

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 précédentes

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 précédentes

On a défini un mini-langage Myrte avec

- les constantes true, false, 0, 1, 2, ...
- les opérations binaires +, =, \wedge .
- les parenthèses

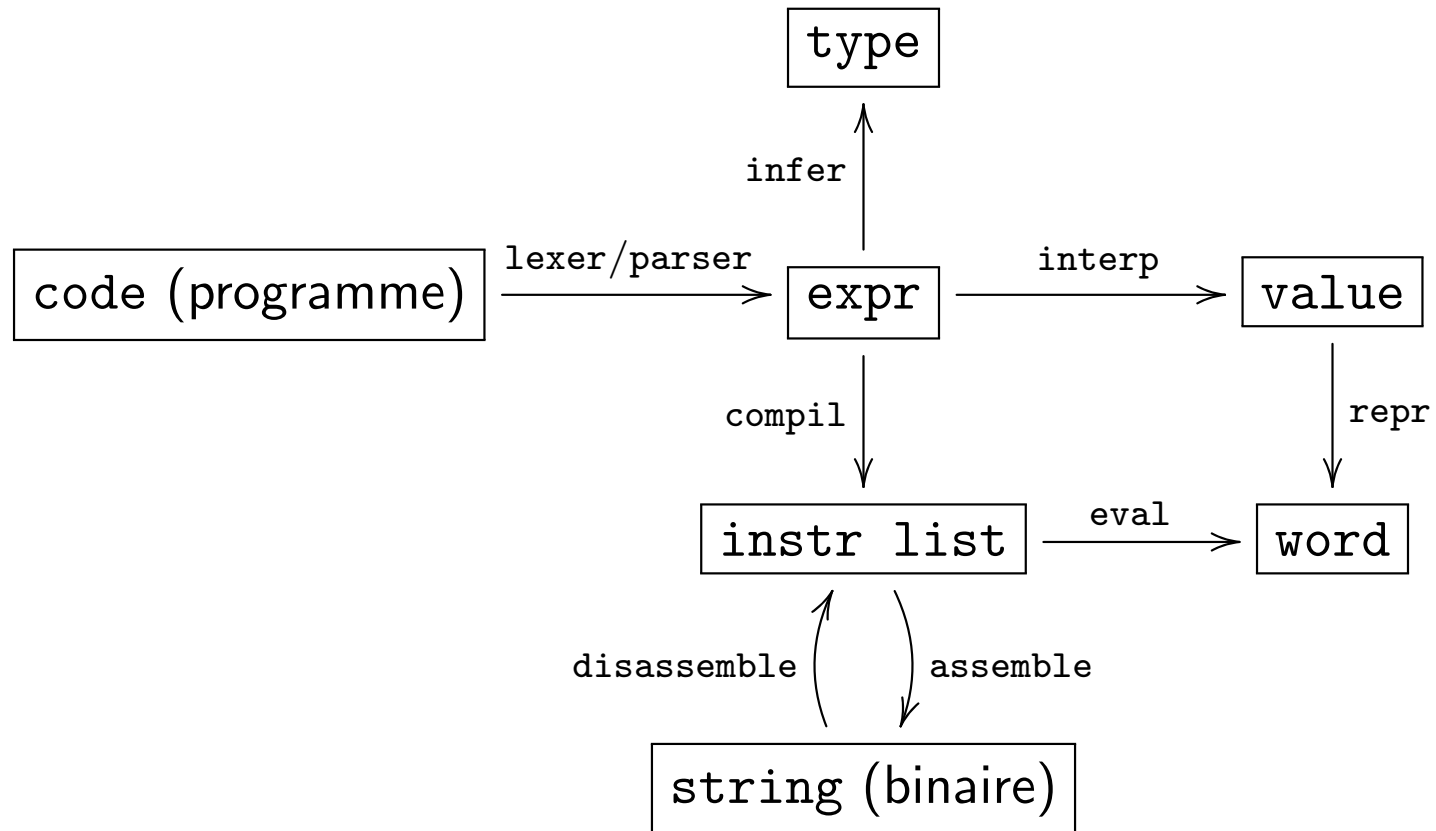
et sa sémantique habituelle.

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 précédentes



Retour sur les séances précédentes

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 $A = 0$ $PC := PC + 1$

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

- Branch n

$PC := PC + n$ inconditionnellement

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 true -> interp e3
    | _ -> failwith "type error"
```

Ajout d'un test

Exemple : Compilation de

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

- Rappel : $\text{true} \simeq 1$ et $\text{false} \simeq 0$
- 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 + length i3) @
                        i3 @
                        [Branch (1 + 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

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

(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

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

(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 value = ... | ValVar of var

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

Un environnement

```
type envexpr = var -> value

let empty_envexpr = fun v -> ValVar v
```

Utilisation de variables (let-in)

```
let rec interp : envexpr * expr -> value = function
...
| env, Var s          -> expr_of_value (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)

let expr_of_value : value -> expr = function
| ValVar s -> Var s
| v        -> Const v
```

Exemple avec

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


Utilisation de variables (let-in)

Exemple :

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

(au tableau)

Utilisation de variables (let-in)

Comment compiler

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

Utilisation de variables (let-in)

Comment compiler

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

Ajout d'une instruction

Acc n place la n-ème valeur de S dans A

Utilisation de variables (let-in)

Comment compiler

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

Ajout d'une instruction

Acc n place la n-ème valeur de S dans A

Utilisation de variables (let-in)

Comment compiler

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

Ajout d'une instruction

Acc n place la n-ème valeur de S dans A

Compilation

- Les valeurs des variables sont stockées dans la pile
- Utilisation d'un **environnement** comme pour l'interprétation

```
type envcomp = var -> int option
```

```
let empty_envcomp = fun v -> None
```

Utilisation de variables (let-in)

```
let rec compil : envcomp * expr -> instr list = function
  | env, Var s          ->
      match env x with
      | None      -> failwith "Variable non-définie"
      | Some n    -> [Acc n]
  | env, Let (s,e1,e2) ->
      let new_env x =
        if (x = s) then Some 0 else match env x with
        | None      -> None
        | Some y    -> Some (1 + y)
      in
        compil (env, e1) @
        [Push] @
        compil (new_env, e2) @
        [Pop]
```

Utilisation de variables (let-in)

On stocke les valeurs des variables dans la pile... Or celle-ci est modifiée par les autres opérations.

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

Utilisation de variables (let-in)

```
let rec compil : envcomp * expr -> instr list = function
  | env, Var s           -> ...
  | env, Let (s,e1,e2)   -> ...
  | env, Const v         -> [Consti (repr v)]
```


Utilisation de variables (let-in)

```
let rec compil : envcomp * expr -> instr list = function
  | env, Var s           -> ...
  | env, Let (s,e1,e2)   -> ...
  | env, Const v         -> [Consti (repr v)]
  | env, Binop(o,e1,e2)  -> compil ??? e1      @
                           [Push]              @
                           compil  ??? e2      @
                           [op o; Pop]
```

Utilisation de variables (let-in)

```
let rec compil : envcomp * expr -> instr list = function
  | env, Var s           -> ...
  | env, Let (s,e1,e2)   -> ...
  | env, Const v         -> [Consti (repr v)]
  | env, Binop(o,e1,e2)  -> compil (env, e1)      @
                           [Push]                @
                           compil (env + 1, e2)    @
                           [op o; Pop]
```

Utilisation de variables (let-in)

```
let rec compil : envcomp * expr -> instr list = function
  | env, Var s          -> ...
  | env, Let (s,e1,e2)  -> ...
  | env, Const v        -> [Consti (repr v)]
  | env, Binop(o,e1,e2) -> compil (env,e1)      @
                           [Push]                @
                           compil (envsucc env,e2) @
                           [op o; Pop]
```

```
let envsucc e = fun x -> match (e x) with
  | None    -> None
  | Some n  -> n + 1
```

Utilisation de variables (let-in)

```
let rec compil : envcomp * expr -> instr list = function
  | env, Var s           -> ...
  | env, Let (s,e1,e2)   -> ...
  | env, Const v         -> ...
  | env, Binop(o,e1,e2)  -> ...
  | env, If (e1, e2, e3) ->
      let i2 = compil (env,e2) in
      let i3 = compil (env,e3) in
      compil (env,e1) @
      [BranchIf (2 + length i3) @
       i3 @
       [Branch (1 + length i2)] @
       i2
```

Utilisation de variables (let-in)

Exemple :

```
let x = 1 + 2 in let y = 4 + x in x + y
```

Utilisation de variables (let-in)

Exemple :

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

```
let y = (if x = 3 then x + 4 else x + 5) in x + y
```

Utilisation de variables (let-in)

Exemple :

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

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

```
let z = y + x in z + y + x
```

Conclusion

On a augmenté la MV avec

- Des branchements
- Des accès arbitraires dans la pile

ce qui permet de compiler un langage d'expressions fonctionnelles
(mais sans fonctions).

Il manque encore

- un **tas** : pour des tableaux, des structures et des fonctions
- Nous verrons cela dans les séances prochaines, avec la MV d'OCaml

Conclusion

Optimisations possibles de notre compilateur Myrte étendu

- Tests sur des constantes

```
if true then e1 else e2
```

- Variables initialisées avec une constante

```
let x = 3 in x + (x + 4)
```

Conclusion

Et si sans effet de bord (par exemple print)

- Variables non-utilisées

```
let x = 3 + 5 in 1
```

- Variables utilisées une seule fois

```
let x = 3+4 in x + 5
```