

Machines virtuelles

Cours de Compilation Avancée (MU4IN504)

Benjamin Canou & Emmanuel Chailloux
Sorbonne Université

Année 2019/2020 – Semaine 2

Principe général

Machine A

- ▶ Langage compris : L_A
- ▶ Implantation : I_A

Machine B

- ▶ Langage compris : L_B
- ▶ Implantation : I_B

J'ai dans ma poche :

- ▶ un programme en langage L_A
- ▶ une machine de type B

Que faire ?

Principe général

Machine A

- ▶ Langage compris : L_A
- ▶ Implantation : $I_A = L_B$

Machine B

- ▶ Langage compris : L_B
- ▶ Implantation : $I_B = \mu$

J'ai dans ma poche :

- ▶ un programme en langage L_A
- ▶ une machine de type B

Que faire ?

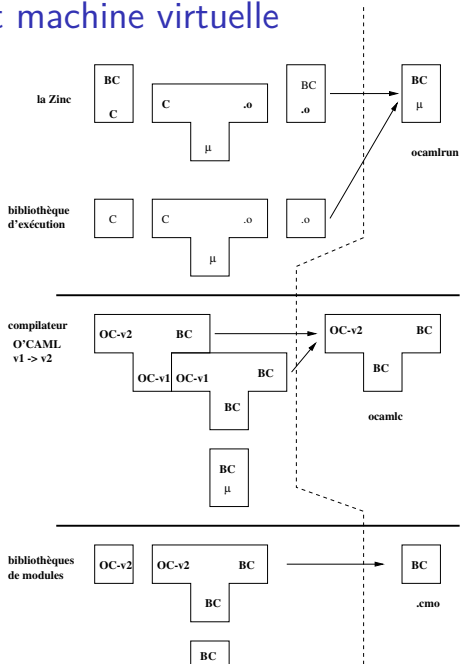
1. Un compilateur $L_A \rightarrow L_B$:

- ▶ Un programme écrit en langage L_B ,
- ▶ transformant mon programme en un équivalent en L_B .

2. Une machine virtuelle A pour ma machine B :

- ▶ Un programme écrit en langage L_B ,
- ▶ capable d'exécuter les programmes en langage L_A .

Compilation et machine virtuelle



Machine virtuelle de plate-forme

(Ce n'est pas le sujet de ce cours)

Dans le cas général, il est trop difficile de recompiler.
Une machine virtuelle est donc la seule possibilité.

Machine PPC

- ▶ Langage compris : asm PPC
- ▶ Implantation : asm x86

Machine x64

- ▶ Langage compris : asm x64
- ▶ Implantation : μ

- ▶ **Autres noms** : émulateur, simulateur, ...
- ▶ **Exemples** : QEMU, DOSBox, VirtualBox, ...

Machine virtuelle applicative

Machine ZAM (ocaml)

- ▶ Langage compris : asm ZAM
- ▶ Implantation : asm x86

Machine x64

- ▶ Langage compris : asm x64
- ▶ Implantation : μ

Dans ce cas, le choix est délibéré :

1. On veut compiler un langage donné.
2. On préfère compiler vers un assembleur adapté.
3. On utilise une machine virtuelle pour l'exécuter.

QUIZZ : Pourquoi ?

Machine virtuelle applicative

Machine ZAM (ocaml)

- ▶ Langage compris : asm ZAM
- ▶ Implantation : asm x86

Machine x64

- ▶ Langage compris : asm x64
- ▶ Implantation : μ

Dans ce cas, le choix est délibéré :

1. On veut compiler un langage donné.
2. On préfère compiler vers un assembleur adapté.
3. On utilise une machine virtuelle pour l'exécuter.

Mots-clefs : *abstraction, portabilité, sécurité, inter-opérabilité*

Machine virtuelle applicative : **portabilité**

Exemples d'implantations de la machine virtuelle OCaml :

- ▶ **ocamlrun** : écrite en C
portable partout où un compilateur C est disponible
- ▶ **obrowser** : écrite en JavaScript
on peut exécuter un programme caml dans un navigateur
- ▶ **ocapic** : écrite en assembleur PIC
un langage de haut niveau sur microcontrôleurs
- ▶ **omicrob** : écrite en C
portable sur plusieurs familles de microcontrôleurs

Implantations alternatives :

- ▶ **OpenJDK** : pour la JVM d'oracle
- ▶ **Mono** : pour la CLR

Machine virtuelle applicative : **abstraction**

- ▶ **Modèle sémantique clair et figé :**
 - ▶ plus facile de théoriser,
 - ▶ exécutables plus durables,
 - ▶ portabilité facile, y compris aux tiers.
- ▶ **Instructions de haut niveau :**
 - ▶ moins d'étapes de compilation,
 - ▶ support du langage → compilation plus simple,
 - ▶ schéma de compilation unique.

Machine virtuelle applicative : **inter-opérabilité**

- ▶ **Entre les langages :**
VB.Net peut appeler des fonctions F# dans la CLR.
- ▶ **Entre les plate-formes :**
représentation spécifiée des chaînes, taille des entiers, etc.
(ex : Sauvegarde sous Win/x86, relecture sous GNU/PPC).
- ▶ **Entre les machines :**
primitives réseau spécifiées \Rightarrow communication plus facile

Machine virtuelle applicative : **sécurité**

- ▶ Exécution isolée (*sandboxing*)
- ▶ Assembleur typé
- ▶ Vérification avant exécution (*bytecode verifier*)
- ▶ Instrumentation (traces, journalisation, etc.)

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 Lisp) : LLM3
 - ▶ Évaluation stricte (comme en ML) : *SECD*, *FAM*, *CAM*
 - ▶ Évaluation paresseuse (comme en Haskell) : *K*, *SK*, *G*-machine
- ▶ Objets
 - ▶ Prototypes : Smalltalk (Smalltalk),
Tamarin, Spider Monkey (JavaScript)
 - ▶ Classes : JVM, CLI

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 / CLI

- ▶ 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 pour les environnements et les points de choix
 - ▶ la piste (trail) pour enregistrer les liaisons des variables et pouvoir les défaire lors d'un backtrack.
- ▶ références :
 - ▶ D. Warren - "An abstract Prolog instruction set".
<http://www.ai.sri.com/pubs/files/641.pdf>
 - ▶ P. Codognet and D. Diaz. "wamcc: Compiling Prolog to C" (ICLP'95). <http://cri-dist.univ-paris1.fr/diaz/publications/WAMCC/iclp95.pdf>

Machines multi-paradigmes, 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

- ▶ Concurrency (π -calcul, join-calcul)
Erlang-VM, CHAM

Types de machines

- ▶ Machines à pile : JVM, ZAM2

- ▶ Machines à registres : Dalvik, LLVM, Parrot

Types de machines

- ▶ Machines à pile : JVM, ZAM2
 - ▶ Pile pour les variables et arguments

- ▶ Machines à registres : Dalvik, LLVM, Parrot
 - ▶ Ensemble de registres pour les variables et arguments

Types de machines

- ▶ Machines à pile : JVM, ZAM2
 - ▶ Pile pour les variables et arguments
 - ▶ → bruit pour accéder aux arguments
`acc 1 ; push ; acc 2 ; push ; add`
- ▶ Machines à registres : Dalvik, LLVM, Parrot
 - ▶ Ensemble de registres pour les variables et arguments
 - ▶ → plus gros opcodes
`add r1 r2 r0`

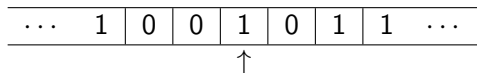
Types de machines

- ▶ Machines à pile : JVM, ZAM2
 - ▶ Pile pour les variables et arguments
 - ▶ → bruit pour accéder aux arguments
`acc 1 ; push ; acc 2 ; push ; add`
 - ▶ → triche : variables (JVM)
- ▶ Machines à registres : Dalvik, LLVM, Parrot
 - ▶ Ensemble de registres pour les variables et arguments
 - ▶ → plus gros opcodes
`add r1 r2 r0`
 - ▶ → triche : pile d'appels (Dalvik) (registres fixes/frame)

Une machine impérative

La machine de Turing

- ▶ Bande infinie de cases.
- ▶ Chaque case \in alphabet fini (ex. 1 ou 0).
- ▶ Une tête de lecture pointe sur une case précise.



- ▶ État \in ensemble fini (avec états spéciaux départ et fin).
- ▶ Table de transitions de la forme :

(état courant, valeur de la case)



(état suivant, nouvelle valeur, direction \in {gauche, droite})

La machine de Turing

- ▶ Capable d'encoder n'importe quelle fonction calculable.
- ▶ Arrêt indécidable.
- ▶ Capable de *simuler* un ordinateur moderne.
- ▶ Deux façons de voir :
 1. bande = mémoire, automate = code
 2. bande = code et mémoire mélangés, automate = processeur
- ▶ Insuffisante pour encoder la concurrence.

Programmation fonctionnelle (rappels)

Modèle des langages fonctionnels : le λ -calcul

Trois possibilités pour un terme T :

1. Variable : x
2. Application : $T_1 T_2$
3. Abstraction : $\lambda x. T$

Très simple mais \equiv à une machine de Turing.

Évaluation du λ -calcul

Évaluation formelle : β -réduction :

1. On choisit un redex $(\lambda x. T_1) T_2$ dans l'expression,
2. on remplace x par T_2 dans T_1 ,
3. on remplace le redex par ce résultat.
4. **Normalisation** : on continue tant qu'il y a des redexes.

Évaluation du λ -calcul

Stratégies d'évaluation :

► Appel par nom :

1. On remplace le paramètre par l'argument dans le corps,
2. on réduit le corps ainsi modifié.

► Appel par valeur :

1. On réduit l'argument,
2. on remplace le paramètre par l'argument réduit dans le corps,
3. on réduit le corps.

► Appel par nécessité :

1. On transforme l'argument en une fonction (*glacçon*),
2. la première fois où l'argument est utilisé, la fonction le calcule,
3. les fois suivantes, il redonne la valeur déjà calculée.

Extensions du λ -calcul

Par encodage (ex: les couples) :

- ▶ Construction : $CONS := \lambda x.\lambda y.(\lambda f.f x y)$
- ▶ Projection 0 : $P0 := \lambda c.c (\lambda a.\lambda b.a)$
- ▶ Projection 1 : $P1 := \lambda c.c (\lambda a.\lambda b.b)$
- ▶ Échange : $SWAP := \lambda c.c (\lambda x.\lambda y.CONST y x)$

Par ajout de termes/opérations de base (ex: entiers) :

- ▶ $val ::= var \mid int \mid add \mid sub$
- ▶ $term ::= \lambda var.term \mid term term \mid val$
- ▶ Ex: $\lambda x.\lambda y.(add x (sub y 3))$

Évaluation

Comment évaluer $CONS\ 1\ 2$ en pratique ?

- ▶ Réécriture de termes : $CONS\ 1\ 2 = \lambda f.f\ 1\ 2$
en pratique, difficile de modifier le code du programme.

- ▶ Fermetures :

$CONS\ 1\ 2$

→ $(\lambda x.\lambda y.\lambda f.f\ x\ y)[]\ 1\ 2$

→ $(\lambda y.\lambda f.f\ x\ y)_{[(x,1)]}\ 2$

→ $(\lambda f.f\ x\ y)_{[(x,1);(y,2)]}$

On crée une **fermeture** :

- ▶ corps de la fonction,
- ▶ environnement : valeurs des variables lors de l'abstraction.

Lors de l'appel, on exécute le corps dans l'environnement, augmenté de la valeur du paramètre.

Codage des couples

En OCaml :

```
1 # let cons = fun x y f -> f x y ;;
2 val cons : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c = <fun>
3 # let p0 = fun c -> c (fun a b -> a) ;;
4 val p0 : (('a -> 'b -> 'a) -> 'c) -> 'c = <fun>
5 # let p1 = fun c -> c (fun a b -> b);;
6 val p1 : (('a -> 'b -> 'b) -> 'c) -> 'c = <fun>
7 # let swap = fun c -> (fun x y -> cons y x);;
8 val swap : 'a -> 'b -> 'c -> ('c -> 'b -> 'd) -> 'd = <fun>
9
10 # let c1 = cons 1 2 ;;
11 val c1 : (int -> int -> '_a) -> '_a = <fun>
12 # let q = p0 c1 ;;
13 val q : int = 1
14 # let r = p1 c1 ;;
15 val r : int = 2
16 # c1 ;;
17 - : (int -> int -> int) -> int = <fun>
```

Autres exemples d'évaluation en OCaml

```
1 # let f x y z = x + y + z ;;
2 val f : int -> int -> int -> int = <fun>
3 # f 1 ;;
4 - : int -> int -> int = <fun>
5 # let g = f 1 2 ;;
6 val g : int -> int = <fun>
7 # g 10 ;;
8 - : int = 13
9 # g 10 20 ;;
10 Error: This function is applied to too many arguments;
11 maybe you forgot a `;'
12 #
```


Un Évaluateur de λ -calcul

Fabriquer une valeur calculable de la forme $\text{terme}_{\text{env}}$.

env	terme	pile	\rightarrow env	term	pile
e	FA	S	$\rightarrow e$	F	$A_e :: S$
e	$\lambda x.C$	$a :: S$	$\rightarrow (x, a) :: e$	C	S
$(= x, A_{e'}) :: e$	x	S	$\rightarrow e'$	A	S
$(\neq x, _) :: e$	x	S	$\rightarrow e$	x	S

Une machine fonctionnelle

La machine de Krivine

- ▶ Exécute du code-octet, compilé depuis un lambda terme,
- ▶ code-octet complètement linéaire (suite d'opcodes),
- ▶ trois opcodes très simples.

De quoi a-t'on besoin ?

env	terme	pile	env	term	pile
e	FA	$S \rightarrow e$	e	F	$A_e :: S$
e	$\lambda x.C$	$a :: S \rightarrow (x, a) :: e$	$(x, a) :: e$	C	S
$(= x, A_{e'}) :: e$	x	$S \rightarrow e'$	e'	A	S
$(\neq x, _) :: e$	x	$S \rightarrow e$	e	x	S

La machine de Krivine

- ▶ Exécute du code-octet, compilé depuis un lambda terme,
- ▶ code-octet complètement linéaire (suite d'opcodes),
- ▶ trois opcodes très simples.

De quoi a-t'on besoin ?

env	terme	pile	env	term	pile
e	FA	S	$\rightarrow e$	F	$A_e :: S$
e	$\lambda x.C$	$a :: S$	$\rightarrow (x, a) :: e$	C	S
$(= x, A_{e'}) :: e$	x	S	$\rightarrow e'$	A	S
$(\neq x, _) :: e$	x	S	$\rightarrow e$	x	S

1. **PUSH addr**

on repère les termes par l'adresse de leur code compilé

La machine de Krivine

- ▶ Exécute du code-octet, compilé depuis un lambda terme,
- ▶ code-octet complètement linéaire (suite d'opcodes),
- ▶ trois opcodes très simples.

De quoi a-t'on besoin ?

env	terme	pile	env	term	pile
e	FA	S	$\rightarrow e$	F	$A_e :: S$
e	$\lambda x.C$	$a :: S$	$\rightarrow (x, a) :: e$	C	S
$(= x, A_{e'}) :: e$	x	S	$\rightarrow e'$	A	S
$(\neq x, _) :: e$	x	S	$\rightarrow e$	x	S

1. PUSH *addr*

on repère les termes par l'adresse de leur code compilé

2. GRAB

La machine de Krivine

- ▶ Exécute du code-octet, compilé depuis un lambda terme,
- ▶ code-octet complètement linéaire (suite d'opcodes),
- ▶ trois opcodes très simples.

De quoi a-t'on besoin ?

env	terme	pile		env	term	pile
e	FA	S	\rightarrow	e	F	$A_e :: S$
e	$\lambda x.C$	$a :: S$	\rightarrow	$(x, a) :: e$	C	S
$(= x, A_{e'}) :: e$	x	S	\rightarrow	e'	A	S
$(\neq x, _) :: e$	x	S	\rightarrow	e	x	S

1. **PUSH** *addr*

on repère les termes par l'adresse de leur code compilé

2. **GRAB**

3. **ACCESS** *idx*

on repère les variables par leur indice de de Bruijn

Machine virtuelle

code corrigé (6.2.19)

```
type closure = C of int * closure list
type instr = ACCESS of int | PUSH of int | GRAB

let interprete code =
  let rec interp env pc stack =
    match (List.nth code pc) with
    | ACCESS n ->
      (try
        let (C (n,e)) = List.nth env n in
        interp e n stack
        with ex -> (C (pc, env)))
    | PUSH n ->
      interp env (pc+1) ((C (n,env))::stack)
    | GRAB ->
      (match stack with
      | [] -> C (pc,env)
      | so::s -> interp (so::env) (pc+1) s)
  in
  interp [] 1 []
```

Compilation vers la machine de Krivine (exos en TD)

Assembleur avec étiquettes :

```
type instr =  
  | IPUSH of lbl  
  | IGRAB  
  | IACCESS of int  
  | ILABEL of lbl
```

Schéma de compilation \mathcal{C} :

$$\mathcal{C}_e(T_1 T_2) = \text{IPUSH } l ; \mathcal{C}_e(T_1) ; \text{ILABEL } l ; \mathcal{C}_e(T_2)$$

$$\mathcal{C}_e(\lambda x. T) = \text{IGRAB} ; \mathcal{C}_{x::e}(T)$$

$$\mathcal{C}_e(x) = \text{IACCESS } nth(x, e)$$

Puis on fait une passe de suppression des étiquettes.

La ZAM

La machine de Krivine est-elle utilisable en pratique ?

Oui, mais : on ne peut pas utiliser l'appel par nom en pratique, si on utilise des opérations de base (opérations arithmétiques, etc).

1. Évaluation stricte (la ZAM : machine de Caml)
2. Évaluation paresseuse.

⇒ voir prochain cours pour la ZAM

Présentation de la machine $MV_{LU3IN018}$

$MV_{LU3IN018}$ est une machine à pile

Plan de présentation: :

- ▶ représentation uniforme des valeurs manipulées
- ▶ une machinerie : du code et des zones mémoire
- ▶ peu d'instructions : une quinzaine mais des primitives pour les calculs d'expression
- ▶ manipulant des valeurs fonctionnelles
- ▶ et ayant un récupérateur automatique de mémoire

nécessité de conserver une information de type sur les valeurs :

- ▶ des valeurs immédiates (représentées par des entiers)
 - ▶ des entiers, des booléens (TRUE et FALSE), des numéros de primitives
- ▶ des valeurs allouées dans le tas
 - ▶ des blocs de taille connue
 - ▶ des valeurs fonctionnelles

Machinerie $MV_{LU3IN018}$: zones mémoire

un compteur ordinal pc (index de l'instruction à exécuter)
et un pointeur de pile sp et des instructions manipulant :

- ▶ du code
- ▶ des primitives
- ▶ un environnement global
- ▶ une pile
- ▶ un tas
- ▶ des environnements locaux (cadres d'appel pour les fonctions et primitives)

Les globales et la pile sont vues comme des vecteurs.

Instructions (1)

- ▶ Instructions (1) : Environnement global
 - ▶ GALLOC : allocation d'une variable globale
 - ▶ GFETCH n : lecture de la variable globale numéro n
 - ▶ GSTORE n : affectation de la variable globale numéro n
- ▶ Instructions (2) : Environnement local
 - ▶ FETCH n : lecture de la variable locale numéro n
 - ▶ STORE n : affectation de la variable locale numéro n
- ▶ Instructions (3) : Opération sur la pile
 - ▶ POP : dépilement
 - ▶ PUSH *type val* : empilement de la valeur *val* de type *type* ;

Instructions (2)

- ▶ Instructions (4) : cadre d'appel
 - ▶ CALL *a* : appel d'une fonction ou d'une primitive d'arité *a*
 - ▶ RETURN : retour de fonction : destruction du cadre d'appel courant

création d'un nouveau cadre d'appel pour le CALL et destruction de cadre courant au RETURN

- ▶ Instructions (5) : opération de contrôle, modification du compteur ordinal *pc*
 - ▶ JUMP *l* : saut inconditionnel au label *l*
 - ▶ JFALSE *l* : saut conditionnel au label *l* si le sommet de pile est faux
- ▶ Instructions (6) : primitives
 - ▶ numérotation dans `constants.h` créé par le compilateur :

```
1 /* Constantes pour les primitives */
2 /** primitive + */
3 #define P_ADD 0
4 /** primitive - */
5 #define P_SUB 1
6 /** primitive * */
7 #define P_MUL 2
```

Exemples (1)

- ▶ programme addition de deux variables :

```
1  var a = 10 ;  
2  var b = 20 ;  
3  a + b ;
```

- ▶ byte-code produit

```
1  GALLOC  
2  PUSH INT 10  
3  GSTORE 0  
4  GALLOC  
5  PUSH INT 20  
6  GSTORE 1  
7  GFETCH 1  
8  GFETCH 0  
9  PUSH PRIM 0  
10 CALL 2  
11 POP
```

Exemples (2)

- ▶ conditionnelle :

```
1  if (true) {
2      42;
3  } else {
4      38;
5  }
```

- ▶ byte-code produit

```
1  PUSH BOOL TRUE
2  JFALSE L1
3  PUSH INT 42
4  POP
5  JUMP L2
6  L1:
7  PUSH INT 38
8  POP
9  L2:
```