

# Implantation d'une machine virtuelle en C

## Cours de Compilation Avancée (MI190)

Benjamin Canou  
Université Pierre et Marie Curie

Année 2012/2013 – Semaine 3

# Interprète de bytecode

# 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 <sub>1</sub> (12)	A <sub>2</sub> (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.

- ▶ Pour effectuer les appels
- ▶ Pour convertir les valeurs entre les deux mondes
- ▶ Pour assurer la gestion mémoire

Exemple : **JNI**

```
char[] str = "TCHOU TCHOU";  
jstring jstr = (*env)->NewStringUTF(env, str);
```

# Interprète de bytecode : **appels de primitives**

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;
```

# Représentation des données

# Représentation uniforme

Nécessité de parcourir les valeurs :

- ▶ Fonctions primitives génériques : égalité, sérialisation, etc.
- ▶ Gestion mémoire (cf. cours prochain)
- ▶ Introspection, affichage générique, etc.

Solution logique : **uniformiser la structure des valeurs**

Question centrale : distinction entre

- ▶ Valeurs immédiates (entiers, caractères, etc.)
- ▶ Valeurs allouées (tableaux, structures, etc.)
- ▶ Différentes sortes de valeurs allouées.

En machine : **un pointeur = un entier = un mot machine**

# Représentation non uniforme

Il faut trouver l'information de type ailleurs que dans la donnée :

- ▶ Méta données issues du compilateur (structure de la pile, etc.)
- ▶ Algorithmes ambigus (c'est **peut-être** un pointeur)
- ▶ Mélange : informations dans les blocs, pas dans les immédiats



# Solution simple : tout est pointeur

Idée : valeurs immédiates stockées dans des valeurs allouées

```
typedef enum { BOOL_TAG, INT_TAG, PAIR_TAG } tag_t ;
struct value ;
typedef struct value {
    tag_t tag ;
    union {
        enum { FALSE, TRUE } as_bool ;
        int as_int ;
        struct value as_pair [2] ;
    } contents ;
} value_t ;
```

## Solution plus avancées

Bit(s) discriminant(s) :

- ▶ On mange un bit sur le mot machine pour discriminer entre entier et pointeur
- ▶ Éventuellement plus de bits pour plusieurs types d'immédiats
- ▶ On utilise un système de tags comme précédemment pour les valeurs allouées
- ▶ On limite l'étendue des immédiats

NaN boxing

- ▶ Les valeurs de base font 64 bits
- ▶ Les flottants sont stockés tels quels
- ▶ Les entiers et les pointeurs sont encodés dans l'espace des NaN
- ▶ On utilise 64 bits pour des immédiats de 32 bits
- ▶ On limite les pointeurs à 4 Go

## Exemple : la machine d'OCaml (pour changer)

```
typedef intnat value;
typedef uintnat header_t;
typedef uintnat msize_t;
typedef unsigned int tag_t;           /* Actually, an unsigned char */
typedef uintnat color_t;
typedef uintnat mark_t;

/* Longs vs blocks. */
#define Is_long(x)    (((x) & 1) != 0)
#define Is_block(x)  (((x) & 1) == 0)

/* Conversion macro names are always of the form "to_from". */
/* Example: Val_long as in "Val from long" or "Val of long". */
#define Val_long(x)    (((intnat)(x) << 1) + 1)
#define Long_val(x)    ((x) >> 1)
#define Max_long      (((intnat)1 << (8 * sizeof(value) - 2)) - 1)
#define Min_long      (-((intnat)1 << (8 * sizeof(value) - 2)))
#define Val_int(x)    Val_long(x)
#define Int_val(x)    ((int) Long_val(x))
#define Unsigned_long_val(x) ((uintnat)(x) >> 1)
#define Unsigned_int_val(x) ((int) Unsigned_long_val(x))
```

## Exemple : la machine d'OCaml

```
/* Structure of the header:
For 16-bit and 32-bit architectures:
    +-----+-----+-----+
    | wosize | color | tag |
    +-----+-----+-----+
bits  31      10 9      8 7  0
*/

#define Tag_hd(hd) ((tag_t) ((hd) & 0xFF))
#define Wosize_hd(hd) ((mlsize_t) ((hd) >> 10))

#define Hd_val(val) (((header_t *) (val)) [-1])      /* Also an l-value. */
#define Hd_op(op) (Hd_val (op))                    /* Also an l-value. */
#define Hd_bp(bp) (Hd_val (bp))                   /* Also an l-value. */
#define Hd_hp(hp) (* ((header_t *) (hp)))          /* Also an l-value. */
#define Hp_val(val) ((char *) (((header_t *) (val)) - 1))
#define Hp_op(op) (Hp_val (op))
#define Hp_bp(bp) (Hp_val (bp))
#define Val_op(op) ((value) (op))
#define Val_hp(hp) ((value) (((header_t *) (hp)) + 1))
#define Op_hp(hp) ((value *) Val_hp (hp))
#define Bp_hp(hp) ((char *) Val_hp (hp))
```

## Exemple : la machine d'OCaml

```
/* The lowest tag for blocks containing no value. */
#define No_scan_tag 251
/* Fields are numbered from 0. */
#define Field(x, i) (((value *) (x)) [i])           /* Also an l-value. */

/* Special case of tuples of fields: closures */
#define Closure_tag 247
#define Code_val(val) (((code_t *) (val)) [0])    /* Also an l-value. */

/* Booleans are integers 0 or 1 */
#define Val_bool(x) Val_int((x) != 0)
#define Bool_val(x) Int_val(x)
#define Val_false Val_int(0)
#define Val_true Val_int(1)
#define Val_not(x) (Val_false + Val_true - (x))
```

# Machines virtuelles fonctionnelles

# La ZAM : machine fonctionnelle stricte

Schéma dérivé de la machine de Krivine :

- ▶ Le corps d'une instruction commence par GRAB,
- ▶ comme les fonctions ont plusieurs arguments, le code ressemble en fait à : `[GRAB;  $n_{args}$ ; \dots ; RETURN]`
- ▶ les arguments sont passés sur la pile par les instructions `APPLY{1,2,3} + compteur extra_args`
- ▶ GRAB applique la fonction (évaluation stricte) si elle trouve les arguments nécessaires, sinon, elle crée une fermeture.

# La ZAM : application générale

Comment s'exécute le programme suivant ?

```
# open Printf;;

# let separe sep =
  let rec aux i str =
    if i < String.length str then (
      printf "%c%c" str.[i] sep ;
      aux (i + 1) str
    )
  in
  aux 0;;
val separe : char -> string -> unit = <fun>

# separe ',';;
- : string -> unit = <fun>

# separe ',' "toto";;
t,o,t,o,
- : unit = ()
```

Grâce à CLOSURE, APPLY, GRAB et RETURN



# La ZAM : CLOSURE

```
Instruct(CLOSURE): {  
    int nvars = *pc++;  
    int i;  
    if (nvars > 0) *--sp = accu;  
    Alloc_small(accu, 1 + nvars, Closure_tag);  
    Code_val(accu) = pc + *pc;  
    pc++;  
    for (i = 0; i < nvars; i++) Field(accu, i + 1) = sp[i];  
    sp += nvars;  
    Next;  
}
```

# La ZAM : APPLY

```
Instruct(APPLY2): {  
    value arg1 = sp[0];  
    value arg2 = sp[1];  
    sp -= 3;  
    sp[0] = arg1;  
    sp[1] = arg2;  
    sp[2] = (value)pc;  
    sp[3] = env;  
    sp[4] = Val_long(extra_args);  
    pc = Code_val(accum);  
    env = accum;  
    extra_args = 1;  
    goto check_stacks;  
}
```

# La ZAM : GRAB

```
Instruct(GRAB): {
    int required = *pc++;
    if (extra_args >= required) {
        extra_args -= required;
    } else {
        mlsize_t num_args, i;
        num_args = 1 + extra_args; /* arg1 + extra args */
        Alloc_small(accum, num_args + 2, Closure_tag);
        Field(accum, 1) = env;
        for (i = 0; i < num_args; i++) Field(accum, i + 2) = sp[i];
        Code_val(accum) = pc - 3; /* Point to the preceding RESTART instr. */
        sp += num_args;
        pc = (code_t)(sp[0]);
        env = sp[1];
        extra_args = Long_val(sp[2]);
        sp += 3;
    }
    Next;
}
```

RESTART effectue la copie environnement  $\rightarrow$  pile.

Compilation : ...RESTART; [GRAB;  $n_{args}$ ; ...;RETURN] ...

# La ZAM : RETURN

```
Instruct(RETURN): {
    sp += *pc++;
    if (extra_args > 0) {
        extra_args--;
        pc = Code_val(accum);
        env = accum;
    } else {
        pc = (code_t)(sp[0]);
        env = sp[1];
        extra_args = Long_val(sp[2]);
        sp += 3;
    }
    Next;
}
```

# Étude de code : appel de méthode