Compilation (LI349)

Machine abstraite et génération de code

Emmanuel Chailloux

Plan du cours 11

1. Généralités sur les machines abstraites
2. Description de la ZAM
3. Compilation vers la ZAM
4. Organisation d'un évaluateur ZAM
   - Description des instructions ZAM
   - Code de la boucle d'interprétation

Machines abstraites

- but : s'abstraire des machines réelles :
  - enrichier le modèle de calcul (fonctions, objets, logique)
  - simplifier la gestion mémoire (GC)
  - portabilité des applications (même code sur différents processeurs)
  - interopérabilité entre langages (même AM cible)

modèle impératif : P-code

machine conçue pour compiler Pascal.

- caractéristiques
  - machine à pile
  - registres : SP (stack pointer), MP (stack frame), ... EP (plus haut niveau de pile d'une procédure) - NP (plus bas niveau du tas)
  - pile : procédure (stack - frame - adresse de retour) + arguments
  - tas (zone allocation dynamique)

plus récente : `LLVM` (Low Level Virtual Machine) pour le compilateur CLang (C, C++, Objective C) :

- instructions en SSA (static single assignment),
- JIT (Just in Time)
- `http://llvm.org/`

modèle fonctionnel : SECD (Landin)

- caractéristique
  - Stack (SP)
  - Env (E)
  - Code (PC)
  - Dump (liste de registres)
- programmation fonctionnelle
  - CLOSURE : création d'une valeur fonctionnelle
  - APPLY : application d'une valeur fonctionnelle

d'autres machines : CAM, FAM, G-machine, ...

modèle objet : JVM / CLR

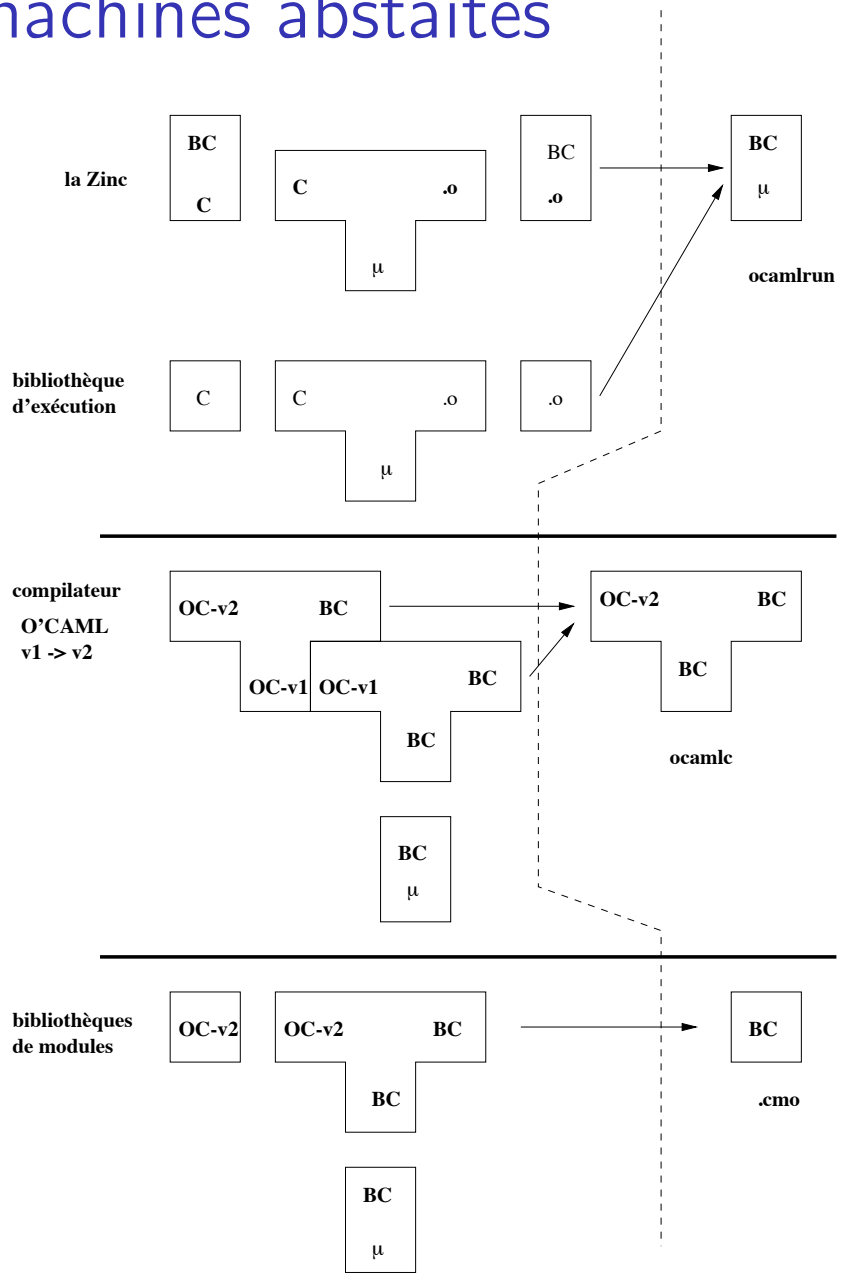
- caractéristiques
  - pile (variables locales à la place des registres)
  - invokestatic : appel de fonction
  - invokevirtual (SEND) : appel de méthode
  - vérification du code : saut, typage statique et dynamique, niveau de pile
- JIT : Just in Time

modèle logique : WAM

Warren Abstract Machine (pour le langage Prolog)

- caractéristiques principales (les zones mémoires) :
  - le tas (ou pile globale) pour allouer les valeurs
  - la pile locale por les environnements et les points de choix
  - la piste (trail) pour enregistrer les liaisons des variables et pourvoir les défaire lors dun backtrack.
- références :
  - D. Warren - An abstract Prolog instruction set". `http://www.ai.sri.com/pubs/files/641.pdf`
  - Hassan Aït-Kaci - A tutorial : `http://www.vanx.org/archive/wam/wam.html`

Compilation et machines abstaites



ZAM ou JVM ?

- |es 2 sont déjà trop riches
  - ZAM pour les langages fonctionnels
  - JVM pour les langages à objets
- désassembleur :
  - javap pour Java
  - ocaml -dinstr pour O'Caml
- interprète de byte-code
  - java, jasmin
  - ocamlrun, outils pédagogiques

machine abstraite Caml : la ZAM

machine abstraite pour langage fonctionnel strict : SECD, FAM, CAM, ...

ZAM : machine utilisée pour caml-light et Objective Caml

- améliore l'efficacité de l'application : A B C
- machine à pile

ZAM : références

- distribution OCaml : `http://caml.inria.fr(ocamlrun)`
- X. Leroy : The ZINC experiment: an economical implementation of the ML language. Rapport technique 117, INRIA, 1990.
- cours P. Letouzey : `http://http://www.pps.univ-paris-diderot.fr/~letouzey/teaching.fr.html`
- projet Cadmium (Zam en Java) : `http://cadmium.x9c.fr/`
- interprète ZAM en javaScript (O'Browser) : `http://http://www.pps.univ-paris-diderot.fr/~canou/obrowser/tutorial`
- Zamcov : interprète ZAM en OCaml (projet Couverture) `http://www.algo-prog.info/zamcov/web/`
- OcaPic : interpète ZAM pour micro-contrôleurs `http://www.algo-prog.info/ocaml_for_pic/web/`

ZAM : mémoires

- registres
  - PC : compteur ordinal
  - SP : pointeur de pile
  - ACCU : accumulateur
  - autres : ENV, EXTRA_ARGS,
- pile : vecteur de valeurs
- tas : zone d'allocation dynamique
- valeurs (représentation uniforme)
  - immédiates : entiers 31 bits, booléens (0 et 1), char en ASCII, () unit
  - allouées (dans le tas)

ZAM : instructions

- arithmétique
- pile
- branchements
- appel et retour de fonction
- manipulation de blocs
- valeurs fonctionnelles et environnement
- environnement global

ZAM : instructions de la pile

pile : tableau de valeurs

- PUSH : empile ACCU
- POP n : dépile de n éléments
- ACC n : ACCU <– le n-ième élément de la pile, accès au sommet de pile ACC 0
- PUSHACC n : correspond à PUSH; ACC n
- ASSIGN n : le nième élément de la pile <– ACCU

ZAM : opérateurs arithmétiques

- constantes : CONSTINT n : met n dans ACCU
  avec CONST0 ... CONST3 en cas particuliers
- ADDINT : `ACCU <-- ACCU + le sommet de pile` (qui est dépilé)
- idem pour les autres opérateurs binaires :
  SUBINT, MULINT, DIVINT, MODINT
  ORINT, XORINT, LSLINT, LSRINT, ASRINT,
  LTINT, LEINT, GTINT, GEINT, EQ, NEQ

ZAM : instructions de branchement

avec ocaml -dinstr , affichage de labels

- saut conditionnel
  - BRANCH d : de d instructions
- saut inconditionnel
  - BRANCHIF d : de d instructions si ACCU vaut true
  - BRANCHIFNOT d : de d instructions si ACCU vaut false
- instructions composées (compraison et branchement)
  - BEQ n d : si n égal ACCU saut de d instructions
  - et les autres : BNEQ, BLTINT, BLEINT, BGTINT, BGEINT

ZAM : manipulation de blocs (1)

un bloc mémoire :

- un entête (tag du type, taille, info pour le GC, ..)
- suivi d'un vecteur de valeurs
- référencé par un pointeur (son adresse)

ZAM : manipulation de blocs (2)

instructions

- MAKEBLOCK n k : construit un bloc de tag k, le premier élément est ACCU et les n-1 suivants proviennent de la pile; à la fin ACCU vaut l'adresse du bloc.
- GETFIELD n : si ACCU contient une adresse de bloc, alors ACCU reçoit la n-ième valeur de ce bloc
- SETFIELD n : si ACCU contient une adresse de bloc, alors la n-ième valeur de ce bloc reçoit le sommet de pile; ACCU vaut ensuite ()
- VECTLENGTH : si ACCU contient une adresse de bloc, alors ACCU reçoit le nombre de données de ce bloc.
- beaucoup de variantes

ZAM : appel et retour de fonctions

pas de procédures, que des fonctions

- valeur fonctionnelle (version simplifiée sans ENV ni EXTRA_ARGS)
  - CLOSURE d 0 : crée un bloc fermeture correspondant à la fonction dont le code est au décalage d; l'adresse du bloc est rangé dans ACCU.
  - APPLY n : applique la fermeture contenue dans ACCU aux n arguments rangés dans la pile en sauvegardant dans la pile de PC, ENV, EXTRA_ARGS
  - RETURN m : retire les m premiers élméments de la pile, puis remet les valeurs des registres suivants : PC, ENV, EXTRA_ARGS (et retourne à PC))
  - APPTERM n,n+m : groupe APPLY n + RETURN m
    ce qui permet d'éviter la sauvegarde intermédiaire dans la pile
    permet de gérer l'appel terminal (dernière expression) à une fonction

ZAM : valeurs fonctionnelles

- CLOSURE n d : crée un bloc fermeture contenant d'une part l'adresse du code (décalé de d) et n+1 éléments (ACCU et n éléments de la pile); vette valeur est rangée dans ACCU

ZAM : environnement global

recherche dun symbole global “g” dans l’environnement :

Toploop :

```
1           const "g"
2           push
3           getglobal Toploop!
4           getfield 0
5           apply 1
```

Pervasives :

```
1   L1:     push
2           getglobal Pervasives!
3           getfield 27
4           appterm 1, 2
```

Exemple 1 (MNL - MIPS)

```
1  ECRIRE ((10 + 20) - ((30 * 40) / 50))
2  .
3          .data
4  MEM: .space 0
5          .text
6  main:   la $30, MEM
7          li $8, 10
8          li $9, 20
9          add $8, $8, $9
10         li $9, 30
11         li $10, 40
12         mul $9, $9, $10
13         li $10, 50
14         div $9, $9, $10
15         sub $8, $8, $9
16         move $4, $8
17         li $2, 1
18         syscall
19         li $2, 10
20         syscall
```

Exemple 1 (CAML - ZAM)

```
print_int (10 + 20 - 30 * 40 / 50);;

        const 50
        push
        const 40
        push
        const 30
        mulint
        divint
        push
        const 10
        offsetint 20
        subint
        push
        getglobal Pervasives!
        getfield 27
        appterm 1, 2
```

Exemple 2 (MNL - MIPS)

```
VAR n;

FONCT fact (VAR n) {
  SI n == 0
     ALORS RETOUR 1
     SINON RETOUR n * fact (n - 1)
}

ECRIRE "Donnez la valeur de n :\n";
LIRE n;
ECRIRE "Sa factorielle est ";
ECRIRE fact(n);
ECRIRE "\n"
.
```

```
fact:   lw $8, 0($sp)      # n local
        li $9, 0
        seq $8, $8, $9
        bne $8, $0, E100
        lw $8, 0($sp)      # n local
        sw $31, -8($sp)
        sw $8, -12($sp)
        lw $8, 0($sp)      # n local
        li $9, 1
        sub $8, $8, $9
        sw $8, -16($sp)
        addi $sp, $sp, -16
        jal fact
        move $9, $8
        addi $sp, $sp, 16
        lw $8, -12($sp)
        lw $31, -8($sp)
        mul $8, $8, $9
        jr $31
        j E101
E100:   li $8, 1
        jr $31
E101:   nop
        jr $31
```

Exemple 2 (CAML - ZAM)

```
let rec fact n =
  if n == 0 then 1
  else n * fact(n-1)
;;

let main () =
  print_string
    "entre votre nombre : ";;
  let n = read_int () in
  print_int (fact n);
  print_newline();;

main();;
```

```
...
L1:     acc 0
        push
        const 0
        eqint
        branchifnot L2
        const 1
        return 1
L2:     acc 0
        offsetint -1
        push
        offsetclosure 0
        apply 1
        push
        acc 1
        mulint
        return 1

        closure L1, 0
...
```

Schémas de compilation (1)

- MNL –> MIPS

  [*SI expr THEN alt_vraie SINON alt_fausse*] =

  [*expr*]
  `bne $8, $0`
  [*alt_fausse*]
  `j etiqfin`
  `etiqv:` [*alt_vraie*]
  `etiqfin: nop`

Schémas de compilation (2)

- MNL –> ZAM

  [*SI expr THEN alt_vraie SINON alt_fausse*] =

  [*expr*]
  `BRANCHIFNOT L2`
  [*alt_vraie*]
  `BRANCH L1`
  `L2:` [*alt_fausse*]
  `L1:`

Générateur de code

- de MIPS –> ZAM
  un peu l'inverse d'un JIT
  comme `ocamlrunjit` de Basile Starynkevitch
- en modifiant le générateur de code du compilateur MNL
  travail à effectuer en TD et TME

Eléments de la ZAM (1)

- mémoire
  - pseudo-registres : accu et compteur ordinal
  - pile
  - blocs mémoire (tas)
- représentation des données
  - booléens, entiers
  - labels
  - blocs
  - fonctions

Eléments de la ZAM (2)

- instructions
  - opérations de calcul
  - manipulation de pile et d'accu
  - accès à un bloc
  - branchements
  - appel et retour de fonction

Exécution des instructions ZAM (1)

- avec un JIT (Just In Time, Juste à temps)
  qui traduit les instructions ZAM en instructions du processeur cible
- technique utilisée pour la JVM (Java) ou CLR (C#)
- et ocamlrunjit pour la ZAM

Exécution des instructions ZAM (2)

- en utilisant un interprète de cette machine :
  - ocamlrun : celui de la distribution OCaml (en C)
  - cadmium : écrit en Java
    `http://cadmium.x9c.fr/`
  - vm.js : écrit en JavaScript
    `http://www.pps.univ-paris-diderot.fr/~canou/obrowser/tutorial`
  - zamcov : écrit en OCaml
    `http://www.algo-prog.info/zamcov/web/`
  - ocapic : écrit en assembleur Pic :
    `http://www.algo-prog.info/ocaml_for_pic/web/`

compatibles avec le format d'exécutable OCaml.
Le dernier est compatible avec la sortie de l'option `dinstr` du compilateur ocamlc. En particulier il ne gère pas l'environnement global.

ocamlrun en C (1)

- des macros :

```
1 #  define Instruct(name) case name
2 #  define Next break
```

- des registres et environnements

```
1   sp = caml_extern_sp;
2   pc = prog;
3   extra_args = 0;
4   env = Atom(0);
5   accu = Val_int(0);
```

ocamlrun en C (2)

fichier : interp.c

- un grand switch
- opérations accès accu et pile

```
1   curr_instr = *pc++;
2
3  dispatch_instr:
4    switch(curr_instr) {
5
6  /* Basic stack operations */
7
8      Instruct(ACC0):
9        accu = sp[0]; Next;
10 ...
11     Instruct(PUSH):
12       *--sp = accu; Next;
13 ...
```

ocamlrun en C (3)

- branchements

```
1  Instruct(BRANCH):
2    pc += *pc;
3    Next;
4  Instruct(BRANCHIF):
5    if (accu != Val_false) pc += *pc; else pc++;
```

- constantes

```
1  Instruct(CONST0):
2    accu = Val_int(0); Next;
3  Instruct(CONST1):
4    accu = Val_int(1); Next;
5  ...
6  Instruct(PUSHCONSTINT):
7    *--sp = accu;
8    /* Fallthrough */
9  Instruct(CONSTINT):
10   accu = Val_int(*pc);
11   pc++;
12   Next;
```

ocamlrun en C (4)

- opérations

```
1  Instruct(NEGINT):
2    accu = (value)(2 - (intnat)accu);
3    Next;
4  Instruct(ADDINT):
5    accu = (value)((intnat) accu + (intnat) *sp++ - 1);
6    Next;
7  Instruct(SUBINT):
8    accu = (value)((intnat) accu - (intnat) *sp++ + 1);
9    Next;
10 Instruct(MULINT):
11   accu = Val_long(Long_val(accu) * Long_val(*sp++));
12   Next;
13
14 Instruct(DIVINT): {
15   intnat divisor = Long_val(*sp++);
16   if (divisor == 0) { Setup_for_c_call; ↩
         caml_raise_zero_divide(); }
17   accu = Val_long(Long_val(accu) / divisor);
18   Next;
19 }
```

ocamlrun en C (5)

- relations

```
1  #define Integer_comparison(typ,opname,tst) \
2    Instruct(opname): \
3      accu = Val_int((typ) accu tst (typ) *sp++); Next;
4
5      Integer_comparison(intnat,EQ, ==)
6      Integer_comparison(intnat,NEQ, !=)
7      Integer_comparison(intnat,LTINT, <)
8      Integer_comparison(intnat,LEINT, <=)
9      Integer_comparison(intnat,GTINT, >)
10     Integer_comparison(intnat,GEINT, >=)
11     Integer_comparison(uintnat,ULTINT, <)
12     Integer_comparison(uintnat,UGEINT, >=)
```

ocamlrun en C (6)

- mécanisme d'application

```
 1      Instruct(APPLY): {
 2        extra_args = *pc - 1;
 3        pc = Code_val(accu);
 4        env = accu;
 5        goto check_stacks;
 6      }
 7  ...
 8      Instruct(APPLY1): {
 9        value arg1 = sp[0];
10        sp -= 3;
11        sp[0] = arg1;
12        sp[1] = (value)pc;
13        sp[2] = env;
14        sp[3] = Val_long(extra_args);
15        pc = Code_val(accu);
16        env = accu;
17        extra_args = 0;
18        goto check_stacks;
19      }
20  ...
```

ocamlrun en C (7)

- retour d'appel

```
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    }
```

ocamlrun en C (8)

- mécanisme d'application

```
1       Instruct(APPTERM): {
2         int nargs = *pc++;
3         int slotsize = *pc;
4         value * newsp;
5         int i;
6         /* Slide the nargs bottom words of the current ↩
              frame to the top
7            of the frame, and discard the remainder of ↩
                 the frame */
8         newsp = sp + slotsize - nargs;
9         for (i = nargs - 1; i >= 0; i--) newsp[i] = sp[i↩
              ];
10        sp = newsp;
11        pc = Code_val(accu);
12        env = accu;
13        extra_args += nargs - 1;
14        goto check_stacks;
15      }
16  ...
```

ocamlrun en C (9)

- fonctions externes

```
1     Instruct(C_CALL1):
2       Setup_for_c_call;
3       accu = Primitive(*pc)(accu);
4       Restore_after_c_call;
5       pc++;
6       Next;
7     Instruct(C_CALL2):
8       Setup_for_c_call;
9       accu = Primitive(*pc)(accu, sp[1]);
10      Restore_after_c_call;
11      sp += 1;
12      pc++;
13      Next;
```

autres points

- exception, appel de méthode, ...
- bibliothèque d'exécution : gestion mémoire (GC), ...
- difficultés variables selon le langage d'implantation
  - OCaml, Java ont un GC et unmécanisme d'exceptions
  - C et l'assembleur n'ont pas de GC,
- format des exécutables
  - zone d'une exécutable
  - différent de la sortie de -dinstr