LI349 — Compilation — 2012/2013

cours 11

Machine abstraite et génération de code

Philippe Wang
philippe.wang@lip6.fr

http://www-apr.lip6.fr/~pwang/LI349-2012-2013--cours-11.pdf

UPMC — Licence Informatique — Lundi 3 Décembre 2012, 14h—15h45, Amphi 55B

Plan du cours 11

les pages de ce cours sont très inspirées de celles d'Emmanuel Chailloux qui a donné ce cours les quelques années précédentes

- Généralités sur les machines abstraites
- Description de la ZAM
- Compiler pour la ZAM
- Implantations de la ZAM

Généralités sur les machines abstraites

Machines Abstraites

• du matériel concret aux modèles de calcul •

S'abstraire des des machines réelles :

- un modèle de calcul bien défini
  - enrichissements: fonctions, objets, formules logiques, etc.
- gestion mémoire
  - manuelle (*free*/*malloc*) vs. automatique (*Garbage Collector*)
- portabilité : comme s'il n'existait qu'une seule machine
  - implanter une fois la machine virtuelle par machine réelle
- interopérabilité des langages : partage d'un modèle de calcul
- autres caractéristiques : performances, compacité du code, etc.

Différents modèles de calcul

• deux modèles (mathématiquement) universels •

mathématiquement capables de faire tous les calculs faisables par une machine
(*cf.* thèse de Church)

machine de Turing

- ruban
(programme/données)
- tête de lecture et écriture
- registre d'état
- table d'instructions
- état : ruban + position de la tête + registre d'état

machine de Krivine

- état :
environnement + terme + pile
- 3 instructions :
grab, push, access
- 3 sortes de termes :
variable (*e.g.*, x),
abstraction (*e.g.*, fun x → y),
application (*e.g.*, z t)

Différents modèles de calcul

• modèle impératif •
P-code : une machine à pile, conçue pour compiler le langage Pascal

- 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 d'allocations dynamiques

• modèle fonctionnel •
SECD (Landin) : une machine à pile, pour la programmation fonctionnelle

- machine à pile
- Pile (SP), Environnement (E), Code (PC), Dump (liste de registres)
- instructions pour la programmation fonctionnelle :
  - CLOSURE : création d'une fermeture (environnement + pointeur de code)
  - APPLY : application d'une fermeture

Différents modèles de calcul

• des modèles plus communément rencontrés dans la pratique •

• modèle pour la prog. à objets •
JVM / .NET CLR

- machine à pile
(pas de registres, mais des variables locales)
- **invokestatic** : pour les fonctions
- **invokevirtual** (send) : pour l'appel de méthodes
(ou "envoi de messages")
- plusieurs vérifications du code (compilation, chargement des classes, exécution) : sauts, typage statique et dynamique, niveau de pile

• modèle pour la prog. logique •
WAM (Warren Abstrat Machine)

- tas (pile globale) pour allouer
- pile locale pour les environnements et les points de choix
- piste (trail) pour les liaisons de variables et pouvoir les défaire lors d'un backtrack
- *cf.* D. Warren "An abstract Prolog instruction set"

D'autres machines abstraites

- FAM (functional abstract machine) pour les programmes fonctionnels (une sorte de SECD optimisée)
- ChAM (chemical abstract machine) pour les programmes concurrents
- CAM (categorical abstract machine) pour CAML
- **ZAM (zinc abstract machine) pour Caml-light / OCaml**
- LLVM (*cf.* llvm.org) pour de nombreux langages mais surtout C/C++
- PVM (parrot virtual machine) pour Perl 6 et d'autres
- AVM (ActionScript Virtual Machine) pour Adobe Flash
- SpiderMonkey pour JavaScript
- Virtualisation, émulation de processeurs et gestionnaires de machines virtuelles : QEMU, KVM, VMware, VirtualBox, Parallels, VirtualPC, Xen, ...

Description de la ZAM

La machine virtuelle OCaml : ZAM

• une machine à pile à la SECD •
• optimisée pour les applications multi-arguments •

Quelques caractéristiques

- 7 registres : **PC**, **SP**, **ACCU**, **ENV**, **EXTRAARGS**, **TRAPSP**, **GLOBALDATA**
- pile : vecteur de valeurs
- tas : zone d'allocations dynamiques
- représentation uniforme des valeurs
  - immédiates : entiers (*e.g.*, types int, bool, char, unit)

  - allouées dans le tas

La machine virtuelle OCaml : ZAM

• jeu de 147 instructions •
(60% sont des raccourcis)

- arithmétique sur les entiers signés
- pile
- branchements
- appels et retours de fonctions
- manipulation de blocs
- valeurs fonctionnelles (fermetures) et environnements
- environnement global

La machine virtuelle OCaml : ZAM

• jeu d'instructions : pour la pile •

- **PUSH** : empile **ACCU**
- **POP n** : dépile **n** éléments
- **ACC n** : **ACCU** reçoit le **n**-ème élément de la pile

  (**ACC 0** : **ACCU** reçoit le sommet de la pile)
- **PUSHACC n** : raccourci pour **PUSH ; ACC n**
- **ASSIGN n** : le **n**-ème élément de la pile reçoit **ACCU**

La machine virtuelle OCaml : ZAM

• jeu d'instructions : opérateurs arithmétiques •

- Constantes : `CONSTINT n` : met `n` dans `ACCU`
  Quelques raccourcis : `CONSTINT0` ; ... ; `CONSTINT3`
- `ADDINT` : `ACCU` reçoit `ACCU +` le sommet de pile (dépilé)
- idem pour les opérateurs suivants :
  `SUBINT` ; `MULINT` ; `DIVINT` ; `MODINT` ; `ORINT` ; `XORINT` ; `LSLINT` ; `LSRINT` ; `ASRINT` ; `LTINT` ; `LEINT` ; `GTINT` ; `GEINT` ; `EQ` ; `NEQ`

La machine virtuelle OCaml : ZAM

• jeu d'instructions : instructions de branchement •

- Saut inconditionnel : **BRANCH d** saut de **d** instructions
- Sauts conditionnels :
  BRANCHIF d saut de **d** instructions si **ACCU** vaut **true**
  BRANCHIFNOT d saut de **d** instructions si **ACCU** vaut **false**
- Instructions composées (comparaison et branchement)
  BEQ n d : si **n** égal **ACCU** alors saut de **d** instructions
  et autres : **BNEQ** ; **BLTINT** ; **BLEINT** ; **BGTINT** ; **BGEINT**

La machine virtuelle OCaml : ZAM

• jeu d'instructions : manipulations de blocs •

- 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-ème valeur de ce bloc
- SETFIELD n : si ACCU contient une adresse de bloc, alors la n-ème valeur de ce bloc reçoit le sommet de pile et ACCU reçoit ()
- VECTLENGTH : si ACCU contient une adresse de bloc, alors ACCU reçoit le nombre de valeurs de ce bloc
- etc.

La machine virtuelle OCaml : ZAM

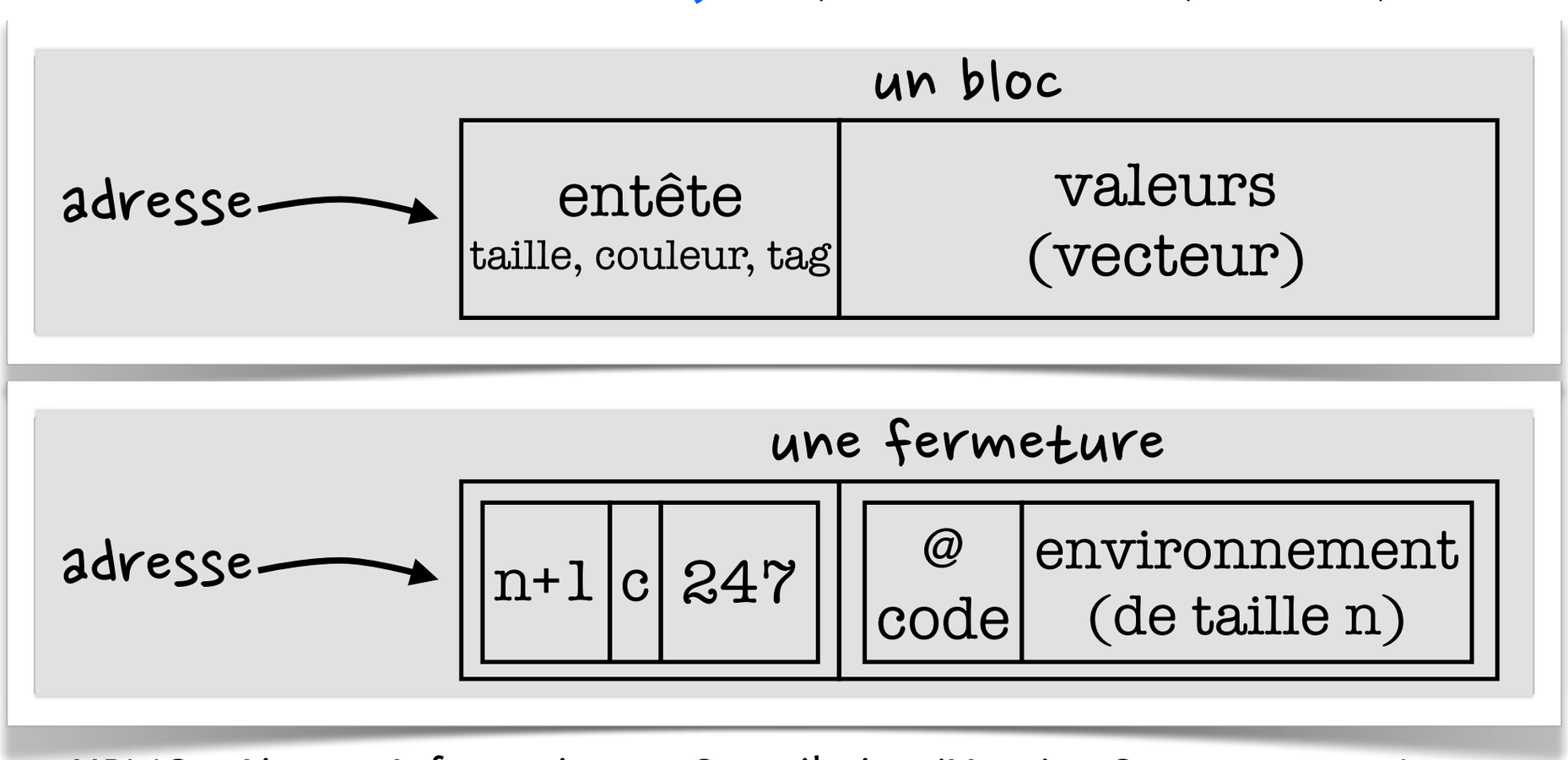
• jeu d'instructions : appels et retours de fonctions •
(pas de procédure, que des fonctions)

- `CLOSURE 0 d` : crée une fermeture correspondant à la fonction dont le code est au décalage `d` ; l'adresse du bloc est rangée dans `ACCU`.
- `APPLY n` : applique la fermeture contenue dans `ACCU` aux `n` arguments rangés dans la pile en sauvegardant dans la pile les registres `PC`, `ENV`, `EXTRAARGS`, et retourne `PC`.
- `RETURN m` : retire les `m` premiers éléments de la pile, puis remet les valeurs des registres `PC`, `ENV` et `EXTRAARGS` ; et retourne à `PC`.
- `APPTERM n, n+m` : groupe `APPLY n` et `RETURN m`, ce qui permet d'éviter la sauvegarde intermédiaire dans la pile ; et réduit l'occupation de la pile !

La machine virtuelle OCaml : ZAM

• jeu d'instructions : appels et retours de fonctions •

- `CLOSURE n d` : crée une fermeture contenant l'adresse du code (décalée de `d`) et `n+1` éléments (`ACCU` et `n` éléments de la pile) ; cette valeur est rangée dans `ACCU`.
- Autres instructions : `GRAB`, `CLOSUREREC`, `CCALL`, `RESTART`, ...



Compiler pour la ZAM

Exemple MNL — MIPS

```
ECRIRE ((10 + 20) - ((30 * 40) / 50))
.
        .data
MEM:  .space 0
        .text
main:   la $30, MEM
        li $8, 10
        li $9, 20
        add $8, $8, $9
        li $9, 30
        li $10, 40
        mul $9, $9, $10
        li $10, 50
        div $9, $9, $10
        sub $8, $8, $9
        move $4, $8
        li $2, 1
        syscall
        li $2, 10
        syscall
```

Exemple MNL — MIPS

```
ECRIRE ((10 + 20) - ((30 * 40) / 50))
.
        .data
MEM:  .space 0
        .text
main:   la $30, MEM
        li $8, 10
        li $9, 20
        add $8, $8, $9
        li $9, 30
        li $10, 40
        mul $9, $9, $10
        li $10, 50
        div $9, $9, $10
        sub $8, $8, $9
        move $4, $8
        li $2, 1
        syscall
        li $2, 10
        syscall
```

Exemple OCaml – 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
```

} obtenu avec ocaml -dinstr

Exemple OCaml — 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
```

`offsetint 20` → optimisation de `(push; const 20; addint)`

`getglobal Pervasives!` / `getfield 27` / `appterm 1, 2` } `print_int`

un module OCaml, dans la ZAM,
c'est simplement un tableau de valeurs

Schémas de compilation

• [code MNL] = code MIPS •

- Soit [ ] la fonction de compilation d'une expression MNL vers MIPS
- [SI exprA ALORS exprB SINON exprC] =

```
        [ exprA ]
        bne $8, $0 etiqv
        [ exprC ]
        j etiqfin
etiqv:  [ exprB ]
etiqfin: nop
```

=> a été vu et mis en pratique en TD/TME

Schémas de compilation

• [code MNL] = code ZAM •

- Soit [ ] la fonction de compilation d'une expression MNL vers ZAM
- [SI exprA ALORS exprB SINON exprC] =

```
    [ exprA ]
    BRANCHIFNOT L2
    [ exprB ]
    BRANCH L1
L2: [ exprC ]
L1:
```

=> sera vu et mis en pratique en TD/TME

Implantations de la ZAM

La machine virtuelle OCaml : ZAM

• une machine à pile à la SECD •
• optimisée pour les applications multi-arguments •

Quelques références

- Distribution OCaml : *cf.* caml.inria.fr (ocamlrun)
- X. Leroy : The ZINC experiment: an economical implementation of ML, rapport technique, 1990, INRIA.
- cours de P. Letouzey : http://www.pps.univ-paris-diderot.fr/~letouzey/teaching.fr.html
- projet Cadmium (ZAM en Java) : http://cadmium.x9c.fr
- O'Browser (ZAM en JavaScript)
- Zamcov (ZAM en OCaml)
- OCaPIC (ZAM en assembleur microcontrôleurs PIC18)

Caractéristiques des implantations de la ZAM

- **ocamlrun**, **cadmium**, **obrowser**, **zamcov** et **ocapic** utilisent le format standard (défini avec **ocamlrun**)
- **ma_cvm_ng** (ZAM partielle, de P. Letouzey) utilise le format donné par la commande `ocamlc -dinstr`
- **ocapic** utilise une VM 16 bits sur PIC18
  (micro-contrôleur 8 bits, max 10MHz, max 4kB de mémoire vive)
- **ocamlcc** traduit le bytecode vers du C pour GCC
- **ocamlrunjit** (et versions ultérieures) compilent le bytecode vers du code natif à la volée
  (techniques de compilation JIT, juste-à-temps)

ocamlrun, la ZAM de référence, en C

- une bibliothèque d'exécution en C
- un boucle avec un grand ***switch*** sur les instructions

```
#define Next break
#define Instruct(name) case name
while (1) {
  ...
  switch(curr_instr) {
    ...
    Instruct (PUSH):
        *--sp = accu; Next;
  } // end switch
} // end while
```

http://www-apr.lip6.fr/~pwang/interp.c.html

Autres caractéristiques de la ZAM

- pose d'un rattrapeur d'exception (try) :
  - sauvegarder les registres sur la pile
  - mettre le niveau courant de la pile dans **TRAPSP**
  => coût constant (*vs.* coût zéro).

- lancement d'une exception (raise) :
  - dépiler jusqu'au niveau indiqué dans **TRAPSP**
  - restaurer les registres (qui sont sur la pile)
  => coût constant (*vs.* coût linéaire).

- ocamlrun :
  - gestionnaire automatique de la mémoire particulièrement efficace pour les programmes séquentiels, écrit en C.
  - coût amorti de l'allocation (toute taille d'objet) : O(1)

Conseils

- plus grande votre culture des langages de programmation sera,

meilleurs vos programmes seront,

mais surtout,
les épreuves de compilation vous sembleront plus accessibles (voire faciles).