

Machines Virtuelles

Cours 2 : La machine virtuelle OCaml

Pierre Letouzey¹
pierre.letouzey@inria.fr

PPS - Université Denis Diderot – Paris 7

janvier 2012

1. Merci à Y. Régis-Gianas pour les transparents

À la découverte d'un fichier « CMO »

À la découverte d'un fichier « CMO »

- | Observons le contenu du fichier .cmo produit par la commande « ocamlc -c » sur le programme O'CamI suivant :

```
let x = 1 + 2 + 3 × 4 / 5
```

⇒ Comment ? À l'aide de quelle commande (sous UNIX) ?

Représentation binaire d'un fichier « CMO »

- À l'aide de la commande `hexedit`, on peut retrouver quelques éléments de notre programme initial :

| | | | | | |
|----------|-------------|-------------|-------------|-------------|------------------|
| 00000000 | 43 61 6D 6C | 31 39 39 39 | 4F 30 30 37 | 00 00 00 58 | Cam119990007...X |
| 00000010 | 67 00 00 00 | 05 00 00 00 | 6C 00 00 00 | 04 00 00 00 | g.....l..... |
| 00000020 | 6B 00 00 00 | 70 00 00 00 | 71 00 00 00 | 69 00 00 00 | k...p...q...i... |
| 00000030 | 7F 00 00 00 | 02 00 00 00 | 6E 00 00 00 | 0A 00 00 00 |n..... |

- On retrouve les entiers 5, 4 et 2.
- C'est tout de même assez cryptique...

Format d'un fichier « CMO »

- | Il existe une spécification du format « CMO » que l'on peut trouver ici :
<http://cadmi.um.x9c.fr/distrib/caml-formats.pdf>
- | Il existe aussi des outils pour obtenir un affichage plus lisible de ces fichiers :
 - ▶ `ocaml dumpobj` ou `ocaml c -di nstr` :
Affiche les descriptions textuelles des instructions.
 - ▶ `obj i nfo` :
Affiche des informations supplémentaires sur le code.

Retour sur l'exemple

- Voici la version lisible du code du fichier .cmo :

```
## start of ocaml dump of "test.cmo"  
0 CONSTANT 5  
2 PUSHCONSTINT 4  
4 PUSHCONST3  
5 MULINT  
6 DIVINT  
7 PUSHCONST1  
8 OFFSETINT 2  
10 ADDINT  
11 PUSHACC0  
12 MAKEBLOCK1 0  
14 POP 1  
16 SETGLOBAL Test  
## end of ocaml dump of "test.cmo"
```

⇒ Il faut comprendre ces instructions et sur quel environnement elles agissent.

La machine virtuelle d'O'Caml

- | Elle est réalisée par le programme exécutable `ocaml run`.
- ⇒ (man `ocaml run`)
- | Elle est programmée en C.
- | Dans les sources d'O'Caml , on la trouve dans le répertoire « `byterun` ».
- | Le fichier principal est « `interp.c` ».
- ⇒ Un programme qui ressemble beaucoup à la machine virtuelle que vous avez programmée en TD ... Toutes deux implémentent le cycle
« **Récupère-Décode-Exécute** » (*Fetch-Decode-Execute*).

L'architecture des ordinateurs en 20 minutes

Architecture des ordinateurs

- | Comprendre l'architecture des ordinateurs est essentiel.
- ⇒ Comment fonctionne un ordinateur ?
- ⇒ Qu'est-ce qui est effectivement réalisable ?

| Deux références à ce sujet :

▶ *Architecture des ordinateurs : une approche quantitative*

John Hennessy, David Patterson

Intern. Thomson Publicat. (January 7, 1999)

ISBN-10 : 2841800229



| *Cours d'Architecture des Ordinateurs*

Emmanuel Viennet

<http://www-gtr.iutv.univ-paris13.fr/Cours/Mat/Architecture/Cours/polyarch.pdf>

Composants principaux d'un ordinateur

- | Processeur (*Central Processing Unit*) : exécute les instructions.
- | Mémoire principale : stockage pour les programmes et les données traitées.
- | Mémoire secondaire : stockage persistants des programmes et des données.
- | Périphérique d'entrée : communication de l'utilisateur vers l'ordinateur.
- | Périphérique de sortie : communication de l'ordinateur vers l'utilisateur.

Le processeur

- | On distingue trois composantes à l'intérieur du processeur :
 - ▶ L'unité de contrôle coordonne l'avancée des calculs.
 - ▶ L'unité arithmétique effectue les calculs.
 - ▶ Les registres servent de mémoire à accès très rapide.
- | On distingue différents types de registres, fonction de leur utilisation :
 - ▶ les registres d'adressage ;
 - ▶ les registres de données d'entrées ;
 - ▶ les registres de résultats.
- | Il existe souvent des registres spéciaux :
 - ▶ IR : un registre qui stocke l'instruction courante.
 - ▶ PC : un registre qui stocke l'adressage de l'instruction courante dans le code.

Les ensembles d'instructions

- | Les processeurs qui ont un ensemble réduit d'instructions généralistes sont de type RISC (*Reduced Instruction Set Computer*)
 - | Les processeurs qui ont un ensemble important d'instructions spécialisées sont de type CISC (*Complex Instruction Set Computer*)
- ⇒ Une frontière pas toujours très nette ...

La mémoire

- | La mémoire est exclusivement formée de 0 et de 1, les bits.
- | Nous choisissons une **interprétation** de cette mémoire.
- | Pour faciliter la définition de leur interprétation et leur manipulation, on a choisi de grouper ces bits en paquets de 8 bits, appelés **octets** (*bytes*).
- | Pour traiter plus rapidement les informations, un processeur groupe ces octets en **mots**. Aujourd'hui, les processeurs traitent les informations par paquet de 8 octets, c'est-à-dire 64 bits.
- | La taille des mots varie d'un processeur à l'autre, mais il s'agit toujours d'un nombre paire d'octets.
- | Les unités pour quantifier la taille d'une information sont des puissances de 1024 octets :

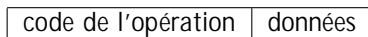
| | | |
|------------|-------|----------------------------------|
| Kilo-octet | Ko/KB | 1024 octets |
| Mega-octet | Mo/MB | 1024 * 1024 octets |
| Giga-octet | Go/GB | 1024 * 1024 * 1024 octets |
| Tera-octet | To/TB | 1024 * 1024 * 1024 * 1024 octets |

Les informations dans la mémoire

- | Une information est localisée dans la mémoire par une **adresse**.
- | La plupart du temps, chaque cellule mémoire contient un octet.
- | Un mot doit donc être réparti sur plusieurs octets consécutifs.
- | Il existe plusieurs façons d'agencer ces octets :
 - ▶ *little endian* (i386, ...) :
Des octets de poids faible vers les octets de poids fort.
 - ▶ *big endian* (SPARC, ...) :
Des octets de poids fort vers les octets de poids faible.
 - ▶ Il existe des agencements plus « exotiques ».
- | Le processeur lit le contenu d'une adresse en le stockant dans un registre.
- | Le processeur écrit dans une adresse en y stockant le contenu d'un registre.
Dans cette situation, le précédent contenu est détruit !
- | La mémoire primaire est rapide mais s'éteint lorsque la machine est éteinte.
- | La mémoire secondaire est lente mais persiste même si la machine est éteinte.

Les programmes dans la mémoire

- | Les programmes sont stockés dans la mémoire secondaire.
- | Pour les exécuter (rapidement), ils sont chargés en mémoire primaire.
- | Une instruction est représentée par une séquence de bits qui suit généralement l'organisation suivante :



- | La taille d'une instruction est généralement proportionnelle à la taille des mots du processeur.
- | Le **code de l'opération** (*opcode*) est un indice faisant référence à la table d'instructions du processeur (add, mul , push, ...).
- | En fonction de ce code, la suite de l'instruction est **décodée** : ce sont des données réparties en deux catégories, les **opérandes** et les **adresses**.

Le cycle « Récupère-Décode-Exécute »

- | Le cycle « Récupère-Décode-Exécute » (*Fetch-Decode-Execute*) est l'activité principale d'un processeur.
- | On peut le décrire (naïvement) ainsi :
 1. Phase « Récupère » :

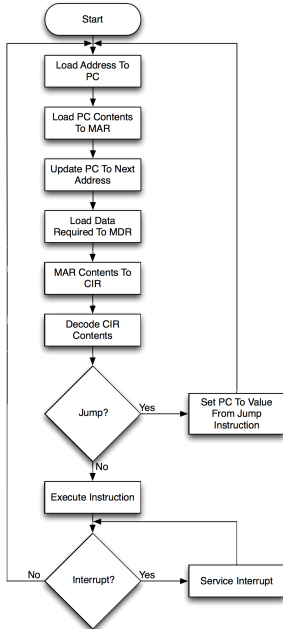
Le processeur détermine la prochaine instruction à partir de l'adresse contenue dans le registre PC et la stocke dans le registre IR.

 - ▶ Quand l'exécution du programme débute, le registre PC est placé au début du programme.
 - ▶ Dès que IR est mis-à-jour, PC est incrémenté pour la prochaine itération.
 2. Phase « Décode » :

Le processeur interprète la partie « données » de l'instruction en fonction du code de l'opération.
 3. Phase « Exécute » :

Le processeur effectue l'opération.
 4. Retour en 1.
- | La vitesse de l'horloge, la fréquence du processeur, correspond au nombre de cycles effectués par unité de temps.

Les machines modernes ...



- | Ce qui précède est encore très éloigné de la réalité.
- | Vous devrez aborder de nombreuses questions pour comprendre réellement les machines sur lesquelles vous travaillez :
 - ▶ Comment traiter l'aspect asynchrone des entrées/sorties ?
 - ▶ Quels procédés ont été inventés pour exécuter plus d'une instruction en un cycle ou encore pour accéder plus rapidement à la partie de la mémoire la plus utile ?
 - ▶ ...

Les entrailles de la machine virtuelle d'O'Caml

Éléments principaux de la machine virtuelle d'O'Caml

- | un programme (liste de bytecodes)
- | un registre **pc** : pointeur de code dans le programme
- | un registre **accu**
- | une **pile**
- | **accu+pile = a-pile**
- | un **tas** (heap) où placer les données **allouées**

Éléments qui nous intéresserons moins :

- | un environnement global (pour les entités définies "à toplevel")
- | un pointeur **env** vers une archive d'environnement
- | un registre **extra_args**

Retour sur l'exemple

| instruction | accu | pile |
|-----------------|------|---------|
| CONST INT 5 | 5 | ... |
| PUSHCONST INT 4 | 4 | 5 ... |
| PUSHCONST 3 | 3 | 4 5 ... |
| MUL INT | 12 | 5 ... |
| DIV INT | 2 | ... |
| PUSHCONST 1 | 1 | 2 ... |
| OFFSET INT 2 | 3 | 2 ... |
| ADD INT | 5 | ... |

Les instructions de la machine virtuelle

Gestion de la pile

- | **PUSH** :
Empile **accu**
- | **POP n** :
Dépile **n** éléments du sommet de la **pile**
- | **ACC n** :
accu reçoit la valeur de la **n**-ième case de la **pile**
($n = 0$ pour le sommet de pile).
Variantes :
 - ▶ **ACC0 ... ACC7** : instructions spécialisées pour $n < 8$
 - ▶ **PUSHACC n** : on sauve **accu** via **PUSH** avant un **ACC n**
 - ▶ **PUSHACC0 ... PUSHACC7** : cf. **PUSHACC n**
- | **ASSIGN n** :
Le **n**-ième élément de la **pile** est remplacé par le contenu de **accu**.
Celui-ci vaut ensuite **()**.

Arithmétique

| **CONSTINT** n :

L'entier n est placé dans **accu**.

Variantes :

- ▶ **CONST0 ... CONST3** : cf **CONSTINT** pour $n < 4$
- ▶ **PUSHCONSTINT** n : **PUSH** + **CONSTINT** n
- ▶ **PUSHCONST0...3** : cf. **PUSHCONSTINT** n

| **OFFSETINT** n :

accu est incrémenté de n

| **NEGINT** :

accu reçoit l'opposé de sa valeur.

| **ADDINT** :

accu reçoit **accu** + sommet de la **pile** (qui est dépilé au passage).

Cf. vision sous forme de **a-pile**.

- | idem avec les autres opérateurs : **SUBINT**, **MULINT**, **DIVINT**, **MODINT**, **ANDINT**, **ORINT**, **XORINT**, **LSLINT**, **LSRINT**, **ASRINT**, **(U)LTINT**, **LEINT**, **GTINT**, **(U)GEINT**, **(N)EQ**.

Aiguillages

- | **BRANCH o** :
Saut obligatoire vers **o** instructions plus loin (ou en arrière selon le signe de **o**).
- | **BRANCHIF o** :
Saut de **o** instructions si **accu** représente **true**
- | **BRANCHIFNOT o** :
Saut de **o** instructions si **accu** représente **false**

(La description des valeurs **true** et **false** viendra un peu plus tard.)

Variantes :

- | **BEQ n o** : compare **n** avec le contenu de **accu** et saute de **o** si le test est positif.
- | Idem avec **BNEQ**, **B(U)LTINT**, **BLEINT**, **BGTINT**, **B(U)GEINT**.

Mémoire et données O'Caml

Toute donnée en O'Caml correspond à **un** mot machine (32/64 bits) :

- | soit cet entier est une valeur immédiate (`int`, `bool`, ...);
- | soit cet entier est un pointeur vers un **bloc** mémoire.

Données codés par des entiers

- | `int` : entiers signés sur 31 ou 63 bits. (Nous reviendrons sur ce point.)
- | `char` : codé par le code ASCII correspondant.
- | `bool` : `false` (resp. `true`) est codé par 0 (resp. 1)
- | `unit` : `()` est codé par 0
- | Dans un type concret `type t = A | B of int | C`,
les constructeurs constants `A`, `C` sont des numéros : 0 et 1.
- | En particulier : `[]` est codé par 0, tout comme `None`, ...

Données allouées par blocs : tout le reste !

- | Un bloc en mémoire est une portion du **tas** (ou **heap**).
- | L'entier référençant cette zone est son adresse (pointeur).
- | Le bloc commence par un entier d'en-tête, suivi des données.

En-tête (sur une machine 32-bits) :

Exemple de *Tag* (cf byterun/ml val ue. h) :

- | Pour un fl oat : 253
- | Pour une fonction (fermeture) : 247
- | $\text{Tag} \in [0..245]$: un constructeur non-constant

Exemples de blocs de données allouées

| Un tableau : [| 1; 2 |]

Wosize = 2

Color = ?? (initialement white, soit 0)

Tag = 0 (1er "constructeur" du type array)

En mémoire :

| | | |
|--------------|---|---|
| $2 * 2^{10}$ | 1 | 2 |
|--------------|---|---|

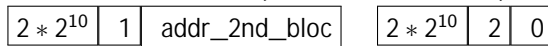
(en fait pas tout à fait exact, cf plus loin)

| Une paire : (1, 2)

Même représentation en mémoire !!

| Une liste : [1; 2]

Deux blocs de taille 2 (Cf. dessin au tableau) :



La séparation entre entiers et pointeurs

- | Tous les blocs alloués peuvent un jour devenir inutiles :
- | Un ramasse-miette (*Garbage Collector (GC)*) fait régulièrement le ménage.
- ⇒ Besoin de **parcourir** les données (cf Color).
- ⇒ Besoin de savoir reconnaître rapidement les pointeurs.
- | **Convention O'Caml :**
Tout entier pair est un pointeur,
Tout entier impair $2n + 1$ représente l'entier n .
- | C'est la raison du bit manquant dans la taille de `int` !
- | Retour sur les exemples précédents :
 - ▶ 0 est codé en mémoire par 1.
 - ▶ 1 par 3, etc.

Instructions de la machine virtuelle pour les blocs

- | **MAKEBLOCK *n* *k*** :

Fabrique un bloc de tag *k* et de données *n* éléments retirés du haut de l'*a-pile*. *accu* reçoit ensuite le pointeur vers le bloc ainsi créé.

- | **GETFIELD *n*** :

accu doit contenir un pointeur vers un bloc. *accu* reçoit alors la *n*-ième donnée du bloc.

- | **SETFIELD *n*** :

accu doit contenir un pointeur vers un bloc. La *n*-ième donnée du bloc est alors remplacée par la valeur prise au somme de la *pile*. *accu* reçoit ensuite *()*.

- | **VECTLENGTH** :

accu reçoit le nombre de données du bloc pointé par *accu*.

Instructions de la V.M. pour les blocs : variantes

- | **MAKEBLOCK1,2,3, GET/SETFIELD0,1,2,3** :
cf. ce qui précède.
- | **GETVECTITEM, SETVECTITEM** :
cf **GETFIELD** et **SETFIELD**, mais avec tous les arguments dans **accu+pile**.
- | **ATOM k** :
Remplace **MAKEBLOCK 0 k** dans le cas d'un bloc avec 0 données (essentiellement [| |]). Il existe aussi : **ATOM0, PUSHATOM k** et **PUSHATOM0**.
- | Les tableaux de float sont traités à part : **MAKEFLOATBLOCK n, GET/SETFLOATFIELD n**
- | Les chaînes de caractères aussi : **GET/SETSTRINGCHAR**
- | **OFFSETREF n** : ajoute **n** au premier champ d'un bloc.

Instructions de la machine virtuelle : switch

- | Cette instruction permet de réaliser le match de O'Caml :

switch $l_0^c \dots l_n^c / l_0^{nc} \dots l_m^{nc}$

- | On examine le contenu de **accu** :
 - ▶ S'il s'agit d'un nombre impair $2 * i + 1$, il représente en fait l'entier i , et plus précisément le i -ème constructeur constant. On effectue donc le déplacement indiqué par l_i^c
 - ▶ S'il s'agit au contraire d'un pointeur, on est en présence d'un constructeur non-constant. On accède au bloc via le pointeur. Si le tag présent dans l'entête du bloc est i , on effectue alors le déplacement l_i^{nc} .

O'Caml : points-clés concernant les fonctions

- | Fonctions anonymes :

`fun x → 1 + x`

- | Fonctions en arguments et/ou en résultat :

`let funincr f = fun x → 1 + f x`

- | Accès hors des frontières de la fonction :

`let a = 10 in (fun x → a + x)`

- | Récursivité (éventuellement mutuelle)

- | Application partielle :

`let add10 = (+) 10`

- | Une optimisation : les appels terminaux

Un exemple élémentaire de fonction

```
% ocaml -dinstr  
# let f x = 1 + x in (f 4) * 2;;
```

closure L1, 0

push
const 2
push
const 4
push
acc 2
apply 1
mulint
return 2

L1 : acc 0

push
const 1
addint
return 1

Description informelle

- | L1 est un **label**, ou position dans le code
- | Le vrai bytecode contient plutôt des décalages (ou offsets)
- | code avant L1 : code d'une sorte de fonction "main"
- | code à partir de L1 : code de f
- | **CLOSURE L1, 0** :
Fabrique un bloc contenant le label L1 et retourne un pointeur ptr vers ce bloc
- | **APPLY 1** :
Lance l'exécution de f (car accu contient ptr) avec 1 argument dans la pile (valant 4).
- | **ACC 0** :
f accède à son premier argument
- | **RETURN 1** :
f rend la main, en nettoyant 1 case de pile.
Le résultat 5 est contenu dans accu.

Sauvegardes/restaurations lors des appels/retours

- | Magie noire : où repartir à la fin d'un appel de fonction?
- ⇒ Besoin de mémoriser pc lors de l'appel.
- | De plus, lors d'appels imbriqués, une case ne suffit pas.
- ⇒ Le pc de retour est sauvé dans la pile, juste après les arguments.
`APPLY1 ... APPLY3` le fait automatiquement, mais pas `APPLY n`, qui doit être précédé d'un `PUSH_RETADDR`.
- | En fait, deux autres éléments sauvegardés en plus : env et extra_args, cf. plus tard.
- | `RETURN n` enlève n éléments de la pile, puis espère trouver ensuite un triplet pc/env/extra_args à restaurer (+ saut à pc).

Environnements

- | Une fonction peut utiliser des objets disparus lors de l'appel :

```
let f =  
  let c = 10 in  
  let g x = x × x in  
  fun x → c + g x  
in 1 + f 1
```

- | Le bloc (ou fermeture) de f sera fait ici avec **CLOSURE L1, 2**.
- | Il contient un environnement (de taille 2) après l'étiquette de f :

| | | | |
|--------------------------|----------|----------|------------|
| Header : size 3, tag 247 | Lbl de f | Val de c | Ptr vers g |
|--------------------------|----------|----------|------------|

- | Le registre env pointe vers la fermeture de la fonction courante.
 - ▶ Accès à c : **ENVACC 1** (suit le registre env, case 1)
 - ▶ Accès à g : **ENVACC 2**

Récurtivité

- | Dans le cas d'une fonction récursive f non-mutuelle, un appel récursif se fait :
 - ▶ en chargeant dans `accu` un pointeur sur la fermeture de f :
`OFFSETCLOSURE0`
 - ▶ en faisant un `APPLY` comme précédemment.
- | Pour les fonctions récursives mutuelles, O'Caml utilise une même fermeture pour tout un paquet mutuel (cf. dessin au tableau).
- | `OFFSETCLOSURE n` permet ensuite de retrouver un pointeur vers une fonction récursive "voisine".

Applications partielles

- | Si une fonction attend deux arguments et n'en reçoit qu'un, l'exécution ne peut avoir lieu. En attendant l'argument manquant, on doit geler l'exécution, *via* une fermeture.
- | Le registre `extra_args` vaut en permanence `nb_args - 1`
- | Au lancement, toute fonction reçoit au moins 1 argument.
- | **GRAB** `n` mis au début d'une fonction vérifie si les `n` arguments suivants sont présents.
 - ▶ OK : exécution normale.
 - ▶ KO : retourne immédiatement une fermeture avec les arguments présents :

| | | | | | |
|--------|----------------|---------|------------------|-----|------------------|
| Header | Lbl de RESTART | ptr env | arg ₀ | ... | arg _k |
|--------|----------------|---------|------------------|-----|------------------|

- | Un **RESTART** précède toujours le **GRAB**. Le jour où la fermeture est appliquée, il s'exécute, et restaure les arguments sauvés.

Appels terminaux

- | Fréquemment, un appel *via* **APPLY** se trouve juste avant un **RETURN**. C'est une perte d'espace de pile et en temps :
 - ▶ le **APPLY** sauvegarde dans la pile `pc/env/extra_args` pour permettre le retour à l'instruction suivante
 - ▶ cette instruction suivante, le **RETURN**, restaure une autre zone `pc/env/extra_args` plus ancienne et saute à ce pc
- | L'idée est alors de grouper **APPLY** n + **RETURN** m en une instruction **APPTERM** $n, n + m$ qui évite la sauvegarde au milieu.
- | Dans le cas de fonctions récursives terminales (tail-recursive), le calcul se fait en pile constante...

Description précise des instructions

| CLOSURE o, n :

Crée une fermeture (bloc de tag 247) contenant le label correspondant à l' o set o suivi de

n éléments d'environnement ôtés de $\text{accu} + \text{pile}$.

Un pointeur vers cette fermeture est placé dans accu .

| CLOSUREREC $o_1 \dots o_k, n$:

Si un seul o set o_1 , cf. CLOSURE + PUSH. Sinon, fabrique la fermeture suivante, en prenant n éléments d'environnement $e_1 \dots e_n$ dans $\text{accu} + \text{pile}$.

| | | | | | | | | | |
|-----|-----------|-----|-----------|-----|-----|-----------|-------|-----|-------|
| hdr | pc+ o_1 | hdr | pc+ o_2 | ... | hdr | pc+ o_k | e_1 | ... | e_n |
|-----|-----------|-----|-----------|-----|-----|-----------|-------|-----|-------|

accu reçoit un pointeur vers cette fermeture, et on fait PUSH

| PUSH_RETADDR :

Empile successivement les valeurs des registres `extra_args`, `env` et `pc`

Description précise des instructions

- | **APPLY n** :

accu doit contenir un pointeur vers une fermeture, et la pile doit contenir **n** arguments suivi d'une zone où sont sauvegardés les pc/env/extra_args courant.

Le registre **env** reçoit **accu**.

Le registre **extra_args** est positionné à **n-1**.

Le point d'exécution **pc** saute au label dans la fermeture.

- | **APPLY1 ... APPLY3** :

cf **APPLY**, sauf que ces variantes sauvent elles-mêmes pc/env/extra_args derrière les arguments : pas besoin de **PUSH_RETADDR** avant.

Description précise des instructions

| RETURN *n* :

Dans tous les cas, on commence par enlever *n* éléments obsolètes de la pile. Deux cas viennent ensuite :

- ▶ Si *extra_args* vaut 0 : c'est le cas usuel d'une fonction ayant reçu exactement assez d'arguments. On termine la fonction en enlevant de la pile trois éléments de plus, qui servent à restaurer respectivement les registres *pc/env/extra_args* (en particulier saut à *pc*).
- ▶ Si *extra_args* > 0, cela signifie que la fonction courante a retourné dans *accu* une fermeture, qu'il faut maintenant exécuter sur les arguments restants : on décroît *extra_args* de 1, *env* reçoit *accu* et on saute au code contenu dans cette fermeture (cf. *APPLY*).

Description précise des instructions

- | **ENVACC n** :
Accède au n -ième champs de la fermeture pointé par le registre **env**.
- | **ENVACC1...ENVACC4** :
idem pour $n = 1..4$
- | **OFFSETCLOSURE n** :
Place dans **accu** le pointeur contenu dans le registre **env**, décalé de n cases.
- | **OFFSETCLOSURE0, OFFSETCLOSURE2, OFFSETCLOSUREM2** :
cf précédent avec $n = 0, 2, -2$
- | ... et toutes les instructions de cette page ont des variantes **PUSHXYZ**

Description précise des instructions

| APPTERM n, m :

Moralement **APPLY** n + **RETURN** $(m-n)$. Plus précisément, la pile doit avoir la forme :

- ▶ n arguments pour l'appel à effectuer
- ▶ $(m-n)$ cases inutiles
- ▶ peut-être de vieux arguments (en nombre **extra_args**)
- ▶ une zone de sauvegarde `pc/env/extra_args` pour sortir de la fonction courante

Les cases inutiles sont supprimées, et on décale les arguments. **extra_args** est incrémenté de $n-1$. Comme pour **APPLY**, `env` reçoit **accu** et on saute au label dans la fermeture.

| APPTERM1 m ... APPTERM3 m :

cf précédent.

Description précise des instructions

| GRAB n :

si $\text{extra_args} \geq n$, on continue avec extra_args décrémenté de n .

Sinon on place les $(\text{extra_args} + 1)$ arguments présents sur la pile dans une fermeture de la forme :

| | | | | | |
|--------|----------------|---------|----------------|-----|----------------|
| Header | Lbl de RESTART | ptr env | arg_0 | ... | arg_k |
|--------|----------------|---------|----------------|-----|----------------|

On retourne alors cette fermeture dans accu et on restaure $\text{pc}/\text{env}/\text{extra_args}$ (cf. **RETURN**).

| RESTART :

env doit pointer vers une fermeture de la forme précédente. On remplace alors les arguments sur la pile, et on incrémente extra_args d'autant.

Synthèse

La Machine Virtuelle O'Caml

- | Quelques aspects d'architecture des ordinateurs à compléter.
- | La machine virtuelle d'O'Caml est une instance du modèle générale qui se distingue par :
 - ▶ l'utilisation d'une a-pi l e.
 - ▶ une représentation homogène des données.
 - ▶ des instructions spécialisées à des cas particuliers courants.
 - ▶ des instructions dédiées aux fonctions de première classe.
 - ▶ la gestion automatique de la mémoire.
- | Nous n'avons pas abordé les exceptions et les objets.
- | Une référence ancienne, plus tout à fait d'actualité, mais encore très instructive :
The ZINC experimentation, Xavier Leroy
<http://caml.inria.fr/pub/papers/xleroy-zinc.ps.gz>