-octet binaire	00010100	00000000	00000001	00000101
↓ decode				
instructions	Consti 2	Push	Addi	Pop
↓ execute				
résultat (dans A)	4			

$\Longrightarrow$ on s'approche au cycle Fetch / Decode / Execute d'une CPU

Exercice

Coder les programmes suivants en code octet:

- Consti 2; Push; Consti 3; Addi; Pop
- Consti 2; Push; Addi; Pop; Pop
- Consti 2; Push; Consti 3; Push; Addi; Addi

Decoder les codes octet suivants:

- 00010100 00000000 00000001 00011100 00000001
- 01001100 00000000 00000001 00000001 00000101

Notion d'assembleur

Définitions

L'assembleur: assemble est la fonction qui encode un programme en code-octet

Le désassembleur: disassemble est la fonction qui décode du code-octet en un programme

Remarque: par abus de langage, on appelle souvent assembleur la syntaxe prise en entrée par l'assembleur.

Propriété: ces deux fonctions forment une bijection:

$$\text{disassemble}\ (\text{assemble}\ p) = p$$

Implémentation en OCaml

Un type pour le code octet

- on represente les octets par le type chr de Ocaml
- et les chaines des octets par string
- pour (de)coder Consti n il est nécessaire déplacer les bits associés à *n* pour faire espace au opcode de Consti:

  `n lsl m` déplace n à gauche par m bits.
  (logical shift left)

  `n lsr m` déplace n à droite par m bits.
  (logical shift right)

L'assembleur

```
(*assembleur pour programmes mv*)
let assemble (p: instr array) : string =
  let encode i =
    match i with
          | Push -> Char.chr 0
          | Addi -> Char.chr 1
          | Eqi ->  Char.chr 2
          | Ori ->  Char.chr 3
          | Consti n ->
            assert (n  < 32) ;
            Char.chr (4 + n lsl 3)
          | Pop -> Char.chr 5
  in Array.fold_right
       (fun i s -> (String.make 1 (encode i))^s)
       p
       ""

```

Le désassembleur

```
1  (*desassembleur pour programmes mv*)
2  let disassemble (s: string) : instr array =
3    let decode c =
4      match Char.code c with
5                | 0 -> Push
6                | 1 -> Addi
7                | 2 -> Eqi
8                | 3 -> Ori
9                | n when (n mod 8 = 4) -> Consti (n lsr 3)
10               | 5 -> Pop
11               | _ -> failwith "invalid byte-code"
12   in
13   Array.init (String.length s) (fun i -> decode s.[i])
```