
Introduction aux machines virtuelles

Pierre Letouzey

`Prenom.Nom@pps.jussieu.fr`

Détails Pratiques

Cours: mardi 12h30-14h30, une semaine sur deux, 481F

TPs: en alternance avec le cours, en salle T.

Notation:

– 1ere session: 2/3 examen + 1/3 c.c.

– 2eme session: max (examen, 2/3 examen + 1/3 c.c.)

Le contrôle continu sera sans doute un TP noté

Plan du cours

- **Semaine 1:**
 - Qu'est ce qu'une machine virtuelle ?
 - La première de toutes: la machine de Turing
 - Trois manières d'exécuter du code : interp/vm/asm
 - (?) Début d'étude de la machine virtuelle Caml
- **Semaines 3 et 5:** Machine virtuelle Caml, suite...
- **Semaines 7, 9, 11:** Machine virtuelle Java
- **Semaine 12 (?)**: La virtualisation de systèmes entiers

Bibliographie

Cours atypique, donc pas de livre "tout-en-un".

Sur les **machines de Turing**:

Jean-Michel Autebert. Calculabilité et décidabilité. Masson, 1992.

Sur **Caml**:

Développement d'applications avec Objective Caml, E. Chailloux, P. Manoury, et B. Pagano, Éditions O'Reilly

www.pps.jussieu.fr/Livres/ora/DA-OCAML/index.html

Voir aussi crystal.inria.fr/~lebotlan/docaml.html et

<http://cadmium.x9c.fr/distrib/caml-instructions.pdf>

Sur **Java**:

The Java Virtual Machine, J. Meyer et T. Downing

Livre general: Virtual Machines, Ian D. Craig, Springer

Machine virtuelle, késako ?

Une machine sans **existence physique**: ni silicium ni engrenage

Différence entre **logiciel** et **matériel** (software vs. hardware):
→ impossible de faire passer le hardware par le réseau internet!

On parle donc ici de logiciels qui **miment** du matériel

Une vaste famille, en plein essor:

- Emulateurs généraux (Qemu, Vmware, Parallels) ou spécifiques (consoles, Thomson, etc). [Démonstration Qemu...](#)
- Virtualisation: OS sur un OS (XEN, UserModeLinux)
- **Environnement d'exécution de programmes**: Caml, Java...

Pourquoi ?

⇒ **Pourquoi des machines virtuelles ?**

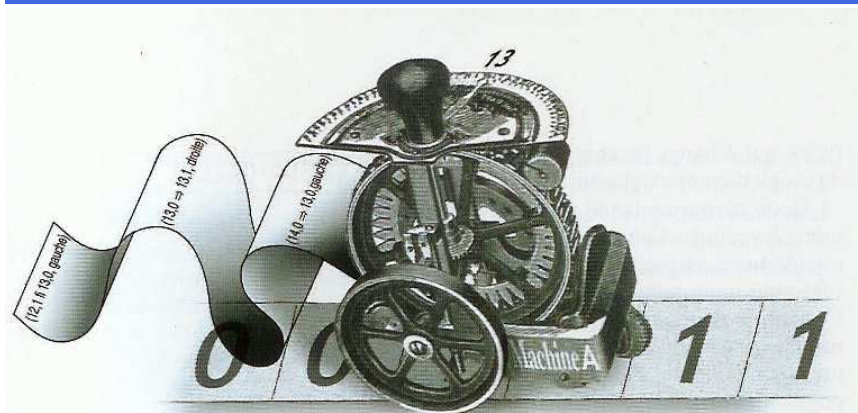
- Intérêt théorique: des machines abstraites servent à établir des résultats scientifiques sur la calculabilité (ou non), sur la complexité, etc.
- Intérêts pratiques: n systèmes dans 1 ordinateur réel, avec isolation, résistance aux crashes et attaques, test aisé, etc...
- Pour les VM Java, Caml: compromis efficacité/portabilité

⇒ **Pourquoi ce cours ?**

Une descente dans les coulisses de Caml et Java:

- Que fait le "toplevel" Caml ? Qu'est-ce qu'un .cmo ?
- Qu'est-ce qu'un .class ?

La première de toutes: la machine de Turing



3. LA MACHINE DE TURING comporte une liste d'instructions qu'un mécanisme exécute sous la forme d'opérations de lecture ou d'écriture sur un ruban, à raison d'un signe par cellule. Après chaque opération, le ruban se décale d'une cellule, vers la droite ou vers la gauche. Cette machine rudimentaire n'en est pas moins extraordinairement puissante: elle est capable de calculer toute fonction effectivement calculable.

Machine de Turing: définition informelle

Ingrédients :

- Une **bande** (infinie) de cases contenant des symboles.
- Une **tête de lecture** en face d'une case
- Un **état courant** (p.ex. un numéro d'état)
- Un **programme** (fini), qui à tout état et symbole lu, associe un nouvel état, un symbole à écrire, et un mouvement (G/D) de la tête de lecture.

Démonstration

Machine de Turing: exercices

1. Écrire une addition de bâtonnets: la bande devra passer de $XX+XXXX$ à $XXXXXX$
2. Écrire une soustraction de bâtonnets.
3. (dur) Écrire une multiplication de bâtonnets.
4. Donner un exemple de machine ne s'arrêtant pas. Comment savoir à l'avance si on est dans ce cas ?

Machine de Turing: un peu d'histoire

Création en 1936 (1er ordinateur : 1945 !).

Sa motivation, l'**Entscheidungsproblem**: existe-t'il une machine répondant à toutes les questions oui/non en math ?

Réponse négative, via l'indécidabilité du **problème de l'arrêt**: pas de méthode générale pour savoir si une machine et une bande initiale vont mener à un arrêt.

Ici "méthode générale" = algorithme = machine de Turing

Ces égalités forment la **Thèse de Church-Turing**: tout ce qui se calcule (même en Java 1.5 sur Amd 64 3500+) peut se faire via une machine de Turing.

Machine de Turing, variantes, ordinateurs

Nombres **variantes**: alphabet des symboles, bandes multiples, bande bornée d'un côté. La plupart sont équivalentes.

Remarque sur la bande **infinie**: c'est juste une convenance.

Un programme ne peut utiliser une infinité de bande en un temps fini ! Ici, infinie = aussi large que nécessaire.

Equivalence entre bande et **deux piles**: voir dessin au tableau.

Analogie avec un ordinateur:

- bande = mémoire
- état = pointeur de code (pc)
- programme = programme (ou plutôt câblage du CPU)

Machine de Turing Universelle

Une seule machine suffit pour tout calculer !

Astuce: on cache un **codage** d'une M.d.T. dans les données de la machine universelle, qui ensuite simule cette M.d.T.

Autrement dit: on peut écrire en M.d.T un **interpréteur** de M.d.T...

Dix ans avant le premier ordinateur polyvalent (encore que, cf Jacquard, Babbage...).

Plus de détails ? cours de M1 Calculabilité et Complexité ...

Modèles d'exécution 1/3: interprétation

Défi: exécutez une phrase Caml sur ordinateur ...
sans donner cette phrase à Caml !

Une solution simple: l'**interprétation**. On découpe la phrase en éléments de base, puis l'on calcule directement dessus, sans plus de transformation.

Exemples de langages interprétés: perl, shell, javascript, lisp, scheme, etc...

Modèles d'exécution 1/3: interprétation

⇒ **Démo d'un interpréteur naif pour Caml...**

fait en Caml pour simplifier, mais possible en C, Java...

Éléments clés:

- un type **expression** mimant les constructions du langage
- un type **value** des résultats possibles (int/bool/fun)
- un **environnement** associant à chaque variable sa valeur.

Bilan: **simple, portable mais très lent**

Modèles d'exécution 2/3: code natif

Un ordinateur est conçu pour exécuter du ... code machine, qui n'est autre que de l'assembleur traduit en 0 et 1.

L'approche opposée à l'interprétation est d'engendrer ce genre de **code natif**.

Difficulté: l'assembleur est délicat à manier, a peu en commun avec les langages haut-niveau (Caml, Java, ...), et diffère de processeur en processeur.

Compilation = traduction d'une expression haut-niveau à de l'assembleur. Voir cours de M1.

Démo des compilateurs natifs ocamlc et gcj.

Bilan: **très ardu, non portable mais très rapide**

Modèles d'exécution 3/3: machine virtuelle

Au lieu gérer chaque processeur comme en code natif, on choisit une fois pour toute une approximation raisonnable:

la **machine virtuelle** (VM) pour le langage:

- pratique pour exécuter le langage
- suffisamment proche du matériel pour être rapide

Objectifs: **relative simplicité, portabilité et rapidité raisonnable**

Modèles d'exécution 3/3: machine virtuelle

Les programmes pour cette VM sont nommés **code-objet** ou **bytecode**. C'est une sorte d'assembleur simplifié.

fichier de bytecode pour Java: `.class`

fichier de bytecode pour Caml: `.cmo` ou "binaires" fait par `ocamlc`

La VM est alors un **programme natif** (p.ex. écrit en C) qui lit du bytecode et l'exécute. C'est une sorte d'interpréteur, en plus rapide car le bytecode est "pré-mâché".

VM pour Java: `java`

VM pour Caml: `ocamlrun`

Modèles d'exécution 3/3: machine virtuelle

La fabrication de bytecode correspond à de la compilation, en plus simple que pour le code natif.

Compilateur de bytecode Java: `javac`

Compilateur de bytecode Caml: `ocamlc`

Le toplevel (ou boucle interactive) `ocaml` enchaîne en permanence: `lecture / ocamlc / ocamlrun / affichage`

Introduction aux machines virtuelles

Cours 2 : La Machine Virtuelle Caml

Pierre Letouzey

Prenom.Nom@pps.jussieu.fr

A la découverte d'un cmo

Phrase exemple: `1+2+3*4/5` dans `test.ml`

Exploration brut: **hexedit** sur `test.cmo`

Le message secret au milieu du cmo:

`103.5.108.4.107.112.113.105.127.2.110.58.57.0`

Les instructions de bytecode associées:

`CONSTINT 5`

`PUSHCONSTINT 4`

`PUSHCONST3`

`MULINT`

`DIVINT`

`PUSHCONST1`

`OFFSETINT 2`

`ADDINT`

(... plus 2 autres sans intérêt)

A la découverte d'un cmo

Exploration structurée:

- `dumpobj` donne directement le bytecode
- `objinfo` donne quelque infos sur la zone finale

Mieux encore:

- `ocaml -dinstr` pour voir le bytecode au vol

Aperçu d'un programme complet

- avec `hexedit`
- avec `dumpobj`

Que fait ocamlrun ?

Voir la fonction `caml_interprete` dans `byterun/interp.c`

Moralement, une grande boucle contenant un switch sur toutes les instructions de bytecode possible.

Cf <http://cadmium.x9c.fr/distrib/caml-instructions.pdf>

Elements de la VM Caml

Éléments principaux:

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

Éléments qui nous intéresseront moins:

- un environnement global (pour les choses "à toplevel")
- un pointeur `env` vers une archive d'environnement
- un registre `extra_args`

Retour sur l'exemple

instruction	accu	pile
CONSTINT 5	5	...
PUSHCONSTINT 4	4	5 ...
PUSHCONST3	3	4 5 ...
MULINT	12	5 ...
DIVINT	2	...
PUSHCONST1	1	2 ...
OFFSETINT 2	3	2 ...
ADDINT	5	...

Instructions de la VM Caml

Gestion de la pile

- **PUSH** : empilement de **accu** sur la **pile**
- **POP n** : dépilage de **n** éléments au 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 est rempli avec **()**.

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* : saute de *o* instructions si **accu** représente **true**
- **BRANCHIFNOT** *o* : saute de *o* instructions si **accu** représente **false**

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**.

A suivre

Même de rien, on a déjà vu environ la moitié des instructions de la VM Caml. La suite au prochain numéro...

Introduction aux machines virtuelles

Cours 3 : La Machine Virtuelle Caml, suite

Pierre Letouzey
Prenom.Nom@pps.jussieu.fr

Mémoire et données Caml

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

- soit cet entier est directement la valeur (*int*, *bool*,...)
- soit cet entier est un pointeur vers un **bloc** mémoire

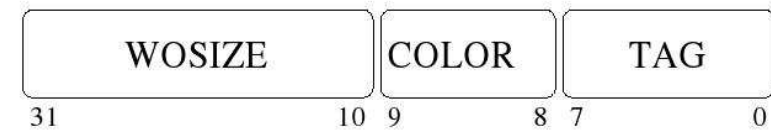
Données codés par des entiers

- int: entiers signés sur 31 ou 63 bits (cf plus tard)
- 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 $t = A \mid B \text{ of } \text{int} \mid C$ 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 : une portion du **tas** (ou **heap**).
L'entier référençant cette zone est juste son adresse (pointeur).
Le bloc commence par un entier d'entête, suivi des données

Entête (sur une machine 32-bits):



Exemple de Tag (cf byterun/mlvalue.h):

- Pour un float: 253
- Pour une fonction (clôture): 247
- 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 !!
Démo: modification magique d'une paire en tableau
- Une liste : [1;2]
Deux blocs de taille 2 (Cf. dessin au tableau) :

$2 * 2^{10}$	1	addr_2nd_bloc
--------------	---	---------------

$2 * 2^{10}$	2	0
--------------	---	---

La séparation entre entiers et pointeurs

Tous les blocs alloués peuvent un jour devenir inutiles:

Un Garbage Collector (GC) fait parfois le ménage.

⇒ Besoin de **parcourir** les données (cf Color).

⇒ Besoin de savoir reconnaître rapidement les pointeurs.

Convention Caml: Tout nombre pair est un pointeur,
Tout nombre impair $2n + 1$ représente en fait 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 V.M. pour les blocs

- **MAKEBLOCK** n k : fabrique un bloc de tag k et de données n éléments retirés du haut de **accu+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. avant.
- **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 V.M. : switch

Cette instruction va permettre de réaliser le `match` de Caml:

switch $l_0^c \dots l_n^c / l_0^{nc} \dots l_m^{nc}$: on regarde 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} .

Les fonctions

Suite et fin de la partie Caml la séance suivante. Reste à voir:

APPLY
APPTERM
CLOSURE
CLOSUREREC
RETURN
RESTART
GRAB
PUSH_RETADDR
(PUSH)OFFSETCLOSURE

Introduction aux machines virtuelles

Cours 4 : les fonctions dans la Machine Virtuelle Caml

Pierre Letouzey

Prenom.Nom@pps.jussieu.fr

Dernières instructions à voir

CLOSURE

APPLY

RETURN

PUSH_RETADDR

ENVACC

CLOSUREREC

OFFSETCLOSURE

GRAB

RESTART

APPTERM

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

Caml: points-clés concernant les fonctions

- fonctions anonymes :
⇒ **une zone de code** (comme les autres fonctions)
- fonctions en arguments et/ou en résultat:
⇒ **des pointeurs vers des blocs**
- accès hors des frontières de la fonction :
⇒ **des environnements**
- récursivité (éventuellement mutuelle): cf plus tard
- application partielle : ⇒ **hibernation** (grab/restart)
- une optimisation: les appels terminaux : cf plus tard

Un exemple élémentaire de fonction

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

<pre>closure L1, 0 push const 2 push const 4 push acc 2 apply 1 mulint return 2</pre>	<pre>L1: acc 0 push const 1 addint return 1</pre>
---	---

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 clôture) de f sera fait ici avec **CLOSURE L1, 2**. Il contient un environnement (de taille 2) après le label de f:

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

Le registre env pointe vers la clôture 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 clôture de f :
OFFSETCLOSURE0
- en faisant un **APPLY** comme précédemment.

Pour les fonctions récursives mutuelles, Caml utilise une même clôture pour tout un paquet mutuel (cf dessin au tableau).

OFFSETCLOSURE n permet ensuite de retrouver un pointeur vers une fonction récursive "voisine".

Appels terminaux

Fréquemment, un appel via **APPLY** est juste avant un **RETURN**. Gâchis en 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... (DEMO).

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 clôture.

Le registre `extra_args` vaut en permanence `nb_args - 1`

Au lancement, toute fonction reçoit au moins 1 arg.

GRAB n mis au début d'une fonction vérifie si les n args suivants sont présents. OK : exécution normale. KO : retourne immédiatement une clôture avec les args présents:

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

Un **RESTART** précède toujours le **GRAB**. Le jour où la clôture est appliqué, il s'exécute, et restaure les args sauvés.

Description précise des instructions

- **CLOSURE o, n** : crée une clôture (bloc de tag 247) contenant le label correspondant à l'offset o suivi de n éléments d'environnement ôtés de **accu+pile**. Un pointeur vers cette clôture est placé dans **accu**.
- **CLOSUREREC o₁ ... o_k, n** : si un seul offset o_1 , cf. **CLOSURE+PUSH**. Sinon, fabrique la clôture suivante, en prenant n éléments d'environnement $e_1 \dots e_n$ dans **accu+pile**.

hdr	pc+o ₁	hdr	pc+o ₂	...	hdr	pc+o _k	e ₁	...	e _n
-----	-------------------	-----	-------------------	-----	-----	-------------------	----------------	-----	----------------

accu reçoit un pointeur vers cette clôture, et on fait **PUSH**

- **PUSH_RETADDR** : empile successivement les valeur des registres `extra_args`, `env` et `pc`

Description précise des instructions

- **APPLY n** : **accu** doit contenir un pointeur vers une clôture, 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 clôture.
- **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** : De toute façon, on commence par enlever **n** éléments obsolètes de la pile. Deux cas 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 clôture, 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 clôture (cf. **APPLY**).

Description précise des instructions

- **ENVACC n** : accède au **n**-ième champs de la clôture 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 couranteLes 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 clôture.
- **APPTERM1 m ... APPTERM3 m** : cf précédent.

Description précise des instructions

- **GRAB n** : si `extra_args` \geq `n`, on continue avec `extra_args` décrémenté de `n`. Sinon on place les (`extra_args+1`) arguments présents sur la pile dans une clôture de la forme:

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

 On

- retourne alors cette clôture dans `accu` et on restaure `pc/env/extra_args` (cf. **RETURN**).
- **RESTART** : `env` doit pointer vers une clôture de la forme précédente. On replace alors les arguments sur la pile, et on incrémente `extra_args` d'autant.

Machine Virtuelle Caml, Conclusion

Quelques portions “subsidiaries” de la C.V.M. non traitées (env global, exceptions, objets).

Important: représentation des données en mémoire (par blocs).

Langage fonctionnel \Rightarrow soin apporté au traitement des fonctions

Questions ?

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>

Introduction aux machines virtuelles

Cours 5 : Java Virtual Machine

Pierre Letouzey

Prenom.Nom@pps.jussieu.fr

Spécification de la JVM

La spécification de la JVM en ligne:

<http://java.sun.com/docs/books/jvms/>

Attention ! ne décrit que le comportement attendu de la JVM (sa *sémantique*). Les codages internes peuvent changer d'une implantation de JVM à une autre.

Traits principaux de la JVM

- **Portabilité:** plus de choses spécifiées qu'en Caml, p.ex. taille des types numériques. Même .class sur \neq archis.
Spec \pm ouverte: plusieurs JVM indépendantes.
Limitations: threads, graphisme
- **Mobilité et Sécurité:** typiquement, pouvoir exécuter des .class venant du web sans (trop de) risque(s).
Pour cela, vérificateurs de bytecode et bacs-à-sable, signatures éventuelles (\neq .cmo et ses appels système/C)

Aspects techniques

- **Typage:** pas totalement statique: object cast, instanceof. Donc des types stockés dans les représentations (\neq Caml).
- **Valeurs:** variées (\neq Caml: "tout est un entier machine")
Des myriades d'instructions selon les types: istore, lstore, fstore, dstore, astore
Attention aux tailles dans la pile: p.ex. dup vs. dup2.
- **Tas:** comme en Caml, les données structurées sont allouées dans un tas (heap), et régies par un ramasse-miette (GC = Garbage Collector).
- **J.I.T.** (compilation Just-In-Time): certaines JVM compilent à la volée vers du code natif.

Organisation de la JVM

- Pour chaque thread, un pc et une pile, faite d'une suite de blocs (ou frame) pour les différentes fonctions en cours d'appel dans cette thread.
- Chaque frame a une taille fixe, connue à l'avance (\neq Caml)
 - une zone de variables locales. Au début: args de la fn
 - une "operand stack", de taille bornée à l'avance.
 - autres éléments, spécifiés (p.ex. renvoi à une zone de constantes) ou non spécifiés (sauvegarde pc de retour)
- De manière globale: une zone de tas partagée pour toutes les threads.

Suite de la comparaison JVM / Caml

- Des appels de fonctions à la fois plus simples (pas d'applications partielles) et plus complexes: selon le cas, invokestatic, invokevirtual, invokespecial ou invokeinterface.
- Appels récursifs possibles mais coûteux (pas de récursion terminal comme en Caml et .Net).
- La représentation des données doit signaler au GC ce qui est un pointeur ou non.
- Un tableswitch/lookupswitch similaire au switch Caml.
- prévu dans la spec de la VM: de la synchronisation (monitor)

Survol des instructions JVM

Voir la spécification de la JVM:

- section 3.11 pour une vision globale des instructions
- section 6 pour le descriptif précis de chaque instruction

Analyse d'un .class

Que contient `Add.class` ?

- En temps normal, un **déssassembleur** est suffisant:
`javap -c -verbose Add`
- Pour une fois, regardons le `.class` avec `hexedit`

Jasmin: un assembleur pour la JVM

Description de `Add2.j`, sa syntaxe, sa compilation via
`jasmin Add2.j`

Exemple de contrainte lié au vérification de `.class`:
la taille de pile