

Machines Virtuelles – MV6

Séance 1

Benoît Valiron
Université Paris Diderot
2015

Organisation du cours

Intervenants	Benoît Valiron (cours + TD/TP gr. 1) Michele Pagani (TD/TP gr. 2)
Horaires	Mardi 9h30 En pratique : TP machine : 3h ; cours et TD : 2h.
Salles	Cours en SG-2035 (ici !) TD/TP gr. 1 en SG-2032 TD/TP gr. 2 en SG-2001
Page du cours	DIDEL : MV62015 http://www.monoidal.net/cours/2014/mv6.html
Évaluation	1/3 projet + 2/3 examen (sur feuille)

Emploi du temps

20 Jan	Cours	17 Mar	Cours
27 Jan	Cours	24 Mar	TP
3 Fév	TD	31 Mar	Cours
10 Fév	Cours	7 Avr	
17 Fév	TP	14 Avr	TP
24 Fév		...	
3 Mar	Cours	...	
10 Mar	TP	5 Mai	Cours

Bibliographie

Cours atypique, pas de livre « tout-en-un »

- Virtual Machines : Versatile Platforms for Systems and Processes (Smith, Nair, 2005)
- Développement d'applications avec Objective Caml (Chailloux, Manoury, Pagano, 2000) ^a
- Caml Virtual Machine – File and data format (Clerc, 2007) ^b
- Caml Virtual Machine — Instruction set (Clerc, 2010) ^c
- Java and the Java Virtual Machine : Definition, Verification, Validation (Stärk, Schmid, Börger, 2001)
- The Java Virtual Machine (Meyer, Downing, Shulmann, 1997)

a. <http://www.pps.univ-paris-diderot.fr/Livres/ora/DA-OCAML/>

b. <http://cadmium.x9c.fr/distrib/caml-formats.pdf>

c. <http://cadmium.x9c.fr/distrib/caml-instructions.pdf>

Machine Virtuelle : C'est Quoi ?

Définition informelle

Machine virtuelle.

L'implémentation d'une machine comme un programme prenant un programme et émulant son execution.

Hôte. Machine sur laquelle tourne la MV.

Invité. Machine émulée.

Une machine **dématérialisée, sans existence physique** : ni silicium, ni engrenage, mais un programme qui exécute un programme !

Objectifs

- Portabilité
- Sécurité
- Langages de haut niveau
- Émulation d'architectures
- Machines virtuelles système
- Multitâche
- Extension du processeur
- Cloud !
- ...

Objectifs

Choix du jeu d'instructions. On n'est plus lié au jeu d'instructions du processeur : émulation, code-octet. . .

Choix des structures. On peut introduire dans la machine des mécanismes inexistant sur l'hôte : ramasse-miette, typage, contrats, permissions. . .

Contrôle de l'exécution. La MV peut observer le programme avant de l'évaluer, sauver et restaurer son état : débogueur, virtualisation, "sandboxing".

Raisonnement sur les programmes. On peut s'abstraire des détails de l'électronique : un cadre formel et universel pour comprendre, i.e. prouver des propriétés sur l'évaluation

La MV comme donnée. Comme tout programme, la MV elle-même peut être téléchargée, mise à jour. . .

Exemples : Il y en a partout !

Virtualisation



Émulation



Cloud computing



Mobile computing



Langages de haut niveau



Définition formelle

Une **machine** est un couple (S, exec) avec

- S un ensemble d'états : mémoire, registres, ...
- exec une fonction de transition

$$\text{exec} : S \longrightarrow S$$

(par ex. execution d'une instruction).

Une **machine virtuelle (MV)** est composée de

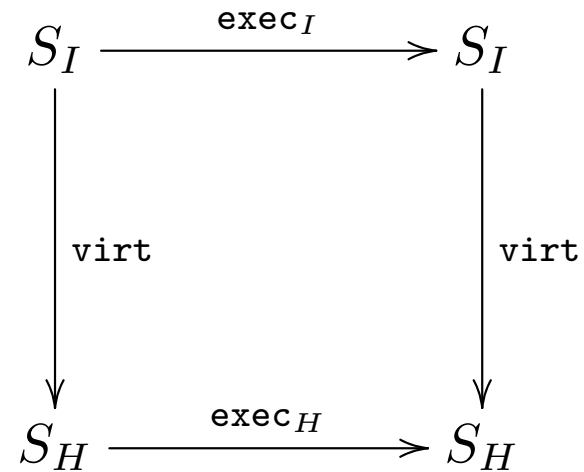
- deux machines (S_H, exec_H) et (S_I, exec_I) ,
- une fonction de virtualisation

$$\text{virt} : S_I \longrightarrow S_H$$

associant à chaque état de l'invité un état de l'hôte.

Définition formelle

Une MV doit vérifier une propriété de correction :



Un programme comme un autre ?

Un programme, pas forcément très gros, et un code-octet.

```
$ ls -lh
total 16M
-rw-r--r-- 1 benoit benoit 16M Jan 16 11:23 codex.um
-rw-r--r-- 1 benoit benoit 1.5K Jan 16 13:25 vm.c
$ wc -l vm.c
55 vm.c
$ gcc -c vm vm.c
$ ./vm codex.um
```

(exemple tiré de <http://www.boundvariable.org/index.shtml>)

```

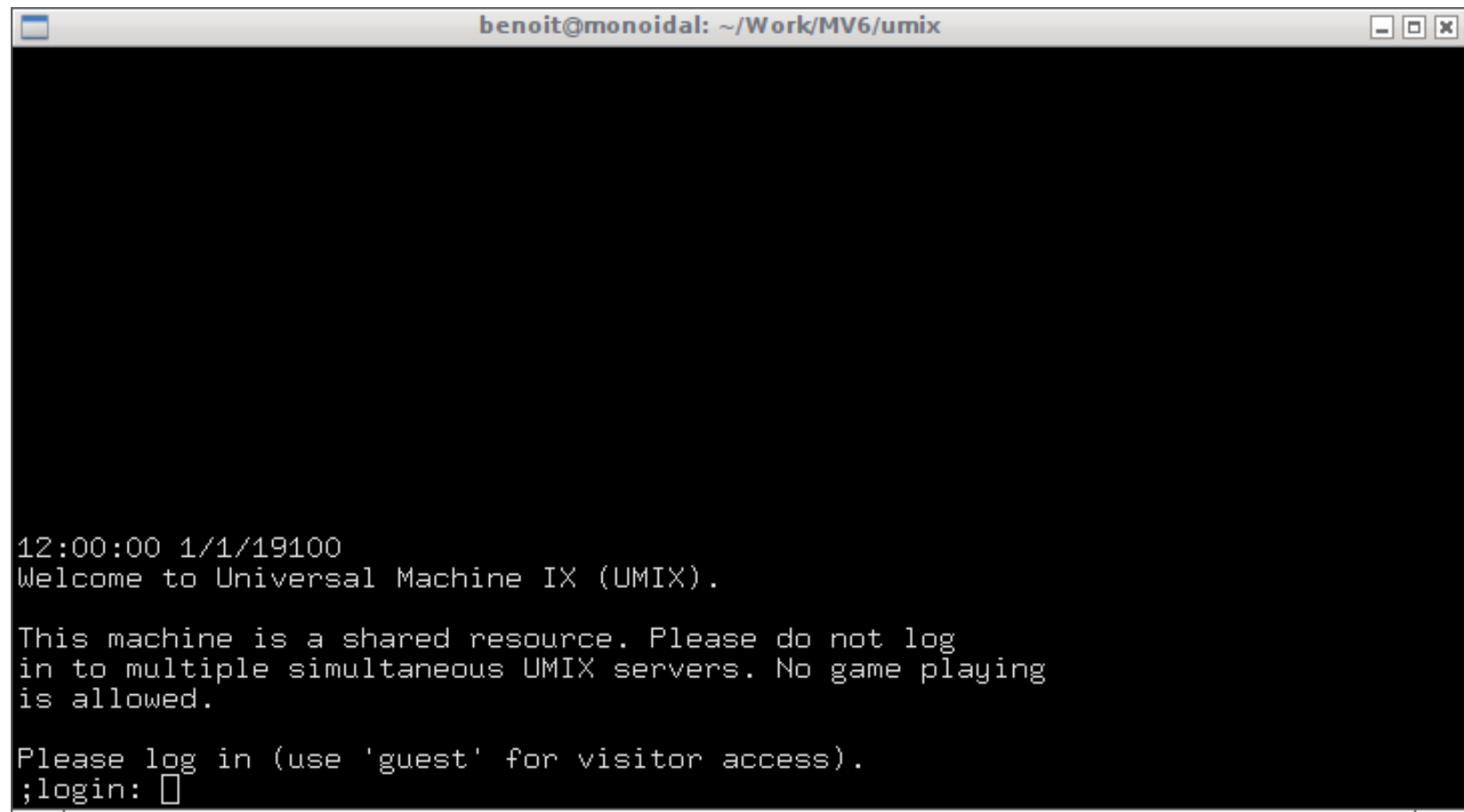
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#define arr(m) (m?(uint*)m:zero)
#define C w & 7
#define B (w >> 3) & 7
#define A (w >> 6) & 7
typedef unsigned int uint;
static uint * ulloc(uint size) {
    uint * r = (uint*)calloc((1+size),4);
    *r = size;
    return (r + 1);
}
int main (int argc, char ** argv) {
    static uint reg[8], ip, * zero;
    FILE * f = fopen(argv[1], "rb");
    if (!f) return -1;
    struct stat buf;
    if (stat(argv[1], &buf)) return -1;
    else zero = ulloc(buf.st_size >> 2);
    int a, n = 4, i = 0;
    while(EOF != (a = fgetc(f))) {
        if (!n--) { i++; n = 3; }
        zero[i] = (zero[i] << 8) | a;
    }
    fclose(f);

```

```

for(;;) {
    uint w = zero[ip++];
    switch(w >> 28) {
        case 0: if (reg[C]) reg[A] = reg[B]; break;
        case 1: reg[A] = arr(reg[B])[reg[C]]; break;
        case 2: arr(reg[A])[reg[B]] = reg[C]; break;
        case 3: reg[A] = reg[B] + reg[C]; break;
        case 4: reg[A] = reg[B] * reg[C]; break;
        case 5: reg[A] = reg[B] / reg[C]; break;
        case 6: reg[A] = ~(reg[B] & reg[C]); break;
        case 7: return 0;
        case 8: reg[B] = (uint)ulloc(reg[C]); break;
        case 9: free(-1 + (uint*)reg[C]); break;
        case 10: putchar(reg[C]); break;
        case 11: reg[C] = getchar(); break;
        case 12:
            if (reg[B]) {
                free(zero - 1);
                int size = ((uint*)reg[B])[-1];
                zero = ulloc(size);
                memcpy(zero, (uint*)reg[B], size * 4);
            }
            ip = reg[C];
            break;
        case 13: reg[7 & (w >> 25)] = w & 0177777777;
    }
}

```

A terminal window with a title bar that reads "benoit@monoidal: ~/Work/MV6/umix". The terminal content shows a login prompt for "Universal Machine IX (UMIX)". It displays the time "12:00:00 1/1/19100", a welcome message, a warning about shared resources, and a login prompt with a cursor.

```
benoit@monoidal: ~/Work/MV6/umix

12:00:00 1/1/19100
Welcome to Universal Machine IX (UMIX).

This machine is a shared resource. Please do not log
in to multiple simultaneous UMX servers. No game playing
is allowed.

Please log in (use 'guest' for visitor access).
;login: 
```

```
benoit@monoidal: ~/Work/MV6/umix

12:00:00 1/1/19100
Welcome to Universal Machine IX (UMIX).

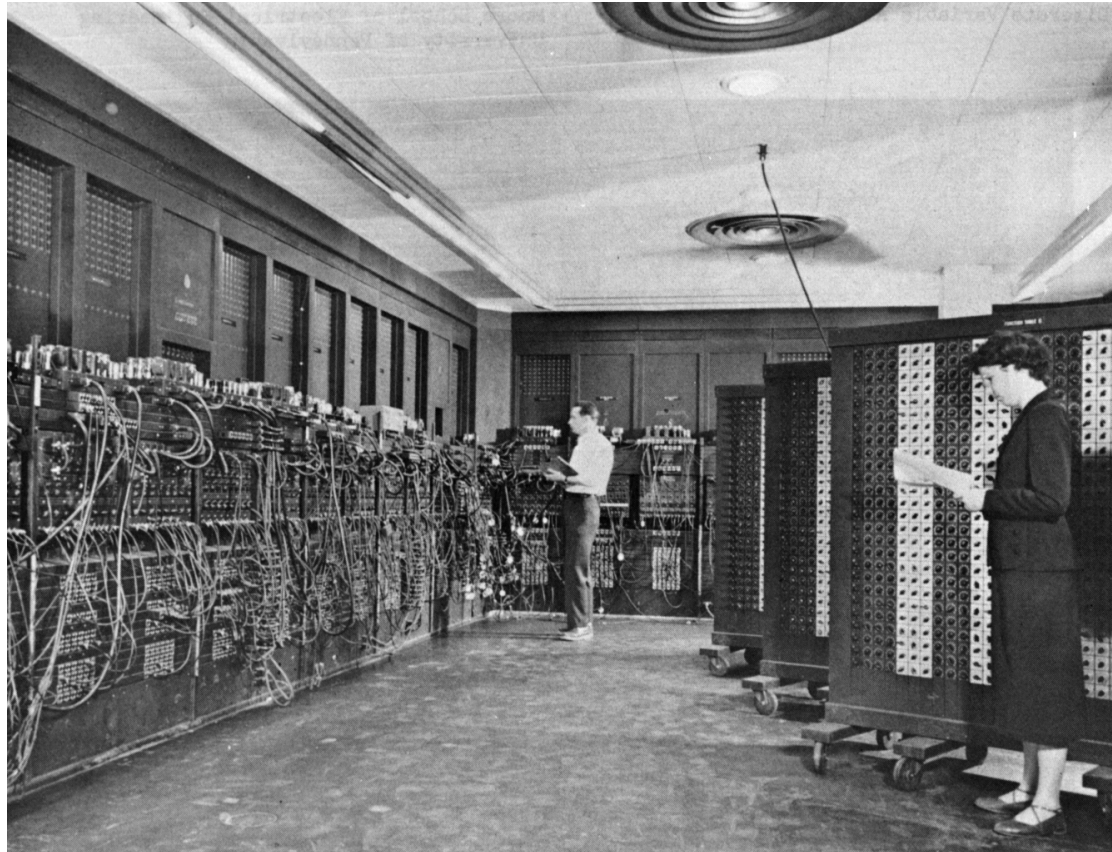
This machine is a shared resource. Please do not log
in to multiple simultaneous UMIK servers. No game playing
is allowed.

Please log in (use 'guest' for visitor access).
;login: guest
logged in as guest
INTRO.LOG=200@2476|16089283a41e17be043af803c7c6306

You have new mail. Type 'mail' to view.
% 
```

Des programmes portables

Modèle de Harvard : données et instructions sont dans des mémoires séparées. Les instructions ne peuvent être qu'exécutées.



Des programmes portables

Modèle de Von Neumann : les instructions sont stockées en mémoire, à côté des données. **Le programme est une donnée.**

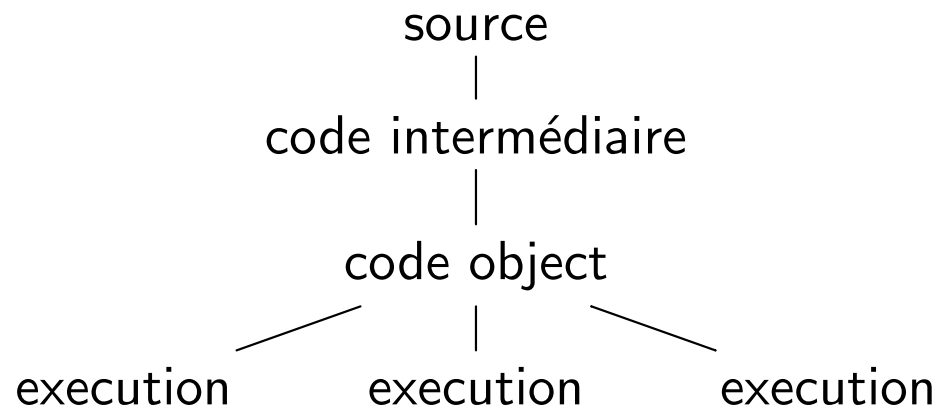
Conséquences capitales :

- une telle machine a plus d'une utilité
- on peut charger un programme depuis un support physique
- télécharger un programme ou une mise à jour puis l'exécuter
- un programme peut générer ou modifier un autre programme (compilateur)
- analyser son code (antivirus, analyses statiques)
- **lire et interpréter les instructions** de son code

Des programmes portables

Sans machine virtuelle :

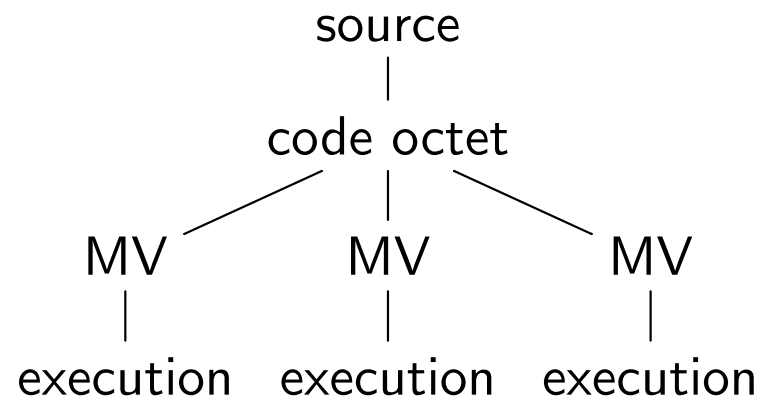
- Un compilateur classique génère du code objet ou natif pour une architecture physique donnée (x86, PPC, MIPS...)
- n architectures prises en charge
- n executables à distribuer.



Des programmes portables

Avec machine virtuelle :

- Ocamlc/Javac génèrent du code-octet pour une MV (Ocamlrun/Java), qui l'interprète ou le traduit en code natif
- Un seul executable distribué
- n portages de la MV



Interprétation et compilation

La machine virtuelle comme un compromis entre interprétation et compilation :

Interprétation. L'exécution se fait au fur et à mesure de l'analyse/sans pré-traitement du code source

Compilation. Traduction du code source d'un programme en « langage machine » (instructions processeur)

Avec une machine virtuelle. Compilation du code source en un langage machine (de plus haut niveau), puis interprétation par une MV

Remarque. La distinction n'est pas si nette :

- même les interprètes travaillent sur une forme pré-traitée du code source (arbre de syntaxe abstraite, que nous verrons plus loin)
- les « langages interprétés » (Python, Javascript) sont souvent à MV
- les instructions processeur sont compilées en un langage de plus bas niveau (microcode)

Interprétation et compilation

Avec une machine virtuelle, le schéma est le suivant.

- Le code source est **compilé** vers du code-octet
 - non-divulgateur du source, optimisations possibles
- Le code-octet est **interprété** par la MV
 - potentiels problèmes d'efficacité
- Il est une donnée, et peut être analysé pour garantir des propriétés sur l'exécution
- L'implémentation de MVs **simples**, **efficaces** et **sûres** est **non-trivial**

C'est l'objectif de ce cours.

Dans ce cours

On apprendra à concevoir et implémenter des machines virtuelles :

- coder/décoder des instructions en code-octet (assembler/désassembler)
- comprendre les machines à pile
- savoir compiler des expressions vers du code-octet
- traiter les appels de fonctions et de méthodes

Deux études de cas :

- OCamlrun, La machine virtuelle de OCaml
- JVM, la machine virtuelle de Java

Spoiler alert : L'étude des machines virtuelles n'est qu'une excuse pour introduire la compilation dans un cadre simple.

Planning des séances

(peut évoluer)

20 Jan	Cours : Introduction ; Machines à a-pile
27 Jan	Cours : Le langage Myrte
3 Fév	TD : Arbre de syntaxe (AST) d'un langage
10 Fév	Cours : MV pour Myrte
17 Fév	TP
24 Fév	
3 Mar	Cours : MV Ocaml, fragment arithmétique
10 Mar	TP
17 Mar	Cours : MV Ocaml, encodage des fonctions
24 Mar	TP
31 Mar	Cours : Survol de la MV Java
7 Avr	
14 Avr	TP
...	
5 Mai	Cours : séance de révision

Une machine à a-pile

Code natif

Tout d'abord, le **code natif** correspond à la machine physique :

- Le code natif est une liste contigue d'instructions **pour le processeur**, stockée en RAM
- Chaque instruction est un **code binaire** qui modifie l'état de la mémoire (RAM + registres)
- L'assembleur est une **syntaxe compréhensible** pour ce code
- Un registre spécial, PC (program count) stocke l'adresse de la prochaine instruction à exécuter
- Le processeur implémente le **cycle fetch-decode-execute** :
 - fetch** charge le code de la prochaine instruction, incrémente PC
 - decode** décode l'instruction chargée (e.g. 0x43AB → « ajouter 2 au contenu du registre 3 »)
 - execute** réalise l'opération décodée
- Le jeu d'instruction et leur codage **dépend du processeur** et constitue l'ISA (instruction set architecture)

Code-octet

Le code-octet des MV est un **code binaire portable**, approximation du code natif. Sa proximité avec le code natif réduit le travail de la MV (la couche d'interprétation) :

- c'est une liste d'instructions
- elles codent des opération sur la mémoire
- le jeu d'instruction constitue la MV
- elles sont souvent moins nombreuses mais de plus haut niveau que sur un processeur
- le modèle de mémoire peut être de plus haut niveau (pile, ramasse-miette...)

Interpréter du code-octet est **plus efficace qu'interpréter le code source directement**. La MV peut même compiler le code-octet en code machine à la volée (**compilation just-in-time**)

Modèles de machines

Modèles concrets :

- machine à pile (e.g. Postscript, LISP)
- machines à registres (RAM, RASP pointeurs)

En général, une machine à pile a quelques registres (machines à a-pile)

Modèles plus abstrait :

- machines de Turing
- systèmes de réécriture de termes

Une machine à a-pile

Les **mots** sont des entiers. Un **état de la machine** est constitué de :

- 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 empile le contenu de A sur S

pop dépile la tête n de S

consti n remplace le contenu de A par n

addi lit la tête n de S , remplace A par $A + n$

andi lit la tête n de S , remplace A par 0 si $A = n = 0$, sinon par 1

eqi lit la tête n de S , remplace A par 1 si $n = A$, sinon par 0

À chaque fois, on incrémente PC de 1 unité.

Une machine à a-pile

Codage possible des instructions de notre MV à a-pile

- une instruction par octet
- pour chaque 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 0 (000) Addi \mapsto 1 (001) Eqi \mapsto 2 (010)
 - Andi \mapsto 3 (011) Consti \mapsto 4 (100) Pop \mapsto 5 (101)

Limitations

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

Une machine à a-pile

00010100	00010 100	Consti 2
00000000	00000 000	Push
00000001	00000 001	Addi
00000101	00000 101	Pop

Évaluation : on commence avec une pile vide et PC=0.

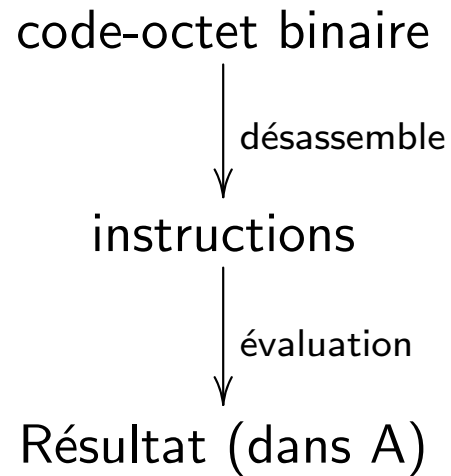
	Instruction lue	Modifications :	PC	A	S
<i>Début</i>	-		0	-	nil
	0 (Const 2)		1	2	nil
	1 (Push)		2	2	2 : nil
	2 (Addi)		3	4	2 : nil
	3 (Pop)		4	4	nil
<i>Fin</i>					

On s'arrête car le PC ne pointe plus sur une instruction.

Le résultat se lit dans $A = 4$.

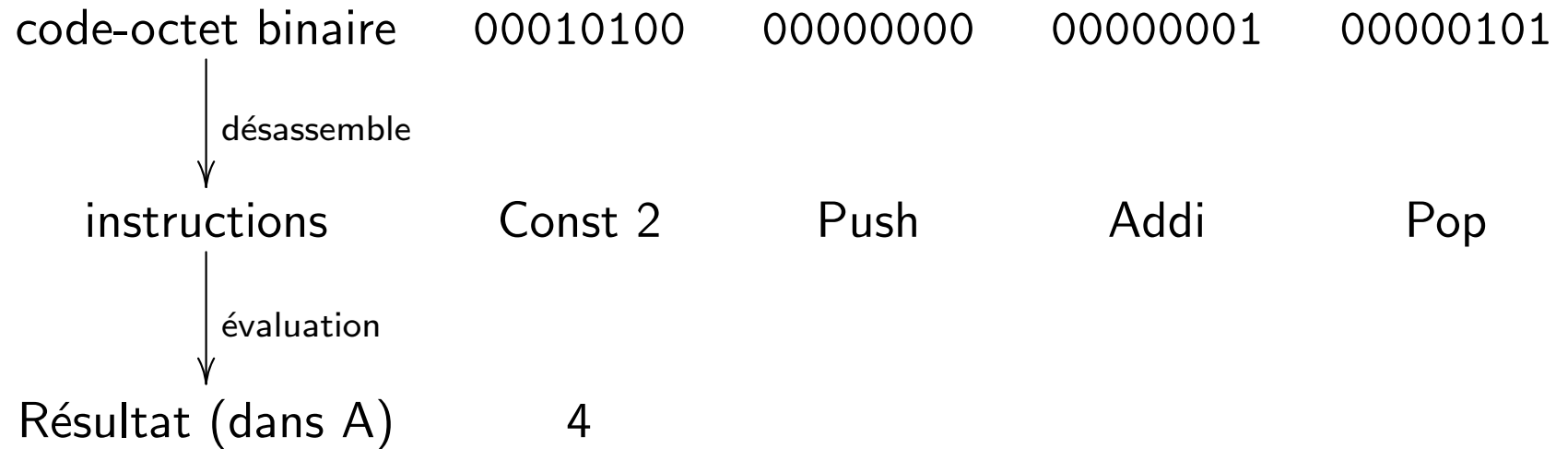
Une machine à a-pile

La machine virtuelle est un programme : prend pour argument le tableau d'instructions C sous la forme du code-octet :



Une machine à a-pile

Sur l'exemple, le schéma de fonctionnement de la machine est



Notion d'assembleur

Définitions

L'assembleur encode les instructions en code-octet

Le désassembleur décode du code-octet en instructions

Remarque

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

Propriété

Ces deux fonctions forment une bijection :

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

Exemples

- Consti 2 ; Push ; Consti 3 ; Push ; Consti 4 ; Addi ; Pop ; Addi ; Pop
- Consti 2 ; Push ; Addi ; Pop ; Pop
- Consti 2 ; Consti 3 ; Push ; Push ; Addi ; Pop ; Addi ; Pop
- Consti 2 ; Push ; Const 3 ; Push ; Addi ; Addi
- Consti 2 ; Push ; Addi ; Push ; Addi ; Push ; Consti3 ; Addi
- Consti 2 ; Push ; Addi ; Consti 3 ; Addi

Codage de la MV

D'abord, souvenez-vous comment encoder une liste avec un tableau :

- un tableau de N elements
- un indice sp qui indique la tête de la liste

Donc :

$1 : 2 : 3 : \text{nil}$ \longrightarrow ($sp = 2$,

3	2	1	-	-
---	---	---	---	---

)

$3 : \text{nil}$ \longrightarrow ($sp = 0$,

3	-	-	-	-
---	---	---	---	---

)

nil \longrightarrow ($sp = -1$,

-	-	-	-	-
---	---	---	---	---

)

Et

- Ajouter un élément = incrémenter sp
- Supprimer un élément = décrémenter sp

Codage de la MV

```
type instr =  
  Push | Consti of int | Addi | Eqi | Andi | Pop
```

```
type state = {  
  mutable acc: int;  
          code: instr array;  
  mutable pc: int;  
          stack: int array;  
  mutable sp: int;  
}
```

```
let init c =  
  { code = c;  
    stack = Array.make 1000 42;  
    pc = 0;  
    sp = -1;  
    acc = 52 }
```

Codage de la MV

```
let machine s =  
  while s.pc < Array.length s.code do  
    begin match s.code.(s.pc) with  
    | Consti n ->  
      s.acc <- n  
    | Push ->  
      s.sp <- s.sp + 1;  
      s.stack.(s.sp) <- s.acc  
    | Addi ->  
      s.acc <- s.stack.(s.sp) + s.acc;  
    | Andi ->  
      s.acc <- s.stack.(s.sp) * s.acc;  
    | Eqi ->  
      s.acc <- if s.stack.(s.sp) = s.acc then 1 else 0;  
    | Pop ->  
      s.sp <- s.sp-1  
    end;  
    s.pc <- s.pc + 1  
  done; s
```

Codage de la MV

Préliminaire : fonction lsl :

`int -> int -> int = <fun>`

- `1 lsl 0 = 1`
- `1 lsl 1 = 2`
- `1 lsl 2 = 4`
- `1 lsl 3 = 8`
- `2 lsl 2 = 8`
- `3 lsl 3 = ?`

Codage de la MV

```
let assemble (p : instr array) : string =  
  let s = String.make (Array.length p) 'z' in  
  for i = 0 to Array.length p - 1 do  
    s.[i] <- match p.(i) with  
    | Push -> Char.chr 0  
    | Addi -> Char.chr 1  
    | Eqi -> Char.chr 2  
    | Andi -> Char.chr 3  
    | Consti n -> assert (n < 32); Char.chr (4 + n lsl 3)  
    | Pop -> Char.chr 5;  
  done; s
```

Codage de la MV

```
let disassemble (s : string) : instr array =  
  let p = Array.make (String.length s) Push in  
  for i = 0 to String.length s - 1 do  
    p.(i) <- match Char.code s.[i] with  
    | 0 -> Push  
    | 1 -> Addi  
    | 2 -> Eqi  
    | 3 -> Andi  
    | n when (n mod 8 = 4) -> Consti (n lsr 3)  
    | 5 -> Pop  
    | _ -> failwith "invalid byte-code"  
  done; p
```