Implantation d'une machine virtuelle en C

Cours de Compilation Avancée (MU4IN504)

Benjamin Canou & Emmanuel Chailloux
Sorbonne Université

Année 2019/2020 – 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++;                        // instruction suivante
        break;
      case GOTO:
        pc = code[pc + 1];          // aller à l'adresse qui suit
        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)	$A_0(12)$	$A_1(12)$

```
#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;
  /* pile 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 branchements 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 rsp = 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) discrimant(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)

```
1  typedef intnat value;
2  typedef uintnat header_t;
3  typedef uintnat mlsize_t;
4  typedef unsigned int tag_t;                /* Actually, an ↩
       unsigned char */
5  typedef uintnat color_t;
6  typedef uintnat mark_t;
7
8  /* Longs vs blocks. */
9  #define Is_long(x)    (((x) & 1) != 0)
10 #define Is_block(x)   (((x) & 1) == 0)
11
12 /* Conversion macro names are always of the form  "to_from"*/
13 /* Example: Val_long as in "Val from long" or "Val of long"*/
14 #define Val_long(x)      (((intnat)(x) << 1) + 1)
15 #define Long_val(x)      ((x) >> 1)
16 #define Max_long (((intnat)1 << (8 * sizeof(value) - 2)) - 1)
17 #define Min_long (-((intnat)1 << (8 * sizeof(value) - 2)))
18 #define Val_int(x) Val_long(x)
19 #define Int_val(x) ((int) Long_val(x))
20 #define Unsigned_long_val(x) ((uintnat)(x) >> 1)
21 #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 (1)

Caractéristiques

- Une machine à pile légère et assez stable
- Seulement 148 instructions
- Gestion des exceptions par chaînage dans la pile

La ZAM : machine fonctionnelle stricte (2)

Schéma dérivé de la machine de Krivine

- Le corps d'une fonction attendant plusieurs paramètres commence par `GRAB` et est précédé de `RESTART`
- comme les fonctions ont plusieurs arguments, le code ressemble en fait à : `[GRAB;`n_{args}`;...;RETURN]`
- les arguments sont passés sur la pile par les instructions `APPLY{1,2,3}` + compteur `extra_args` indiquant le nombre d'arguments effectivement sur la pile
- `GRAB` applique la fonction (évaluation stricte) si elle trouve les arguments nécessaires, sinon, elle crée une fermeture pointant sur `RESTART`.
- `RETURN` vérifie qu'il n'y a pas d'argument non utilisé, et relance un appel sinon.

La ZAM : application générale

Comment s'exécute le programme suivant ?

```
1  # open Printf;;
2
3  # let separe sep =
4    let rec aux i str =
5      if i < String.length str then (
6        printf "%c%c" str.[i] sep ;
7        aux (i + 1) str
8      )
9    in
10   aux 0;;
11 val separe : char -> string -> unit = <fun>
12
13 # separe ',';;
14 - : string -> unit = <fun>
15
16 # separe ',' "toto";;
17 t,o,t,o,
18 - : unit = ()
```

Grâce à CLOSURE, APPLY, GRAB et RETURN

Bytecode du programme separe (1)

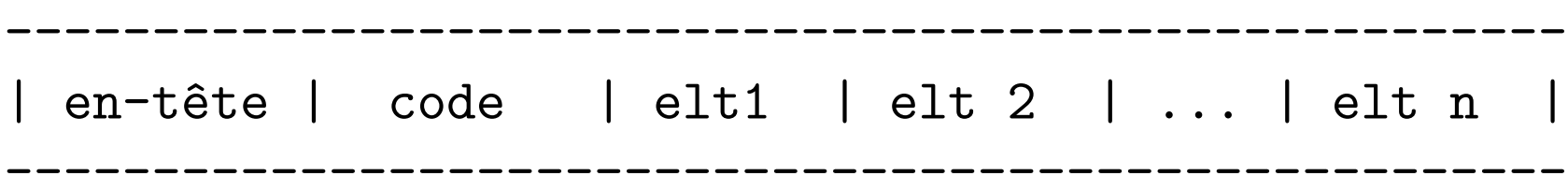
```
1           branch L2
2           restart
3   L3:     grab 1
4           acc 1
5           ccall caml_ml_string_length, 1
6           push
7           acc 1
8           ltint
9           strictbranchifnot L4
10          envacc 1
11          push
12          acc 1
13          push
14          acc 3
15          ccall caml_string_get, 2
16          push
17          const [0: [0: [0: 0a]] "%c%c"]
18          push
19          getglobal Printf!
20          getfield 1
21          apply 3
22          acc 1
```

Bytecode du programme separe (2)

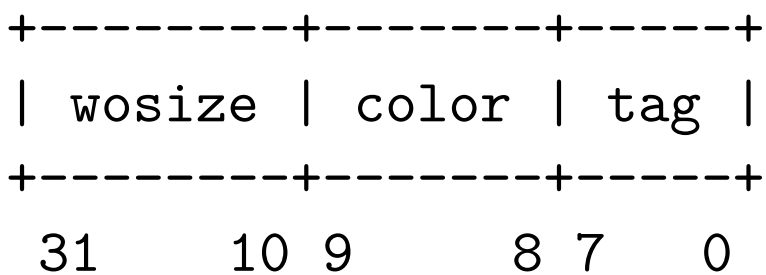
```
1           push
2           acc 1
3           offsetint 1
4           push
5           offsetclosure 0
6           appterm 2, 4
7   L4:     return 2
8   L1:     acc 0
9           closurerec 3, 1
10          const 0
11          push
12          acc 1
13          appterm 1, 3
14  L2:     closure L1, 0
15          push
16          const "toto"
17          push
18          const ','
19          push
20          acc 2
21          apply 2
22          acc 0
23          makeblock 1, 0
24          pop 1
25          setglobal Separe!
```

Représentation d'une fermeture

un bloc classique (c'est-à-dire avec un en-tête) :



avec un en-tête sur 32 bits :



le tag des fermetures est :

```
#define Closure_tag 247
```

La ZAM : CLOSURE

`CLOSURE` *n ofs* : Si $n > 0$ alors l'accu est empilé dans la pile. Une fermeture de $n + 1$ élémeents est crée dans l'accu, le code de la fermeture est *pc* + *ofs*. Les autres éléments de la fermeture sont alors dépilés de la pile.

```
1  Instruct(CLOSURE): {
2        int nvars = *pc++;
3        int i;
4        if (nvars > 0) *--sp = accu;
5        Alloc_small(accu, 1 + nvars, Closure_tag);
6        Code_val(accu) = pc + *pc;
7        pc++;
8        for (i = 0; i < nvars; i++) Field(accu, i + 1) = sp[i];
9        sp += nvars;
10       Next;
11     }
```

La ZAM : APPLY

APPLY2 : dépile les deux arguments de la pile, et empile extraArgs, env et pc puis rempiler les deux arguments. Alors pc est mis au code de la fermeture (accu), env à l'environnement de la fermeture, et extraArgs à 1.

```
1  Instruct(APPLY2): {
2        value arg1 = sp[0];
3        value arg2 = sp[1];
4        sp -= 3;
5        sp[0] = arg1;
6        sp[1] = arg2;
7        sp[2] = (value)pc;
8        sp[3] = env;
9        sp[4] = Val_long(extra_args);
10       pc = Code_val(accu);
11       env = accu;
12       extra_args = 1;
13       goto check_stacks;
14    }
```

La ZAM : GRAB

[GRAB; n_{args}; ...] : si extra_args $\geq$ n, alors extra_args est décrémenté de n. Sinon une fermeture est créé.

```
1  Instruct(GRAB): {
2   int required = *pc++;
3   if (extra_args >= required) {
4    extra_args -= required;
5   } else {
6    mlsize_t num_args, i;
7    num_args = 1 + extra_args; /* arg1 + extra args */
8    Alloc_small(accu, num_args + 2, Closure_tag);
9    Field(accu, 1) = env;
10   for (i = 0; i < num_args; i++) Field(accu, i + 2) = sp[i];
11   Code_val(accu) = pc - 3; // Point to the preceding RESTART
12   sp += num_args;
13   pc = (code_t)(sp[0]);
14   env = sp[1];
15   extra_args = Long_val(sp[2]);
16   sp += 3;
17  }
18  Next;
19 }
```

RESTART effectue la copie environnement $\rightarrow$ pile.

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

La ZAM : RETURN

`RETURN` *n* : dépile *n* éléments de la pile. Si `extraArgs` est strictement positif, alors il est décrémenté puis *pc* vaut le pointeur de code de la fermeture et l'envirnnement l'environnement de la fermeture. Sinon les 3 valeurs sont dépilées et assignées) pc, env et extraArgs.

```
1  Instruct(RETURN): {
2        sp += *pc++;
3        if (extra_args > 0) {
4          extra_args--;
5          pc = Code_val(accu);
6          env = accu;
7        } else {
8          pc = (code_t)(sp[0]);
9          env = sp[1];
10         extra_args = Long_val(sp[2]);
11         sp += 3;
12       }
13       Next;
14     }
```