

Machines virtuelles

Cours de Compilation Avancée (MI190)

Benjamin Canou
Université Pierre et Marie Curie

Année 2012/2013 – Semaine 2

Machines virtuelles

Principe général

Machine A

- ▶ Langage compris : L_A
- ▶ Implantation : I_A

Machine B

- ▶ Langage compris : L_B
- ▶ Implantation : I_B

J'ai dans ma poche :

- ▶ un programme en langage L_A
- ▶ une machine de type B

Que faire ?

Principe général

Machine A

- ▶ Langage compris : L_A
- ▶ Implantation : $I_A = L_B$

Machine B

- ▶ Langage compris : L_B
- ▶ Implantation : $I_B = \Phi$

J'ai dans ma poche :

- ▶ un programme en langage L_A
- ▶ une machine de type B

Que faire ?

1. Un compilateur $L_A \rightarrow L_B$:

- ▶ Un programme écrit en langage L_B ,
- ▶ transformant mon programme en un équivalent en L_B .

2. Une machine virtuelle A pour ma machine B :

- ▶ Un programme écrit en langage L_B ,
- ▶ capable d'exécuter les programmes en langage L_A .

Machine virtuelle de plate-forme

(Ce n'est pas le sujet de ce cours)

Dans le cas général, il est trop difficile de recompiler.
Une machine virtuelle est donc la seule possibilité.

Machine PPC

- ▶ Langage compris : asm PPC
- ▶ Implantation : asm x86

Machine x64

- ▶ Langage compris : asm x64
- ▶ Implantation : Φ

- ▶ **Autres noms** : émulateur, simulateur, ...
- ▶ **Exemples** : QEMU, DOSBox, VirtualPC, ...

Machine virtuelle applicative

Machine ZAM (ocaml)

- ▶ Langage compris : asm ZAM
- ▶ Implantation : asm x86

Machine x64

- ▶ Langage compris : asm x64
- ▶ Implantation : Φ

Dans ce cas, le choix est délibéré :

1. On veut compiler un langage donné.
2. On préfère compiler vers un assembleur adapté.
3. On utilise une machine virtuelle pour l'exécuter.

QUIZZ : Pourquoi ?

Machine virtuelle applicative

Machine ZAM (ocaml)

- ▶ Langage compris : asm ZAM
- ▶ Implantation : asm x86

Machine x64

- ▶ Langage compris : asm x64
- ▶ Implantation : Φ

Dans ce cas, le choix est délibéré :

1. On veut compiler un langage donné.
2. On préfère compiler vers un assembleur adapté.
3. On utilise une machine virtuelle pour l'exécuter.

Mots-clefs : *abstraction, portabilité, sécurité, inter-opérabilité*

Machine virtuelle applicative : **portabilité**

Exemples d'implantations de la machine virtuelle OCaml :

- ▶ **ocamlrun** : écrite en C
portable partout où un compilateur C est disponible
- ▶ **obrowser** : écrite en JavaScript
on peut exécuter un programme caml dans un navigateur
- ▶ **ocapic** : écrite en assembleur PIC
un langage de haut niveau sur microcontrolleurs

Implantations alternatives :

- ▶ **OpenJDK** : pour la JVM d'oracle
- ▶ **Mono** : pour la CLR

Machine virtuelle applicative : **abstraction**

- ▶ **Modèle sémantique clair et figé :**
 - ▶ plus facile de théoriser,
 - ▶ exécutables plus durables,
 - ▶ portabilité facile, y compris aux tiers.
- ▶ **Instructions de haut niveau :**
 - ▶ moins d'étapes de compilation,
 - ▶ support du langage → compilation plus simple,
 - ▶ schéma de compilation unique.

Machine virtuelle applicative : **inter-opérabilité**

- ▶ **Entre les langages :**
VB.Net peut appeler des fonctions F# dans la CLR.
- ▶ **Entre les plate-formes :**
représentation spécifiée des chaînes, taille des entiers, etc.
(ex : Sauvegarde sous Win/x86, relecture sous GNU/PPC).
- ▶ **Entre les machines :**
primitives réseau spécifiées \Rightarrow communication plus facile

Machine virtuelle applicative : **sécurité**

- ▶ Exécution isolée (*sandboxing*)
- ▶ Assembleur typé
- ▶ Vérification avant exécution (*bytecode verifier*)
- ▶ Instrumentation (traces, journalisation, etc.)

Machines mono-paradigme, quelques exemples

- ▶ Langages procéduraux
p-machine (Pascal)
- ▶ Machines impératives bas-niveau :
GNU lightning, LLVM
- ▶ Langages fonctionnels (λ -calcul)
 - ▶ Évaluation stricte (comme en ML) : *SECD*, *FAM*, *CAM*
 - ▶ Évaluation paresseuse (comme en haskell) : *K*, *SK*, *G*-machine
- ▶ Concurrence (π -calcul, join-calcul)
Erlang-VM, CHAM
- ▶ Objets
 - ▶ Prototypes : Smalltalk (Smalltalk), Tamarin, SM (JavaScript)
 - ▶ Classes : JVM

Machines multi-paradigmes, quelques exemples

- ▶ Machines à objets étendues : JVM (Java), CLR (.Net)
- ▶ Machines fonctionnelles étendues : ZAM2 (OCaml)
- ▶ Machine hypothétique : Parrot (Perl 6)
- ▶ Machines impératives bas-niveau : GNU lightning, LLVM

Types de machines

- ▶ Machines à pile : JVM, ZAM2

- ▶ Machines à registres : Dalvik, LLVM, Parrot

Types de machines

- ▶ Machines à pile : JVM, ZAM2
 - ▶ Pile pour les variables et arguments

- ▶ Machines à registres : Dalvik, LLVM, Parrot
 - ▶ Ensemble de registres pour les variables et arguments

Types de machines

- ▶ Machines à pile : JVM, ZAM2
 - ▶ Pile pour les variables et arguments
 - ▶ → bruit pour accéder aux arguments
`acc 1 ; push ; acc 2 ; push ; add`
- ▶ Machines à registres : Dalvik, LLVM, Parrot
 - ▶ Ensemble de registres pour les variables et arguments
 - ▶ → plus gros opcodes
`add r1 r2 r0`

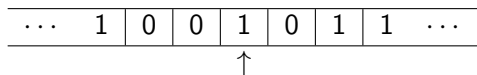
Types de machines

- ▶ Machines à pile : JVM, ZAM2
 - ▶ Pile pour les variables et arguments
 - ▶ → bruit pour accéder aux arguments
`acc 1 ; push ; acc 2 ; push ; add`
 - ▶ → triche : variables (JVM)
- ▶ Machines à registres : Dalvik, LLVM, Parrot
 - ▶ Ensemble de registres pour les variables et arguments
 - ▶ → plus gros opcodes
`add r1 r2 r0`
 - ▶ → triche : pile d'appels (Dalvik) (registres fixes/frame)

Une machine impérative

La machine de Turing

- ▶ Bande infinie de cases.
- ▶ Chaque case \in alphabet fini (ex. 1 ou 0).
- ▶ Une tête de lecture pointe sur une case précise.



- ▶ État \in ensemble fini (avec états spéciaux départ et fin).
- ▶ Table de transitions de la forme :

(état courant, valeur de la case)



(état suivant, nouvelle valeur, direction \in {gauche, droite})

La machine de Turing

- ▶ Capable d'encoder n'importe quelle fonction calculable.
- ▶ Arrêt indécidable.
- ▶ Capable de *simuler* un ordinateur moderne.
- ▶ Deux façons de voir :
 1. bande = mémoire, automate = code
 2. bande = code et mémoire mélangés, automate = processeur
- ▶ Insuffisante pour encoder la concurrence.

Programmation fonctionnelle (rappels)

Modèle des langages fonctionnels : le λ -calcul

Trois possibilités pour un terme T :

1. Variable : x
2. Application : $T_1 T_2$
3. Abstraction : $\lambda x. T$

Très simple mais \equiv à une machine de Turing.

Évaluation du λ -calcul

Évaluation formelle : β -réduction :

1. On choisit un redex $(\lambda x. T_1) T_2$ dans l'expression,
2. on remplace x par T_2 dans T_1 ,
3. on remplace le redex par ce résultat.
4. **Normalisation** : on continue tant qu'il y a des redexes.

Évaluation du λ -calcul

Stratégies d'évaluation :

► Appel par nom :

1. On remplace le paramètre par l'argument dans le corps,
2. on réduit le corps ainsi modifié.

► Appel par valeur :

1. On réduit l'argument,
2. on remplace le paramètre par l'argument réduit dans le corps,
3. on réduit le corps.

► Appel par nécessité :

1. On transforme l'argument en une fonction (*glaçon*),
2. la première fois ou l'argument est utilisé, la fonction le calcule,
3. les fois suivantes, il redonne la valeur déjà calculée.

Extensions du λ -calcul

Par encodage (ex: les couples) :

- ▶ Construction : $CONS := \lambda x.\lambda y.(\lambda f.f x y)$
- ▶ Projection 0 : $P0 := \lambda c.c (\lambda a.\lambda b.a)$
- ▶ Projection 1 : $P1 := \lambda c.c (\lambda a.\lambda b.b)$
- ▶ Échange : $SWAP := \lambda c.c (\lambda x.\lambda y.CONST y x)$

Par ajout de termes/opérations de base (ex: entiers) :

- ▶ $val ::= var \mid int \mid add \mid sub$
- ▶ $term ::= \lambda var.term \mid term term \mid val$
- ▶ Ex: $\lambda x.\lambda y.add x (sub y 3)$

Évaluation

Comment évaluer $CONS\ 1\ 2$ en pratique ?

- ▶ Réécriture de termes : $CONS\ 1\ 2 = \lambda f.f\ 1\ 2$
en pratique, difficile de modifier le code du programme.

- ▶ Fermetures :

$CONS\ 1\ 2$

→ $(\lambda x.\lambda y.\lambda f.f\ x\ y)_{[]} 1\ 2$

→ $(\lambda y.\lambda f.f\ x\ y)_{[(x,1)]} 2$

→ $(\lambda f.f\ x\ y)_{[(x,1);(y,2)]}$

On crée une **fermeture** :

- ▶ corps de la fonction,
- ▶ environnement : valeurs des variables lors de l'abstraction.

Lors de l'appel, on exécute le corps dans l'environnement, augmenté de la valeur du paramètre.

Exemple en OCaml

```
# let f x y z = x + y + z ;;
val f : int -> int -> int -> int = <fun>
# f 1 ;;
- : int -> int -> int = <fun>
# let g = f 1 2 ;;
val g : int -> int = <fun>
# g 10 ;;
- : int = 13
# g 10 20 ;;
Error: This function is applied to too many arguments;
maybe you forgot a `;'
#
```

Un Évaluateur de λ -calcul

Fabriquer une valeur calculable de la forme $\text{terme}_{\text{env}}$.

env	terme	pile	\rightarrow env	term	pile
e	$F A$	S	$\rightarrow e$	F	$A_e :: S$
e	$\lambda x. C$	$a :: S$	$\rightarrow (x, a) :: e$	C	S
$(= x, A_{e'}) :: e$	x	S	$\rightarrow e'$	A	S
$(\neq x, _) :: e$	x	S	$\rightarrow e$	x	S

Une machine fonctionnelle

La machine de Krivine

- ▶ Exécute du code-octet, compilé depuis un lambda terme,
- ▶ code-octet complètement linéaire (suite d'opcodes),
- ▶ trois opcodes très simples.

De quoi a-t'on besoin ?

env	terme	pile		env	term	pile
e	$F A$	S	\rightarrow	e	F	$A_e :: S$
e	$\lambda x. C$	$a :: S$	\rightarrow	$(x, a) :: e$	C	S
$(= x, A_{e'}) :: e$	x	S	\rightarrow	e'	A	S
$(\neq x, _) :: e$	x	S	\rightarrow	e	x	S

La machine de Krivine

- ▶ Exécute du code-octet, compilé depuis un lambda terme,
- ▶ code-octet complètement linéaire (suite d'opcodes),
- ▶ trois opcodes très simples.

De quoi a-t'on besoin ?

env	terme	pile		env	term	pile
e	FA	S	\rightarrow	e	F	$A_e :: S$
e	$\lambda x.C$	$a :: S$	\rightarrow	$(x, a) :: e$	C	S
$(= x, A_{e'}) :: e$	x	S	\rightarrow	e'	A	S
$(\neq x, _) :: e$	x	S	\rightarrow	e	x	S

1. *PUSH addr*

on repère les termes par l'adresse de leur code compilé

La machine de Krivine

- ▶ Exécute du code-octet, compilé depuis un lambda terme,
- ▶ code-octet complètement linéaire (suite d'opcodes),
- ▶ trois opcodes très simples.

De quoi a-t'on besoin ?

env	terme	pile		env	term	pile
e	FA	S	\rightarrow	e	F	$A_e :: S$
e	$\lambda x.C$	$a :: S$	\rightarrow	$(x, a) :: e$	C	S
$(= x, A_{e'}) :: e$	x	S	\rightarrow	e'	A	S
$(\neq x, _) :: e$	x	S	\rightarrow	e	x	S

1. **PUSH** *addr*

on repère les termes par l'adresse de leur code compilé

2. GRAB

La machine de Krivine

- ▶ Exécute du code-octet, compilé depuis un lambda terme,
- ▶ code-octet complètement linéaire (suite d'opcodes),
- ▶ trois opcodes très simples.

De quoi a-t'on besoin ?

env	terme	pile		env	term	pile
e	FA	S	\rightarrow	e	F	$A_e :: S$
e	$\lambda x.C$	$a :: S$	\rightarrow	$(x, a) :: e$	C	S
$(= x, A_{e'}) :: e$	x	S	\rightarrow	e'	A	S
$(\neq x, _) :: e$	x	S	\rightarrow	e	x	S

1. **PUSH** *addr*

on repère les termes par l'adresse de leur code compilé

2. **GRAB**

3. **ACCESS** *idx*

on repère les variables par leur indice de de Bruijn

Machine virtuelle

```
type closure = C of int * closure list

let interprete code =
  let rec interp env pc stack =
    match (nth code pc) with
    | ACCESS n ->
      begin try
        let (C (n,e)) = nth env n in
          interp !e n stack
        with ex -> (C (pc, ref env))
      | PUSH n ->
        interp env (pc+1) ((C (n,ref env))::stack)
      | GRAB ->
        begin match stack with
        | [] -> C (pc,ref env)
        | so::s -> interp (so::env) (pc+1) s)
    in
    interp [] 1 []
```

Compilation vers la machine de Krivine (exos en TD)

Assembleur avec étiquettes :

```
type instr =  
  | IPUSH of lbl  
  | IGRAB  
  | IACCESS of int  
  | ILABEL of lbl
```

Schéma de compilation \mathcal{C} :

$$\mathcal{C}_e(T_1 T_2) = \text{IPUSH } l ; \mathcal{C}_e(T_1) ; \text{ILABEL } l ; \mathcal{C}_e(T_2)$$

$$\mathcal{C}_e(\lambda x. T) = \text{IGRAB} ; \mathcal{C}_{x::e}(T)$$

$$\mathcal{C}_e(x) = \text{IACCESS } \text{nth}(x, e)$$

Puis on fait une passe de suppression des étiquettes.

La ZAM

La machine de Krivine est-elle utilisable en pratique ?

Oui, mais :

La ZAM

La machine de Krivine est-elle utilisable en pratique ?

Oui, mais : on ne peut pas utiliser l'appel par nom en pratique, si on utilise des opérations de base (opérations arithmétiques, etc).

1. Évaluation stricte (la ZAM : machine de Caml)
2. Évaluation paresseuse.

Implantation d'une VM en C

Interprète de bytecode : **boucle de base**

Flux d'entrée : opcodes simples et valeurs.

Plus courant dans les machines à registres.

Ex: [NOP ; GOTO ; 0]

```
void run(int code[]) {
    int pc = 0;
    while (TRUE) {
        switch (code[pc]) {
            case NOP:
                pc++;
                break;
            case GOTO:
                pc = code[pc + 1];
                break;
            /* ... */
        }
    }
}
```

Interprète de bytecode : **boucle de base**

Variante : arguments dans l'opcode, à décoder.

Plus courant dans les machines à registres.

Ex: [NOP ; GOTO(0)]

```
void run(int code[]) {
    int pc = 0;
    while (TRUE) {
        /* décodage */
        int op, arg0, arg1;
        decode (code[pc], &op, &arg0, &arg1);
        switch (op) {
            case NOP:
                pc++;
                break;
            case GOTO:
                pc = arg0;
                break;
            /* ... */
        }
    }
}
```


Interprète de bytecode : instructions

Exemple d'encodage :

OPCODE(8)	A ₁ (12)	A ₂ (12)
-----------	---------------------	---------------------

```
#define NOP    0x00
#define GOTO  0x01
/* ... */

void decode(int code, int *op, int *a0, int *a1) {
    *op = (code >> 24) & 0xFF ;
    *a0 = (code >> 12) & 0xFFF ;
    *a1 = (code) & 0xFFF ;
}
```

Autre possibilité : arguments variables pour chaque opcode

Interprète de bytecode : **pile**

Pile préallouée, vérifications de taille.

```
void run(int code[]) {
    int pc = 0;
    /* pila dans un tableau pré-alloué */
    int stack = malloc (MAX * sizeof (int));
    int sp = 0;
    while (TRUE) {
        switch (code[pc]) {
            case PUSHINT:
                stack[sp++] = code[pc + 1];
                if (sp > MAX) exit (1);
                pc += 2;
                break;
            /* ... */
        }
    }
}
```

Interprète de bytecode : registres

Table de registres.

Autre possibilité : variables (optimisées) pour certains registres.

```
void run(int code[]) {
    int pc = 0; /* program counter : indice de l'op en cours */
    /* tableau de registres */
    int regs[16];
    while (TRUE) {
        int op,a0,a1,a2;
        decode (code[pc],&op,&a0,&a1,&a2);
        switch (op) {
            case MOVE:
                regs[a2] = regs[a0] + regs[a1];
                pc++;
                break;
            /* ... */
        }
    }
}
```

Interprète de bytecode : **branchements**

On change seulement le pointeur de code.

On n'utilise pas les branchement du langage hôte.

```
case BRA_EQ_INT:
    int a = stack[sp - 1];
    int b = stack[sp - 2];
    sp -= 2;
    if (a == b) {
        /* on change le pc pour le prochain tour */
        pc = code[pc + 1];
    } else {
        pc += 2;
    }
    break;
```

Interprète de bytecode : appels

Exemple avec machine à registres.

On ajoute une pile d'appels (*frame stack*).

Paramètres dans les registres, retour dans r_0 .

```
int regs[16];
int cstack[MAX][16];
int csp = 0;

case CALL:
    /* sauvegarde registres et pc */
    memcpy(&cstack[rsp][1], &regs[1], 15 * sizeof(int));
    cstack[rsp][0] = pc + 1;
    if (++rsp > MAX) exit (2);
    /* jump */
    pc = a0;
    break;
case RETURN:
    /* resultat dans r0 */
    memcpy(&regs[1], &cstack[--rsp][1], 15 * sizeof(int));
    pc = cstack[rsp][0];
    break;
```

Une VM bas niveau pourrait laisser faire le compilateur.

Interprète de bytecode : **appels de primitives**

Il faut un mécanisme d'inter-opérabilité.

- ▶ OCaml : fonctions C + `mvalues.h`
- ▶ Java : JNI

Sur un exemple :

- ▶ Machine à registres.
- ▶ Instruction d'appel : `EXT_CALL(prim,nbargs)`.
- ▶ Passage de paramètres comme une procédure normale.

Interprète de bytecode : appels de primitives

Il faut une table de primitives :

```
int print_int(int v) ;
int read_int(void) ;
int add(int a, int b) ;
/* ... */

typedef int (*) () prim ;
prim prims [N] = {
    print_int,
    read_int,
    add,
    /* */
}
```

Interprète de bytecode : appels de primitives

```
EXT_CALL:
switch (a1 /* nb args */) {
case 0 :
    r0 = prims[a0]();
    break;
case 1 :
    r0 = prims[a0](regs[0]);
    break;
case 2 :
    r0 = prims[a0](regs[0], regs[1]);
    break;
/* ... */
}
pc++;
break;
```


Étude de code : ocamlrun