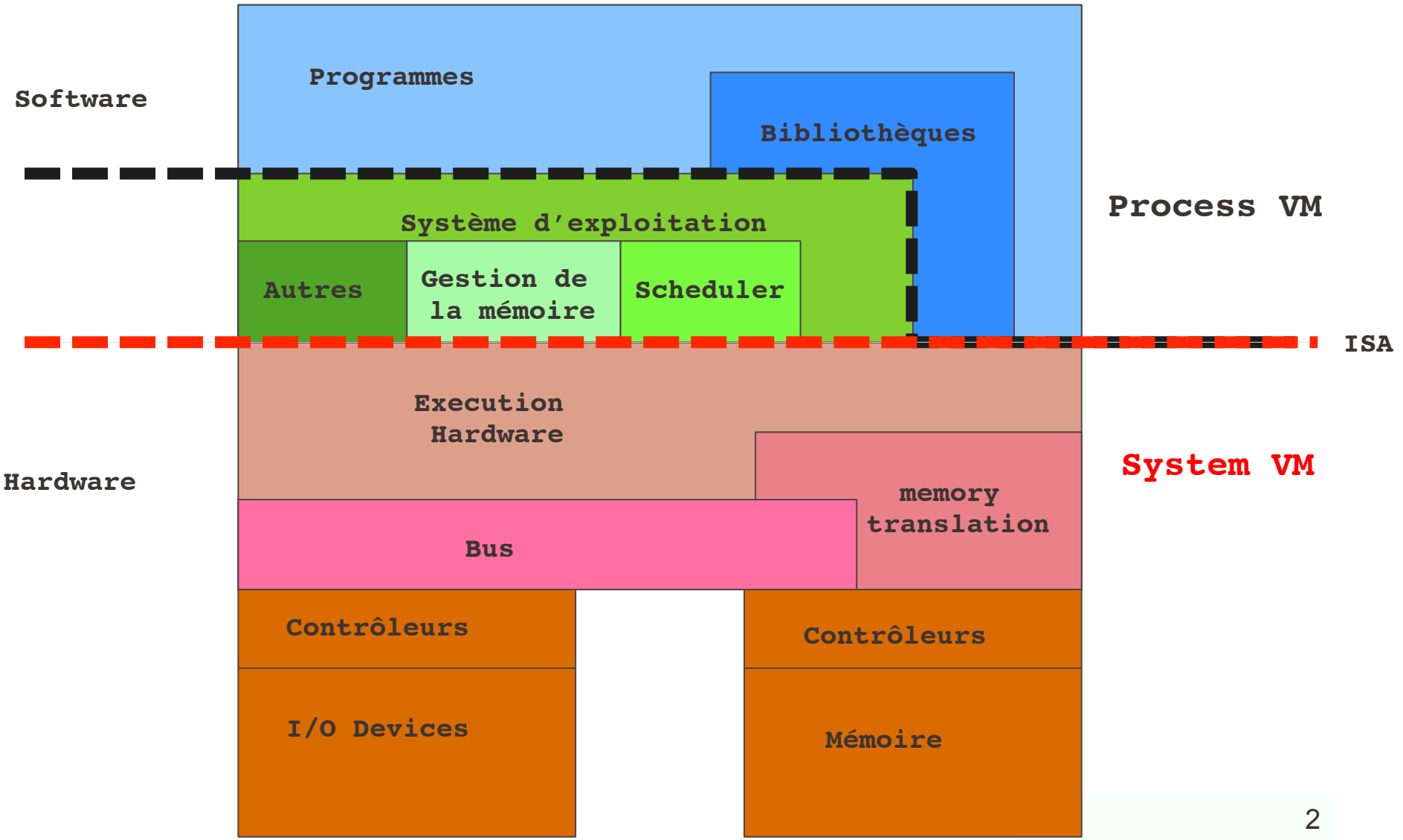
UPMC Paris Universitas
Master Informatique - STL

Compilation Avancée (MI190) 2014
Cours 3 : Machines Virtuelles

Carlos Agon
agonc@ircam.fr

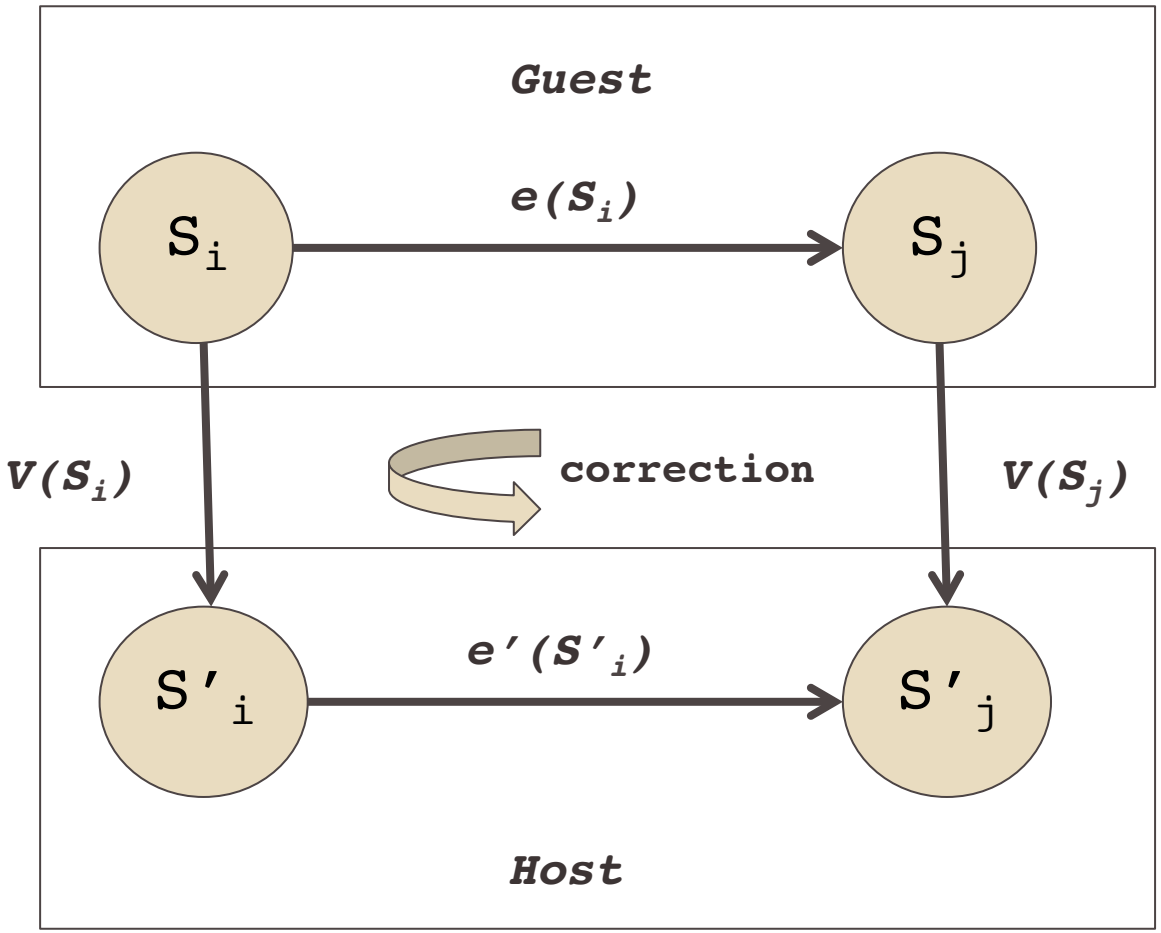
UPMC PARIS UNIVERSITAS

La virtualisation



La virtualisation

MV : *Une machine dématérialisée, sans existence physique: ni silicium, ni engrenage, mais un programme qui exécute un programme!*



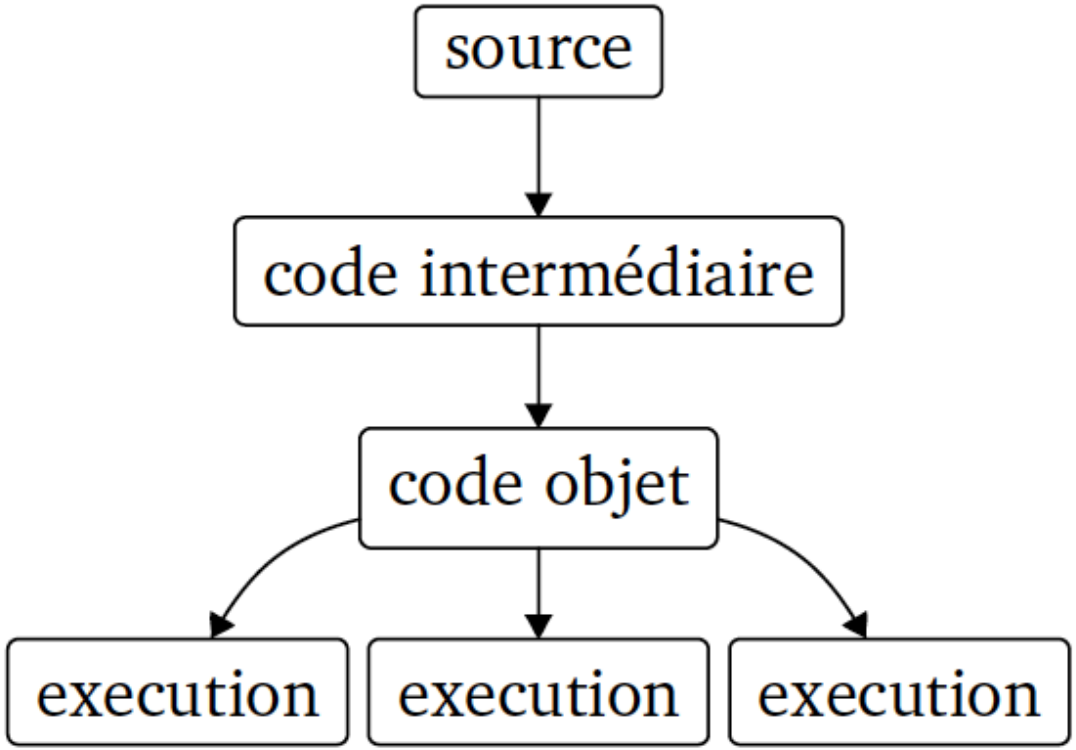
Machine émulée

Machine sur laquelle tourne la MV

VM et approche classique

- **Un compilateur classique génère du code objet ou natif pour une architecture physique donnée (x86, PPC, MIPS. . .)**

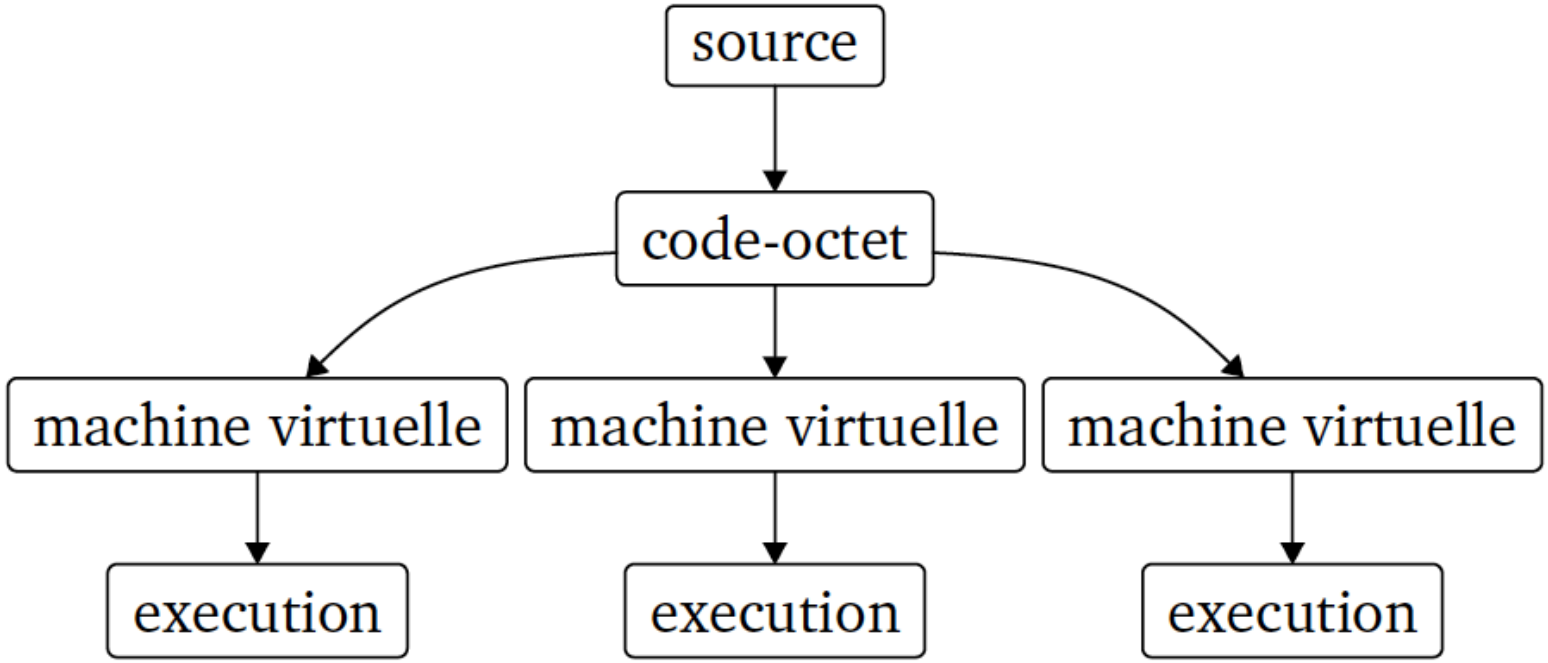
- **n architectures prises en charge n exécutables à distribuer.**



VM et approche classique

- **Le compilateur génère du code-octet pour une MV interprète ou traduit en code natif**

- **Un seul exécutable distribué, n portages de la MV**



Le bytecode

Les instructions, sont bien représentées par une suite d'entiers, mais c'est un évidemment un programme qui les lira et les interprétera.

L'avantage de cette technique est la portabilité. « *Compile once, run everywhere* » comme on dit chez Java.

Mais c'est plus lent

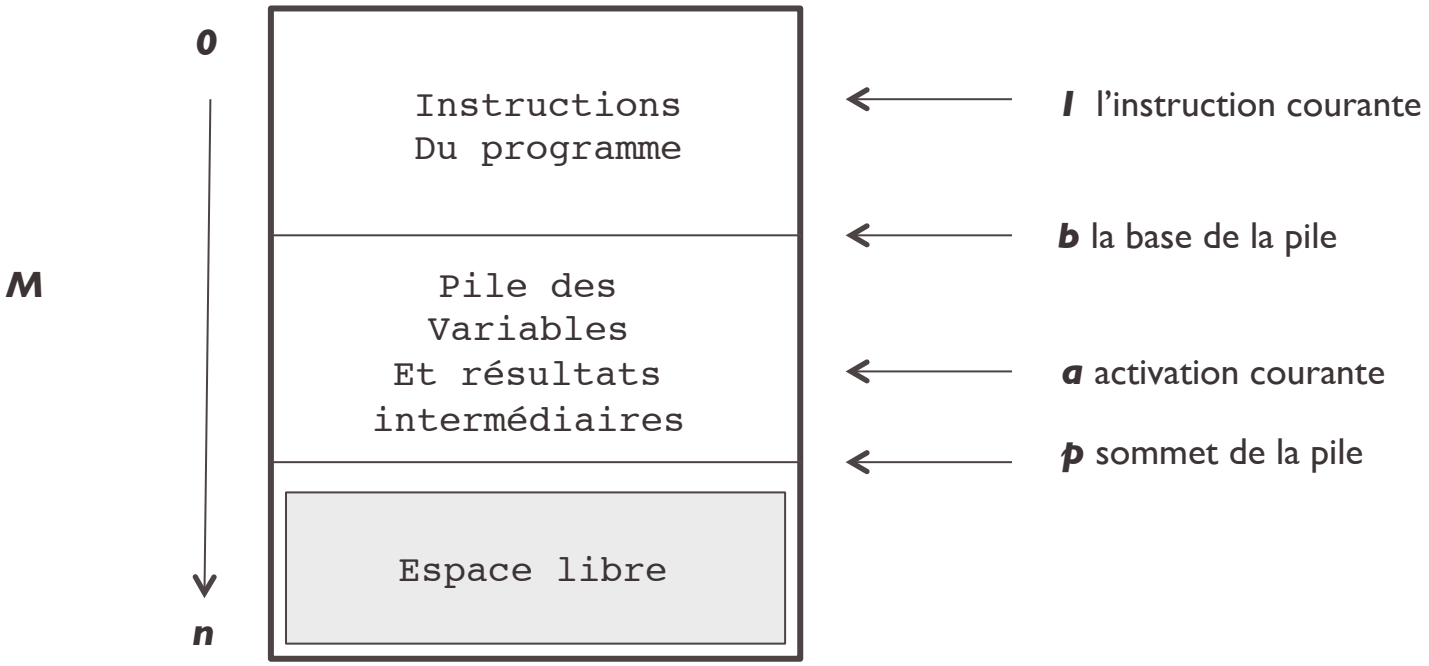
Il est possible, de la première exécution d'une fonction, de transformer le *bytecode* en instructions de la machine hôte, on parle alors de compilation à la volée (Just In Time ou JIT).

P-code

Développée par N.Wirth 1966 pour Pascal

Une unité de stockage *M*, ensemble de mots regroupés en une mémoire, un ensemble *R* de 4 registres et un processeur *P* exécutant un groupe de 39 instructions:

$$Mv = \{M, R, P\}$$



P-code

Begin (Initialement la pile est vide)

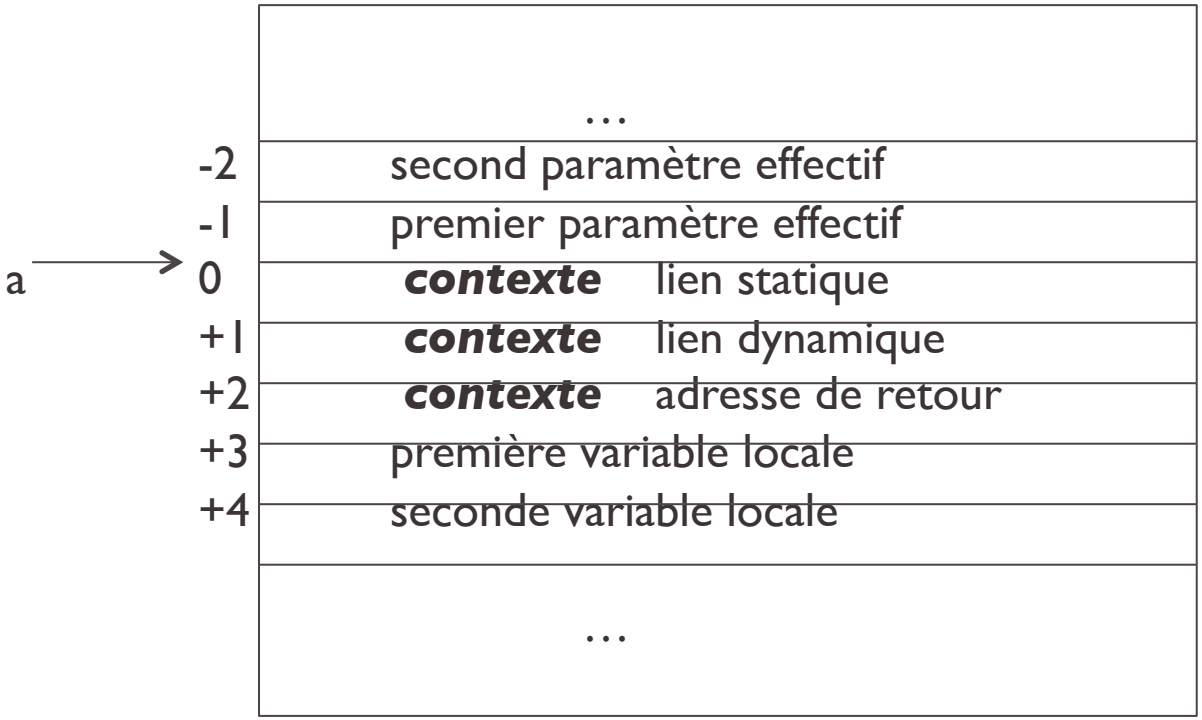
➔ allocation des variables globales sur la pile

➔ activation d'une procédure (*structure d'activation*) définition des variables locales

➔ retour à l'appelant
End

Le programme principal est considéré comme une procédure sans paramètre.

P-code (*structure d'activation)*



Variables par niveau

Historique ses appels

Un copie du registre i

P-code (*Instructions*)

Programme et procédures

Deb	nVar depl lgne	Créer une structure d'activation de taille nvar+3
Fin		Finir l'exécution d'un programme

DebP	nVar depl lgne	Créer une structure d'activation
Apl	niv depl	Activer une procédure
FinP	nPar	Finir l'exécution d'une procédure

P-code (*Instructions*)

Calcul d'adresse et d'affectation

```
AdrV  niv  depl           Empiler l'adresse d'une variable
AdrP  niv  depl           Empiler l'adresse d'un paramètre (indirection)
AdrI  bInf bSup dim lgne  Empiler l'adresse d'une var. indexée
AdrC  depl                Empiler l'adresse d'un champs

ValV  lng                 Empiler la/les valeurs d'une variable
Cnst  val                 Empiler la valeur d'une constante
AffV  lng                 Dépiler la/les valeurs d'une var. & affecter
```

P-code (*Instructions*)

Instructions arithmétiques et logiques

Add	Additionner
Diff	Soustraire (différence)
Mul	Multiplier
Div	Diviser
Mod	Calculer le modulo
Et	Effectuer un ET logique
Ou	Effectuer un OU logique (inclusif)
Cpp	Plus petit que
CNpp	Non plus petit
Ceg	DéteÉgal
CNeg	Non égal
Cpg	Plus grand
CNpg	DétermiNon plus grand

Les 2 instructions suivantes modifient le sommet de pile uniquement.

Neg	Effectuer la négation sommet de pile
InvS	Inverser le signe du sommet de pile

P-code (*Instructions*)

Branchement

Br	depl	Sauter sans condition
BrC	depl	Sauter si condition satisfaite
BrNc	depl	Sauter si condition Non satisfaite

Les instructions BrC et BrNc dépile le sommet de pile puis effectuent le branchement.

Le branchement s'effectue par modification du registre **i:** si *i>=0* alors *i = i+depl+1* sinon *i = i+depl* .

P-code (*Instructions*)

Entrée/sortie

Lire	Lire un entier
Ecr	Écrire un entier suivi de <cr>

Lecture d'un entier à partir du clavier puis, range sa valeur à l'adresse spécifiée par le sommet de pile. Dépile l'adresse.

Ecriture de l'entier sur le sommet de la pile. Dépile la valeur.

P-code (*Instructions*)

Composites

```
AdrL   depl        Empiler l'adresse d'une variable locale
VarL   depl        Empiler la valeur d'une variable locale
AdrG   depl        Empiler l'adresse d'une variable globale
VarG   depl        Empiler la valeur d'une variable globale
Val1               Empiler la valeur d'une variable de lng=1 mot
Aff1               Dépiler une valeur & faire l'affectation lng=1 mot
AplG   depl        Activer une procédure globale
```

P-code (*Interpréteur*)

Exécute les instructions de la machine virtuelle sur une machine hôte.

```
program ExempleA;
   var a,b,c :entier;
   Begin
      lire(a);
      b := 5;
      c := a*b div 3;
      écrire(c);
   end.
```

```
Deb    3   5  1       # program
AdrL   3               # lire(a)
Lire
AdrL   4               # b := 5
Cnst   5
Aff1
AdrL   5               # c := a*b div 3
VarL   3
VarL   4
Mul
Cnst   3
Div
Aff1
VarL   5               # écrire(c)
Ecr
Fin                    # fin
```

```
PROGRAM Factr;
VAR n,result :INTEGER;

PROCEDURE Fact(e :INTEGER;
VAR x :INTEGER);
   VAR y :INTEGER;
   BEGIN
     IF e=0 THEN
           x := 1
     ELSE
        BEGIN
           Fact(e-1,y);
           x := y*e
        END;
   END;

BEGIN
 READ(n);
 Fact(n,result);
 Write(result)
END.
```

```
Deb     2       45    1
DebP    1        5    5   # -- Procédure Fact
VarL   -2
Cnst    0
Ceg
BrNc    10
AdrP    0   -1
Cnst    1
Aff1
Br      20
VarL   -2
Cnst    1
Diff
AdrL    3
AplG   -27                  # Appel récursif de Fact
AdrP    0   -1
VarL    3
VarL   -2
Mul
Aff1
FinP    2                   # -- Fin de la procédure Fact
AdrL    3                   # Programme principal
Lire
VarL    3
AdrL    4
Apl     0   -47            # Appel initial de Fact
VarL    4
Ecr
Fin
```

La machine virtuelle ocamlrun

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
- **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

Besoin d'une autre ?

ZAM = ZINC Abstract Machine

Une machine de Krivine avec évaluation stricte.

Schéma de compilation :

C[n] = **ACCESS(n)**

C[λa] = **GRAB** ; C[a]

C[a b] = **PUSH** (C[b]) ; C[a]

C[a b]s = **REDUCE**(C[b]) ; C[a]

ZINC machine

Transactions de la machine :

Etat avant			Etat après		
Code	Env	Pile	Code	Env	Pile
ACCESS(n);c	e	s	c’	e’	s si e(n)=[c’,e’]
GRAB;c	e	c’[e’].s	c	c’[e’].s	s
REDUCE(c’);c	e	s	c’	e	<c[e]>.s
GRAB;c	e	<c’[e’]>.s	c’	e’	(GRAB;c[e]).s
PUSH(c’);c	e	s	c	e	c’[e].s

ZAM

- 7 registres : PC, SP, ACCU, ENV, EXTRAARGS, TRAPSP, GLOBALDATA
- pile : vecteur de valeurs
- tas : zone d'allocations dynamiques
- représentation uniforme des valeurs
- jeu de 147 instructions (60% sont des raccourcis)

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

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

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

et autres : BNEQ ; BLTINT ; BLEINT ; BGTINT ; BGEINT
Instructions composées (comparaison et branchement)
BEQ n d : si n égal ACCU alors saut de d instruction

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

Appels et retours de 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.

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.

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 !

Autres instructions : GRAB, CLOSUREREC, CCALL, RESTART, ...

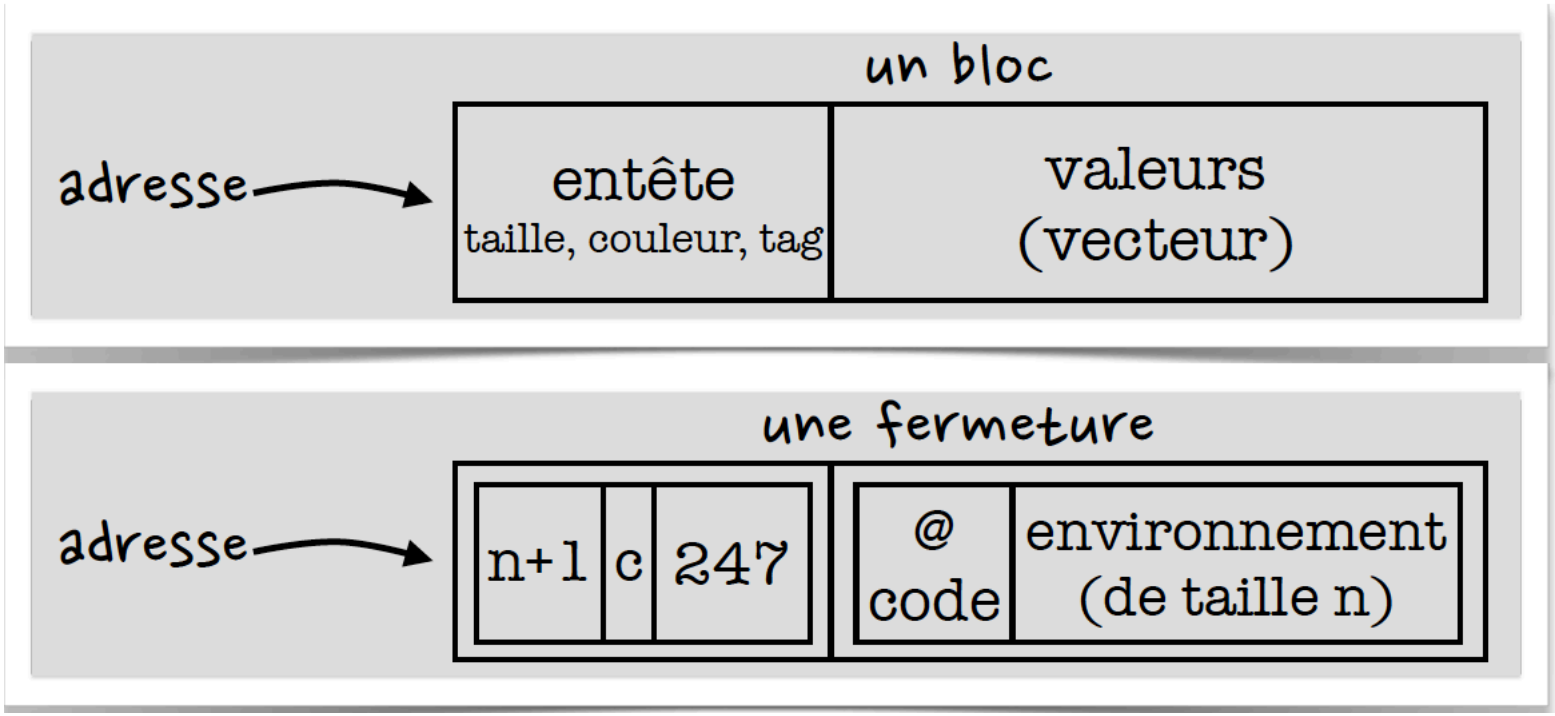
Exemple

```
Exemple
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
```

Appels et retours de fonctions



ZAM Comparasion

```
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
```

ZAM Comparasion

```
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
```

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

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

ZAM Implantation

Interprète de bytecode : boucle de base
Flux d'entrée : opcodes simples et valeurs.

Ex: [NOP ; GOTO ; 0]

```
void run(int code[]) {
        int pc = 0;
        while (TRUE) {
        switch (code[pc]) {
        case NOP:
                pc++;
                break;
        case GOTO:
                pc = code[pc + 1];
                break;
/* ... */
}
}
```

ZAM Implantation

Variante : arguments dans l'opcode, à décoder.
Si les instructions sont homogènes.
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;
/* ... */
}}}}
```

ZAM Implantation

Interprète de bytecode : branchements
On change seulement le pointeur de code.

```
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;
```

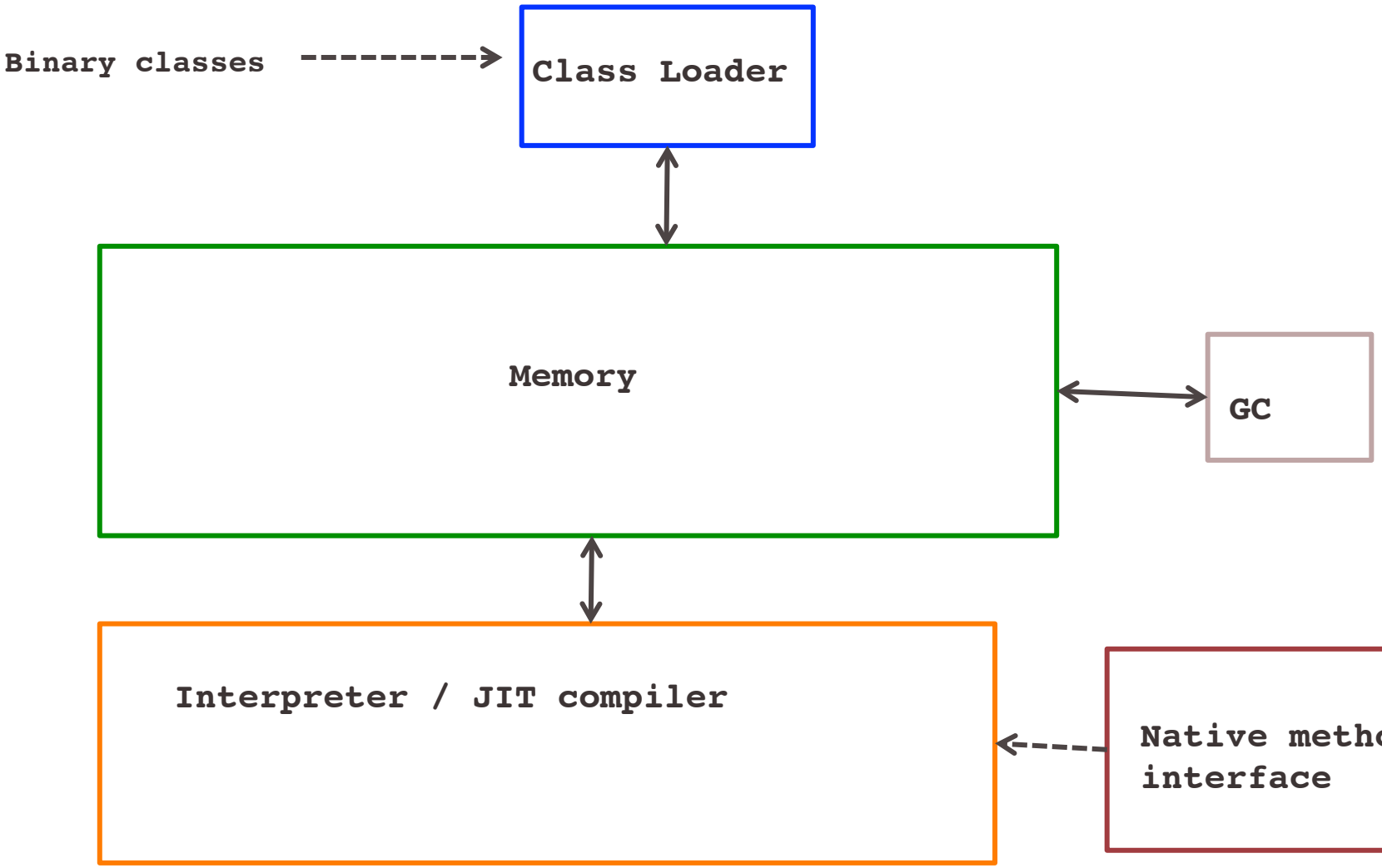
ZAM Implantation

```
Instruct(APPLY2): {
        value arg1 = sp[0];
        value arg2 = sp[1];
        sp -= 3;
        sp[0] = arg1;
        sp[1] = arg2;
        sp[2] = (value)pc;
        sp[3] = env;
        sp[4] = Val_long(extra_args);
        pc = Code_val(accu);
        env = accu;
        extra_args = 1;
        }
```

Lire le code ocamlrun

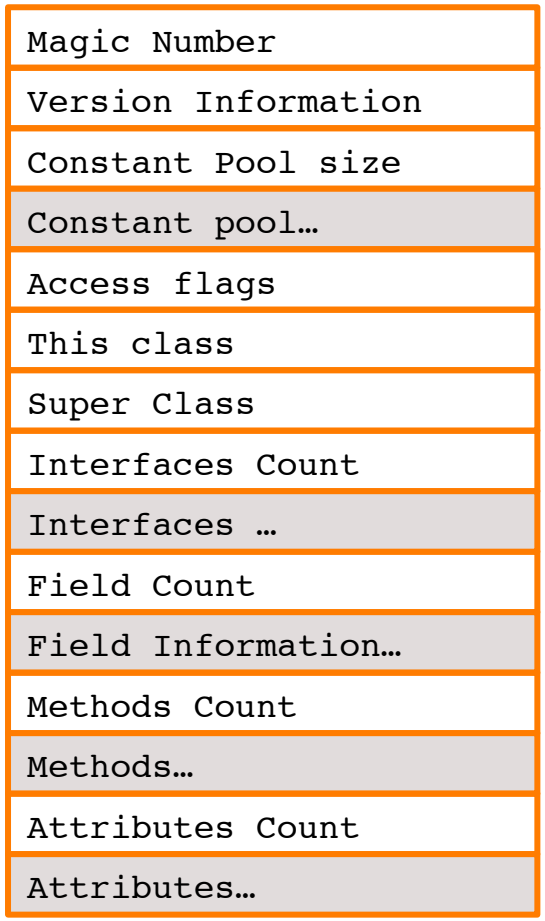
JVM

JVM Modules


Binary classes
Class Loader
Memory
GC
Interpreter / JIT compiler
Native method interface

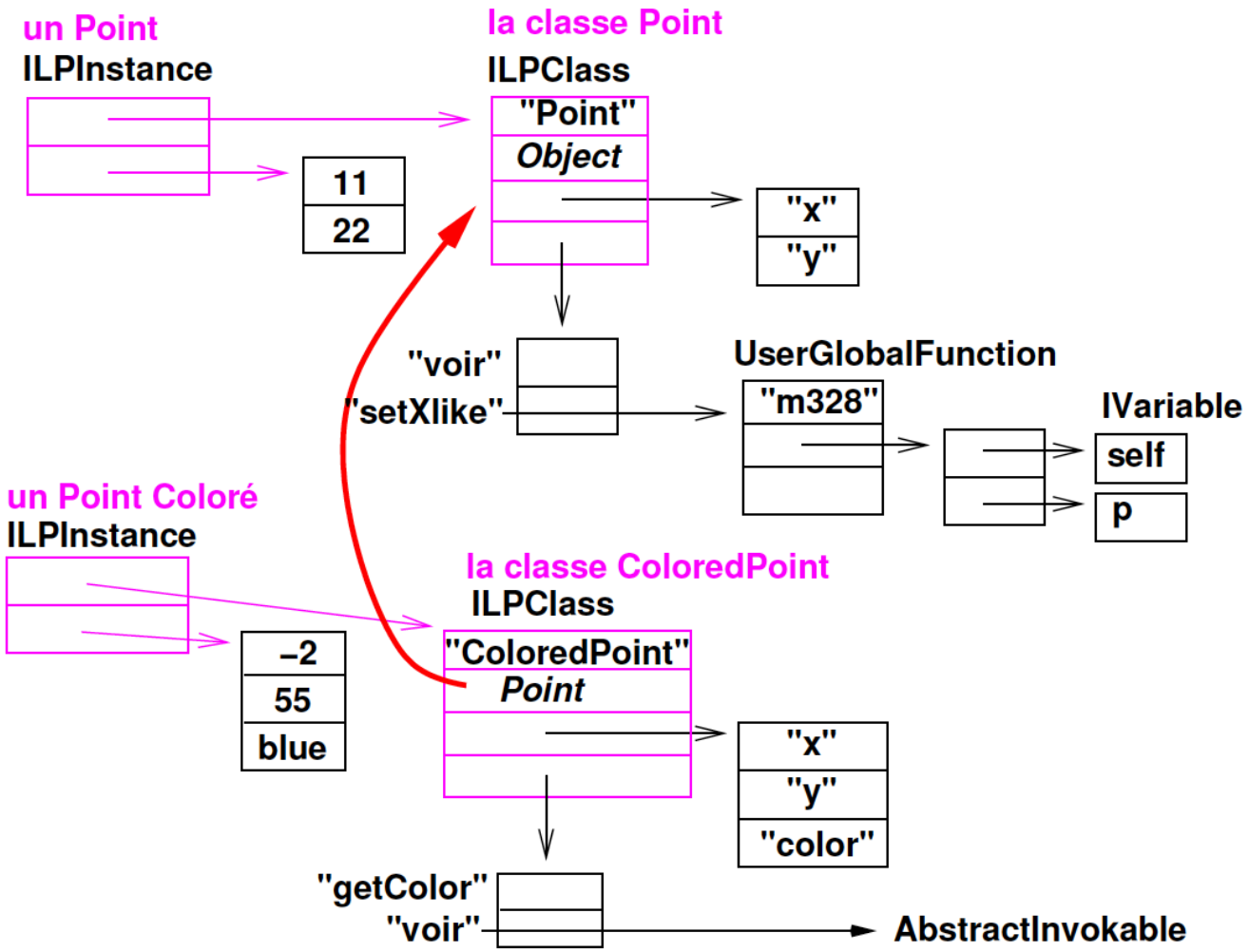
JVM *binary class*

Le format binaire d'une classe est l'interface officielle de la machine



La taille du bloc est connue avant d'être parsé et gardé en mémoire

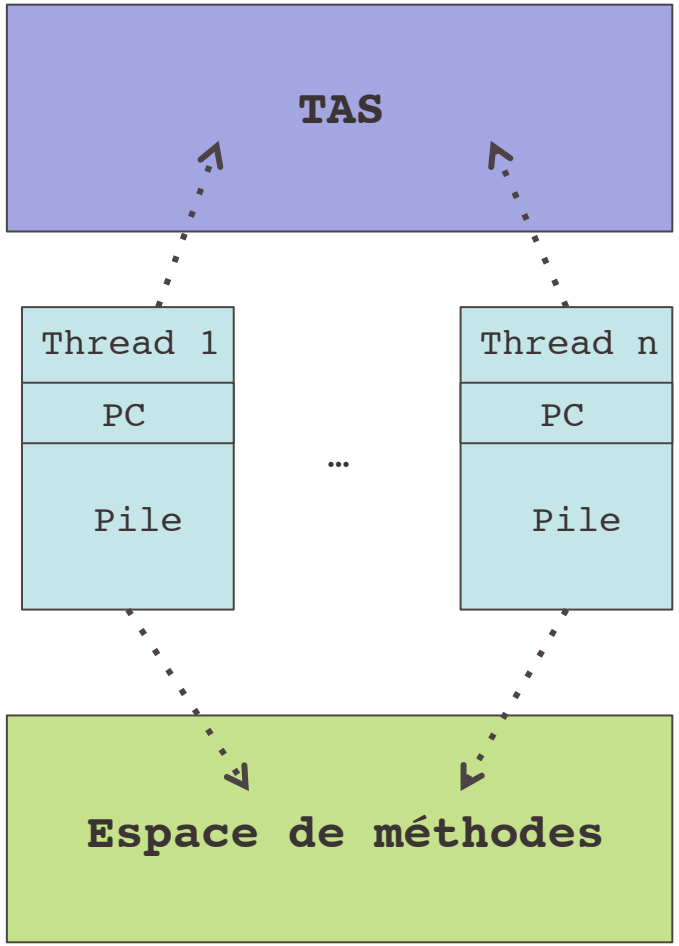
JVM *class loader*



Les espaces de mémoire

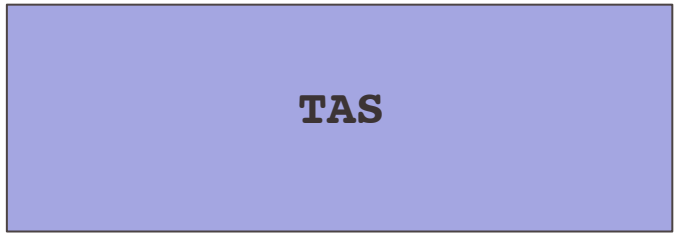
1. Chaque thread a ses propres PC et pile.
2. Le tas est partagé entre les threads.
3. L'espace des Méthodes.

- Le code est partagé entre les threads.
- Le tas est partagé entre les threads.
- Le contenu de toutes ses composantes évolue durant l'exécution.



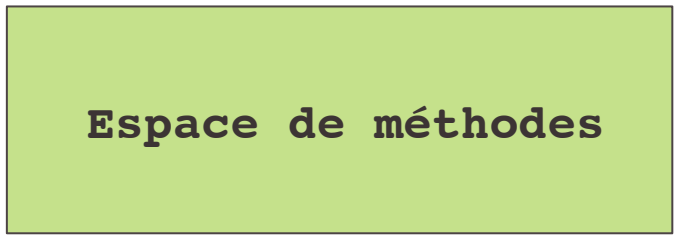
JVM (la mémoire)

Les objets (créés par new)



Blocs d'activation : les variables locales, les arguments, les résultats et les adresses de retour.

Les données données par la définition des classes



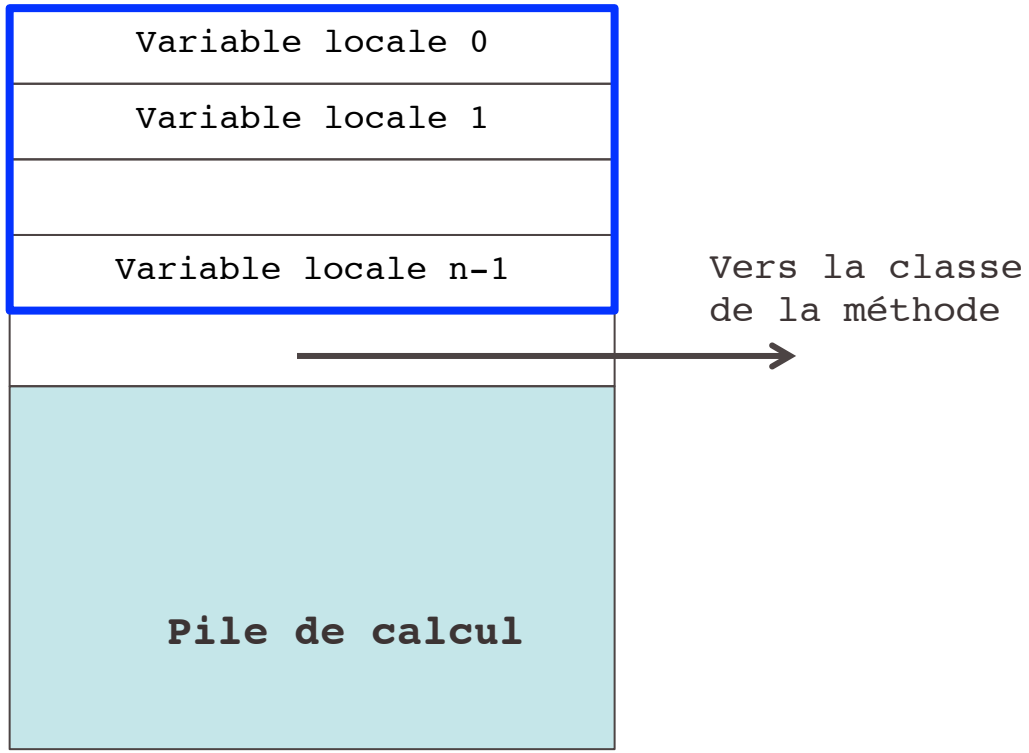
L’espace de méthodes

Pour chaque classe, l’espace des méthodes contient :

- un ensemble de constantes ;
- les champs notés static ;
- des données liées aux méthodes ;
- le code des méthodes et des constructeurs ;
- le code de méthodes spéciales pour l’initialisation des instances de classes et de type d’interface.

L'appel de méthode

- À chaque fois qu'une méthode est invoquée, un bloc d'activation est empilé.
- Quand l'exécution de la méthode est terminée, ce bloc est dépilé.



L'espace pour la pile de calcul est calculable -> l'espace du bloc est connu

Exécution

deux grandes familles de données :

- les données de types primitifs :
  - -les valeurs numériques : entières ou à virgule flottante ;
  - -les booléens ;
  - -les adresses de code.

- Les références qui sont des pointeurs vers des données allouées dans le tas(des instances de classe ou des tableaux).

Les types entiers
byte 8 bits signé
short 16 bits signé
int 32 bits signé
long 64 bits signé
char 16 bits non signé

Les types flottants
float 32 bits simple-précision
double 64 bits double-précision

Les type booléen
1 pour true
0 pour false

Types de références

Trois types de références :
on ne défini pas le nombre de bytes d'un pointer en la Java ISA

1. vers les instances de classes ;
2. vers les implémentations d'interface ;
3. vers les tableaux.

La référence spéciale null ne fait référence à rien et a ces trois types

Jeu d'instructions VM

Les *opcodes* sont stockés sur un octet et commencent par une lettre de type :

i: int
l: long
s: short
b: byte
c: char
f: float
d: double
a: reference

Lecture et écriture des variables locales

xload, **x**load_<n>,
Charge une variable locale au sommet de la pile

xstore, **x**store_<n>,
Écrit le contenu du sommet de la pile dans une variable locale.

bipush, sipush,
Pour mettre un int (32-bit) dans la pile.

ldc, ldc_w, ldc2_w, aconst_null, iconst_m1, iconst_<i>, fconst_<f>, dconst_<d>
D'autres *opcodes* pour empiler une constante.

wide,
Avant une instruction pour augmenter le range des variables locales de 8 à 16 bits

Operations arithmétiques

Addition : iadd , ladd , fadd , dadd .
Soustraction : isub , lsub , fsub , dsub .
Multiplication : imul , lmul , fmul , dmul .
Division : idiv , ldiv , fdiv , ddiv .
Reste de la division : irem , lrem , frem , drem .
Négation : ineg , lneg , fneg , dneg .

Opérations sur la pile

pop, pop2
Dépiler : (1 et 2 single-word).

dup, dup2,
Dupliquer : (1 et 2 single-word).

Swap,
Echanger le top de la pile.

Opérations de flot de contrôle

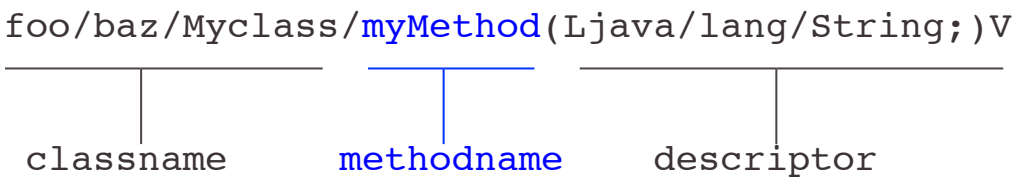
ifeq, iflt, ifle, ifne, ifgt, ifnull, ifnonnull, etc
Branchement conditionnel.

goto, goto_w, jsr, jsr_w, ret
Branchement inconditionnel.

Invocation de méthode

Une page avec toutes les instructions :
http://cs.au.dk/~mis/dOvs/jvmspec/ref-ret.html

Invokevirtual <method-spec>
Invoquer une méthode où <method-spec> est un triplet *classname*, *methodname*, *descriptor*.



Invokeinterface
Invoquer une méthode d'une interface dans une instance qui l'implémente :

invokespecial
Invoquer une initialisation d'instance, une méthode privée ou une méthode d'une classe mère

Invokestatic
Invoquer une méthode de classse statique

Exemple

```
class Rectangle {
protected int sides [] ;
 public Rectangle (int length, int width) {
  sides = new int [2] ;
  sides [Ol = length;
  sides [1] = width;
}

public int perimeter ( ) {
 return 2*(sides[O] +sides [1]);
 }
}
```

```
O: iconst_2
1: aload_O
2: getfield#2; //Field: sides reference
5: iconst_O
6: iaload      //Load sides[0]
7: aload_O
8: getfield#2; //Field: sides reference
11: iconst_l
12: iaload     //Load sides[1]
13: iadd
14: imul
15: ireturn
```

JVM implantations

- Spécification publique :

http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

- Nombreuses implémentations :

JBed - JamaicaVM - JBlend - JRockit - Jamiga - JamVM - Jaos - JC - Jelatine JVM - JESSICA - Jikes RVM - JNode - JOP - Juice - Jupiter - JX –

- La spécification laisse une importante liberté d'implémentation.

Jasmin

Jasmin

Jasmin est un assembler pour la JVM

Pourquoi faire ?

Pour compiler vers la JVM, même si on n'est pas objet

Jasmin (expressions)

Un fichier Jasmin est fait d'expressions de type :

directives
Instructions
labels

Directives

```
 .limit stack 10
```

Instructions

```
ldc    "Hello World"
```

Labels

```
Label:
```

Jasmin (analyse lexicale)

Un fichier Jasmin est fait d'expressions de type :

directives
Instructions
labels

Directives

```
 .limit stack 10
```

Instructions

```
ldc    "Hello World"
```

Labels

```
Label:
```

Jasmin

Comments : ✔foo ; baz ding ✖abc;def

Numbers and Strings : ✔1, 123, .25, 0.03, 0xA ✖1e-10, 'a', '\u123'

Class Names : java/lang/String ‘/’ à la place de ‘.’

Type Descriptors
I speficies an integer,
[Ljava/lang/Thread; is an array of Threads

Jasmin (files)

Information sur la class

```
.source MyClass.j
.class  public MyClass
.super  java/lang/Object
```

Ou une interface

```
.interface public foo
```

implements

```
.class foo
    .super java/lang/Object
    .implements Edible
    .implements java/lang/Throwable
```

Jasmin (field)

.field <access-spec> <field-name> <descriptor> [= <value>]

Où :
<access-spec> : public, private, protected, static, final, volatile, transient

<field-name> : le nom du field.

<descriptor> le descripteur de type.

<value> : initial value of the field (for final fields).

public int foo;	**.field** public foo I
public static final float PI = 3.14;	**.field** public static final PI F = 3.14

Jasmin (method)

.method <access-spec> <method-spec> <statements> **.end method**

Où :
<access-spec> : public, private, protected, static, final, synchronized, native, abstract

<method-spec> :nom et descripteur de type de la méthode

<statements> :le code

int foo(Object a, int b[]) { ... }

.method (Ljava/lang/Object;[I)I foo
…
.end method

Jasmin (method)

Information sur la class

```
.source MyClass.j
.class  public MyClass
.super  java/lang/Object
```

Ou une interface

```
.interface public foo
```

implements

```
.class foo
    .super java/lang/Object
    .implements Edible
    .implements java/lang/Throwable
```

Jasmin (instructions)

Un peu à la JMV voir :

http://jasmin.sourceforge.net/instructions.html

Faire plutôt une démo

Conclusion

Types de machines

- Machines à pile :
  JVM, ZAM2
  Pile pour les variables et arguments
  acc 1 ; push ; acc 2 ; push ; add

- Machines à registres :
  LLVM, Parrot
  Ensemble de registres pour les variables et arguments
  donc plus gros opcodes
  add r1 r2 r0

Machines mono-paradigme, quelques exemples

- Langages procéduraux
  p-machine (Pascal)
- Machines impératives bas-niveau :
  GNU lightning, LLVM
- Langages fonctionnels (λ-calcul)
  - Évaluation stricte (comme en ML) : SECD; FAM; CAM
  - Évaluation paresseuse (comme en haskell) : K; SK; G-machine
- Concurrence (π-calcul, join-calcul)
  Erlang-VM, CHAM
- Objets
  - Prototypes : Smalltalk (Smalltalk), Tamarin, SM (JavaScript)
  - Classes : JVM

Machines multi-paradigme, quelques exemples

- Machines à objets étendues :
  JVM (Java), CLR (.Net)
- Machines fonctionnelles étendues :
  ZAM2 (OCaml)
- Machine hypothétique :
  Parrot (Perl 6)
- Machines impératives bas-niveau :
  GNU lightning, LLVM

Références

James Smith et Ravi Nair
Virtual machines

Xavier Leroy Cours collègue de France

Matthias Puech
Machines Virtuelles Université Paris Diderot – Paris 7

Pierre Letouzey
La machine virtuelle Java Université Paris Diderot – Paris 7

Xavier Leroy
Compiling functional languages

Machine abstraite et génération de code
Philippe Wang –UPMC

Jean-Pierre FOURNIER pour P-code
http://www.infeig.unige.ch/support/cpil/lect/mvp/web.html