Machines Virtuelles – MV6

Séance 3

Benoît Valiron
Université Paris Diderot

2015

Retour sur les séances précédantes

On a défini une MV dont un état consiste en

- une pile S
- un registre A (l'accumulateur)
- un tableau d'instructions C
- un pointeur PC vers l'instruction courante

Jeu d'instructions constituant C :

push, pop, consti n, addi, andi, eqi,

À chaque fois, on incrémente PC de 1 unité.

Retour sur les séances précédantes

Les instructions sont codés en code-octet :

On utilise pas les instructions comme langage « processeur » pour la machine virtuelle.

Dans la machine virtuelle :
- les booléens sont codés comme `false` $= 0$ et `true` $= 1$.
- les données sont non-typés dans la machine (`int` et `bool`)

Au cours de la compilation :

Le code est compilé en instructions puis assemblé en code-octet

La machine virtuelle :

Lit un fichier contenant le code octet, le désassemble en instructions puis exécute les instructions.

Retour sur les séances précédantes

On a défini un mini-langage Myrte avec
- les constantes true, false, 0, 1, 2, ...
- les opérations binaires $+$, $=$, $\wedge$.
- les parenthèses

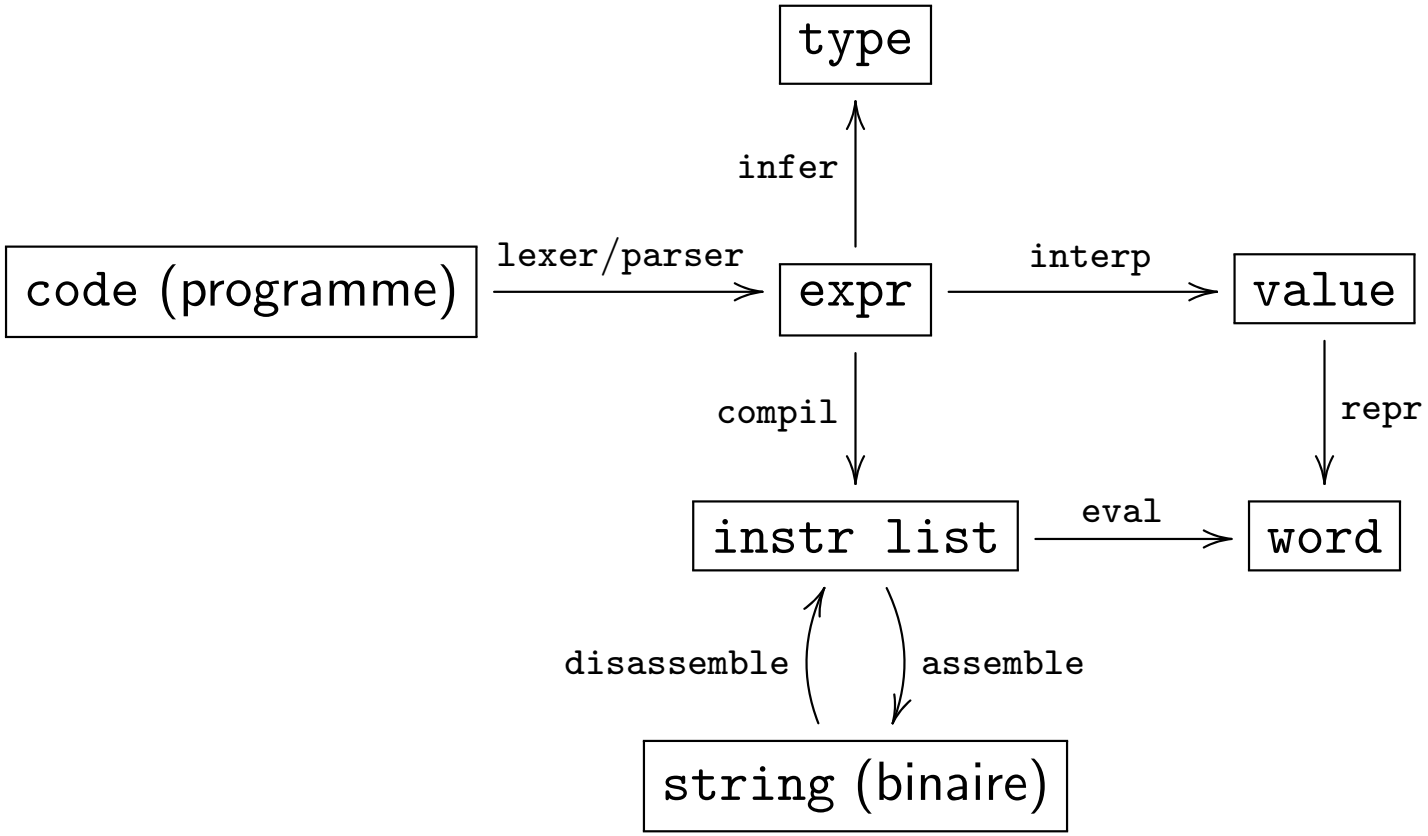
et sa sémantique habituelle.

On a vu comment compiler une expression Myrte en une liste d'instructions.

Préservation d'un invariant qui dit que quand la machine a fini de traiter l'encodage d'une expression,

- **le résultat est lu dans A**
- **le PC pointe juste après l'encodage**
- **la pile est restaurée**

Retour sur les séances précédantes



Retour sur les séances précédantes

Exemples (de l'exercice 3.1 du TD) :

1. `(((1 + 2) + 3) + 4) = 4`
2. `(1 = 2) + 3`

Extension du langage Myrte

Conventions

- Les entiers sont 32 bits signés (et non plus juste entre 0 et 31)
- On oublie les contraintes du code-octet proposé en séance 1
- On travaille uniquement sur
  - La définition des instructions
  - L'extension de la grammaire des expressions
  - La compilation `expr -> instr list`.

Ajout d'un test

C'était le but de l'exercice 4.1 du TD :

- le PC doit pouvoir « sauter » d'un point à un autre du code.
- ajout de branchements

Au niveau des instructions

- `BranchIf n`

  si $A = 0$ $\quad PC := PC + 1$

  si $A \neq 0$ $\quad PC := PC + n$

- `Branch n`

  $PC := PC + n$ inconditionellement

Au niveau du langage

```
type expr =
    ...
  | If of expr * expr * expr
```

Ajout d'un test

Exemple : Évaluation de

```
if (4=5) then e1 else e2
```

(au tableau)

Ajout d'un test

Exemple : Évaluation de

```
if (4=5) then (2+3) else (6+7)
```

(au tableau)

Ajout d'un test

```
let rec interp : expr -> value = function
| Const v              -> v
| Binop (b, e1, e2) ->
      (match b, interp e1, interp e2 with
        | Add, Int i,  Int j  -> Int  (i + j)
        | Eq,  Int i,  Int j  -> Bool (i = j)
        | And, Bool i, Bool j -> Bool (i && j)
        | _ -> failwith "type error")
| If (e1, e2, e3) ->
      match interp e1 with
        | Bool true -> interp e2
        | Bool false -> interp e3
        | _ -> failwith "type error"
```

Ajout d'un test

Exemple : Compilation de

```
if (4=5) then (2+3) else (6+7)
```

- Rappel : `true` $\simeq 1$ et `false` $\simeq 0$
- Arbre de syntaxe
- Utilisation de l'invariant
- Étiquettes (« *labels* ») pour la lisibilité

(au tableau)

Ajout d'un test

```
let rec compil : expr -> instr list = function
  | Const v            -> [Consti (repr v)]
  | Binop (o, e1, e2) -> compil e1   @
                         [Push]      @
                         compil e2   @
                         [op o; Pop]
  | If (e1, e2, e3) -> let i2 = compil e2 in
                       let i3 = compil e3 in
                         compil e1 @
                         [BranchIf (2 + List.length i3)] @
                         i3 @
                         [Branch (1 + List.length i2)] @
                         i2
```

Ajout d'un test

Exemple : Compilation de

```
(if (3 = 1+2) then 4 else 5) + (6 + 7)
```

(au tableau)

Ajout d'un test

Exemple : Compilation de

$$\left(\begin{array}{l} \texttt{if } \Big(4 = \big(\texttt{if } (2 = 3) \texttt{ then } 4 \texttt{ else } 5\big)\Big) \\ \texttt{then } 1 + \big(\texttt{if false then } 6 \texttt{ else } 7\big) \\ \texttt{else } 8 \end{array}\right) + 9$$

(au tableau)

Utilisation de variables (`let-in`)

Exemple : Évaluation de

```
let x = 1 + 2 in x + 3
```

(au tableau)

Utilisation de variables (`let-in`)

Exemple : Évaluation de

```
let x = (let x = 1 in x + 2) in x + 3
```

- Portée d'une variable
- Variables liées

(au tableau)

Utilisation de variables (`let-in`)

En ocaml :

```
- let x = 1 in let x = x + 2 in x
- let x = 1 in let rec x = x + 2 in x
```

Utilisation de variables (`let-in`)

Exemple : Portée et lien des variables dans

$$\texttt{let x} = \left(\texttt{ let x} = \texttt{1 in x} + \texttt{3} \right) \texttt{ in}$$
$$\left(\left(\texttt{ let x} = \left(\begin{array}{l} \texttt{let x} = \texttt{x} + \texttt{4} \\ \texttt{in x} + \texttt{5} \end{array} \right) \texttt{ in x} + \texttt{2} \right) + \texttt{x} \right)$$

(au tableau)

Utilisation de variables (`let-in`)

Évaluation de

```
let x = e1 in e2
```

– Un environnement identifie des variables à des valeurs
– Appel par valeur / appel par nom

Utilisation de variables (`let-in`)

Évaluation de

```
let x = (print_string "hello"; 1) in x + x
```

Ocaml est appel-par-valeur.

(au tableau)

Utilisation de variables (`let-in`)

Ajout de `let-in` en Myrte (appel-par-valeur)

Extension de `expr`

```
type var   = string

type expr =
    ...
  | Var of var
  | Let of var * expr * expr
```

Un environnement

```
type envexpr = var -> value

let empty_envexpr = fun v -> failwith "Variable not there"
```

Utilisation de variables (`let-in`)

```
let rec interp : envexpr * expr -> value = function
...
| env, Var s              -> env s
| env, Let (s, e2, e3) ->
    let r = interp (env, e2) in
    let new_env v = if (v = s) then r else env v in
    interp (new_env, e3)
```

Exemple avec

```
let x = 1 in let y = x in let x = y + 3 in y + x
```