Machines Virtuelles

MV6

Matthias Puech

Université Paris Diderot — Paris 7

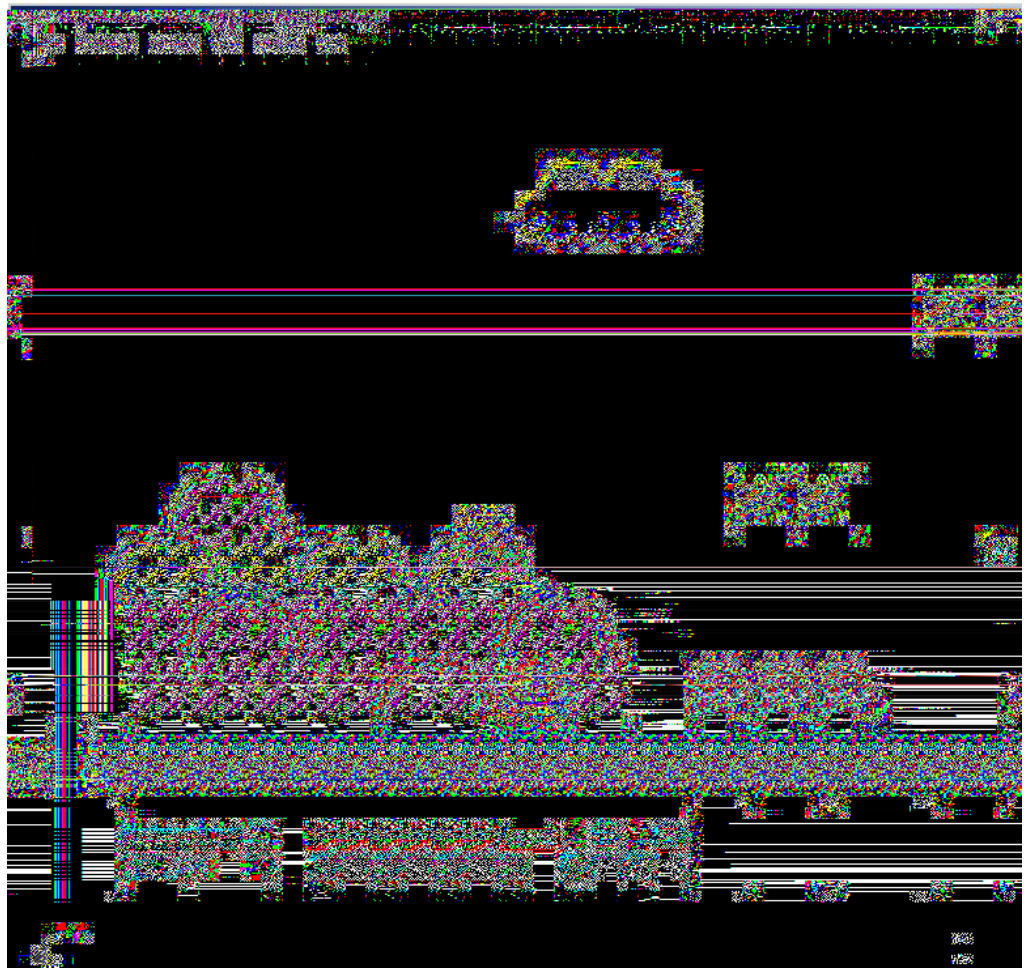
2013

Plan du cours

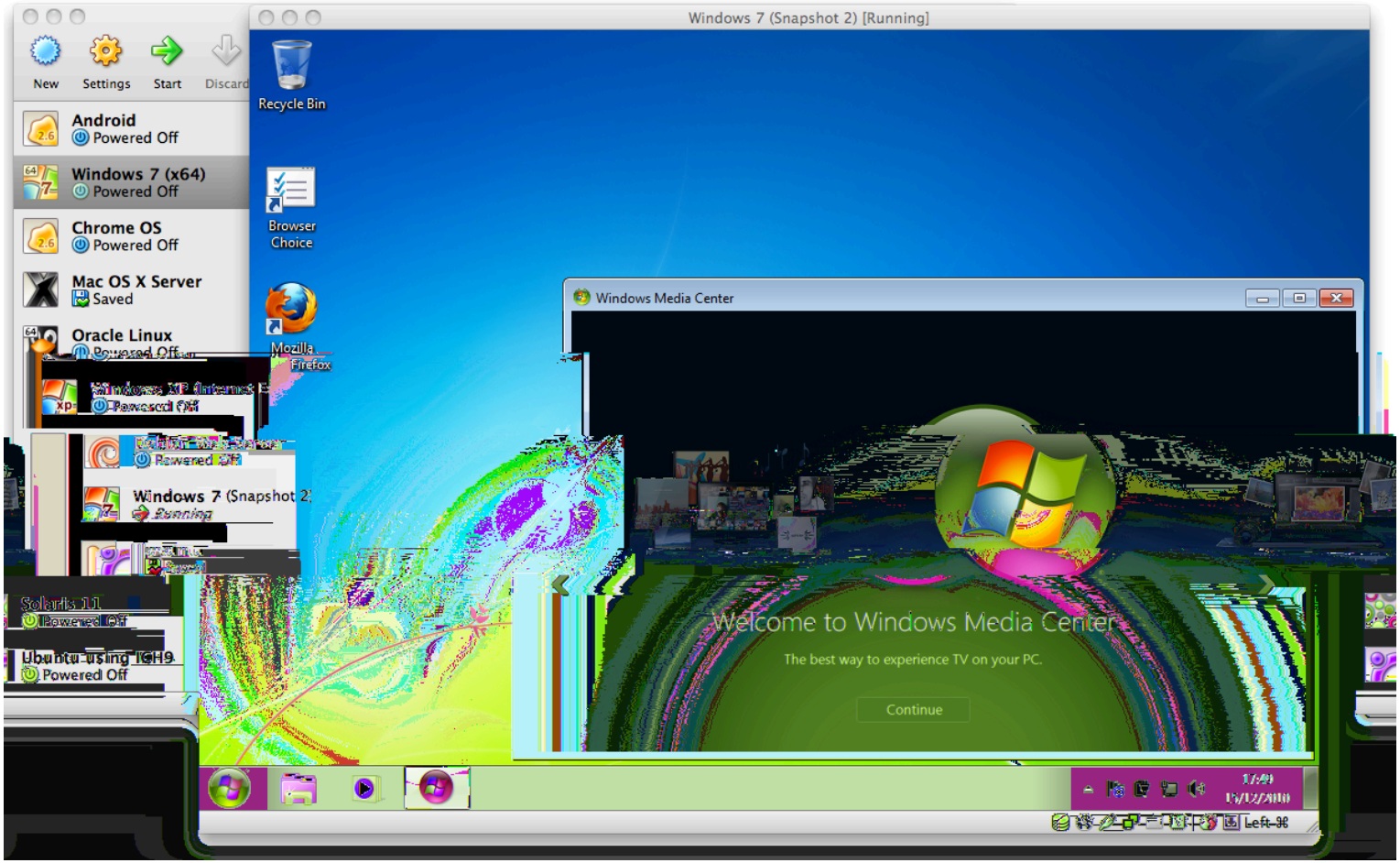
Qu’est-ce qu’une machine virtuelle?

?

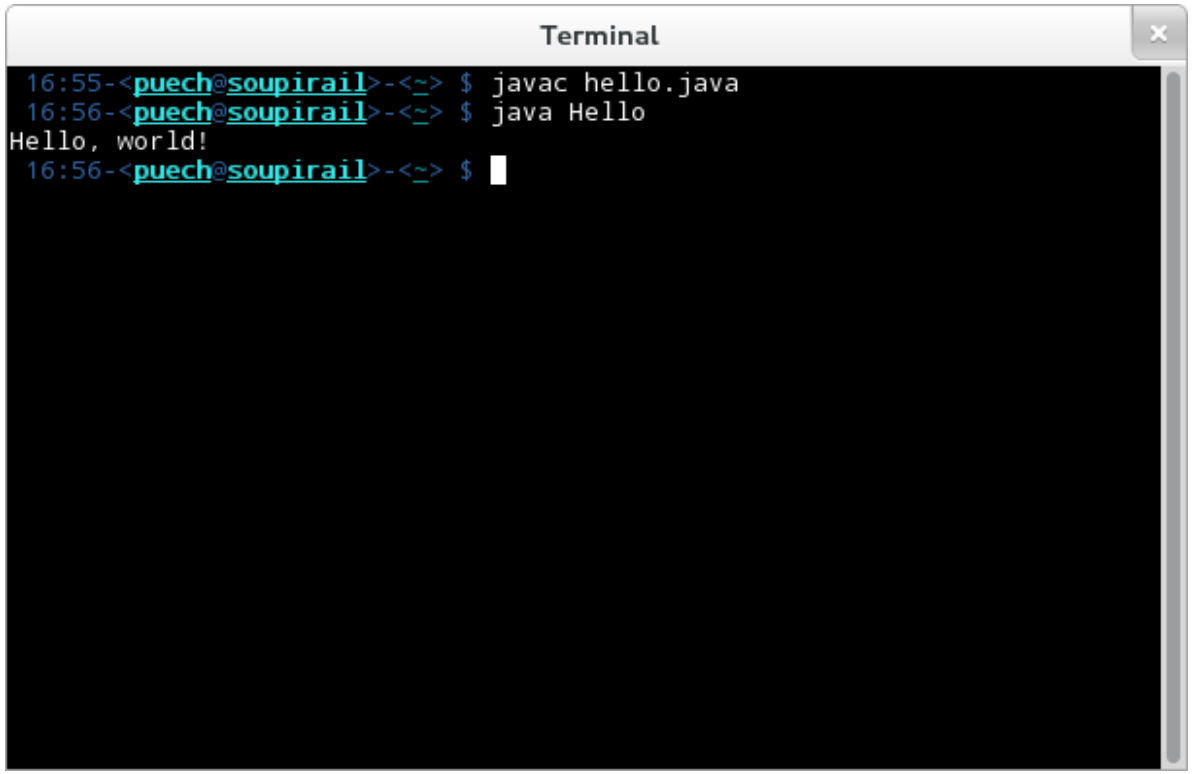
Qu’est-ce qu’une machine virtuelle?



Qu’est-ce qu’une machine virtuelle?



Qu’est-ce qu’une machine virtuelle?



Qu'est-ce qu'une machine virtuelle?

Définition (Machine virtuelle)

L'implémentation d'une machine comme un programme prenant un programme et émulant son execution.

Définition (Hôte)

Machine sur laquelle tourne la MV

Définition (Invité)

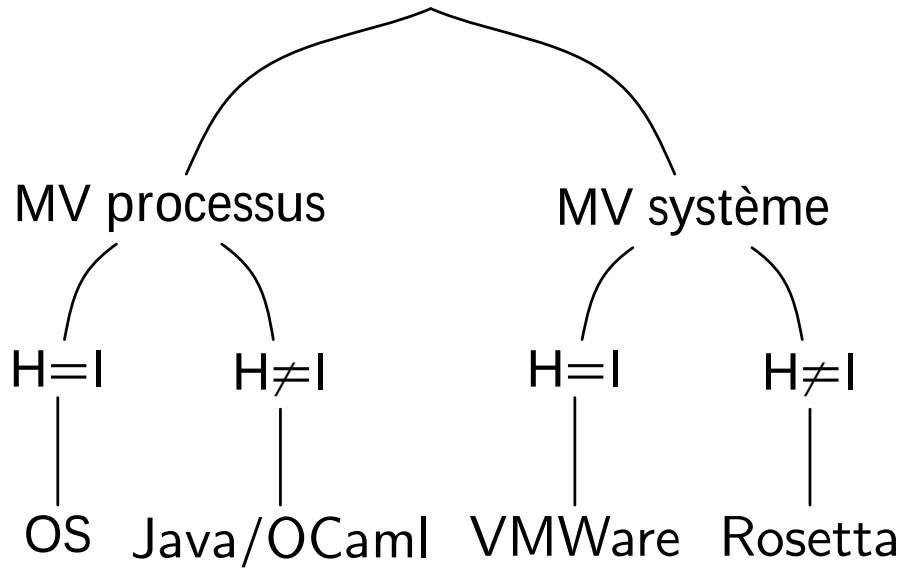
Machine émulée

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

Exemples

But	**Implémentations** (MV)	**Hôte** (H)	**Invité** (I)
Émulation d'architectures	Rosetta	x86	PPC
Machines virtuelles système	VMWare, VirtualBox	x86+OS	x86
Langages de haut niveau	Java, OCaml	x86	code-octet
«Multitâche»	systèmes d'exploitation	x86	x86
Dans le processeur	microprogramme	circuit	x86

Taxonomie


MV processus
MV système
H=I
H≠I
H=I
H≠I
OS
Java/OCaml
VMWare
Rosetta

Une définition «mathématique»

Définition (Machine)

La donnée d'un ensemble d'états S (e.g. état mémoire et registres d'un processeur), et une fonction

$$exec : S \to S$$

de transition d'un état vers le suivant (e.g. execution d'une instruction)

Définition (Machine virtuelle)

La donnée de deux machines $\langle S, exec \rangle$ et $\langle S', exec' \rangle$ (resp. invité et hôte) et une fonction

$$virt : S \to S'$$

de virtualisation, associant à chaque état de l'invité un état de l'hôte

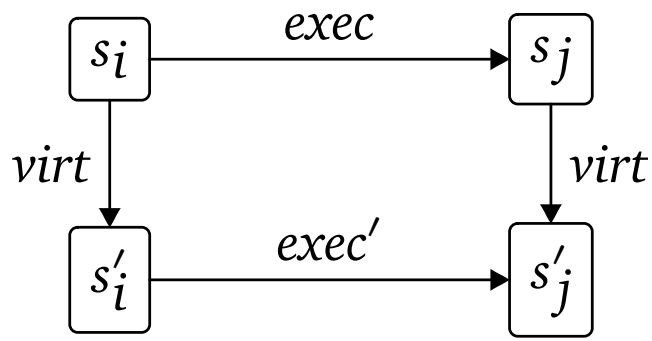
Une définition «mathématique»

Propriété (Correction)

Soit $s \in S$ un état de l'invité. Alors:

$$exec'(virt(s)) = virt(exec(s))$$

autrement dit, que ce diagramme commute:



Le programme, une donnée comme une autre

Le *modèle de Von Neumann*: les instructions sont stockées en mémoire, à côté des données. Le programme *est* une donnée.
($\neq$ du *modèle de Harvard* où données et instructions sont dans des mémoires séparées, les instructions ne pouvant être que executées).

Conséquences capitales[1]:

- une telle machine a plus d'une utilité
- on peut charger un programme depuis un support physique
- télécharger un programme ou une mise à jour puis l'executer
- un programme peut générer ou modifier un autre programme (compilateur)
- analyser son code (antivirus, analyses statiques)

[1] Comparer à: *(i)* une calculette *(ii)* l'ENIAC *(iii)* une machine à expressions régulières

La machine virtuelle, un programme comme un autre

...

- lire et interpréter les instructions de son code

Dématérialiser la machine a de nombreuses conséquences:

choix du jeu d'instructions on n'est plus lié au jeu d'instructions du processeur: émulation, code-octet ...

contrôle de l'execution la MV peut observer le programme avant de l'évaluer, sauver et restaurer son état: débogueur, virtualisation, «sandboxing»

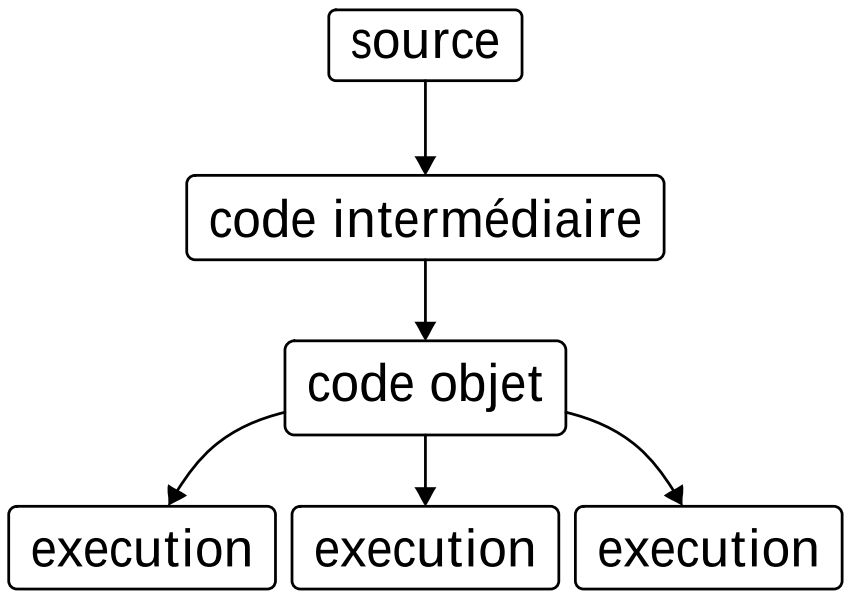
raisonnement sur les programmes on peut s'abstraire des détails de l'électronique: un cadre formel et universel pour comprendre, i.e. prouver des propriétés sur l'évaluation

la MV comme donnée comme tout programme, la MV elle-même peut être téléchargée, mise à jour...

L'avantage majeur: des programmes portables

Sans machine virtuelle

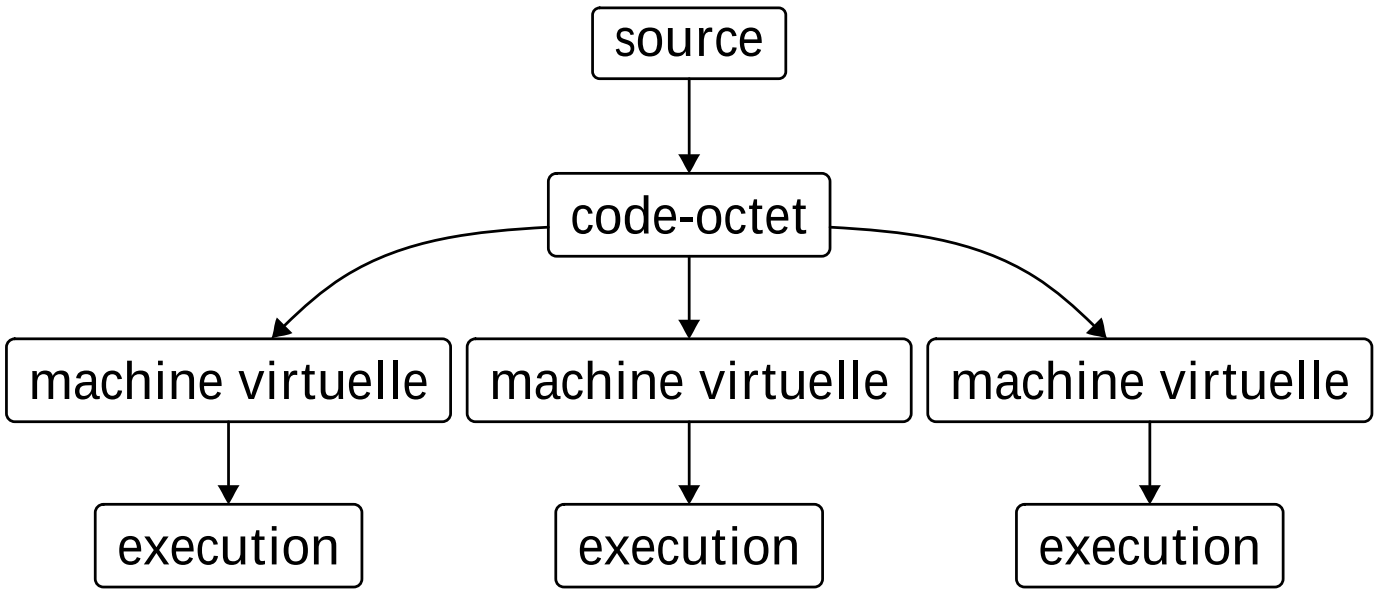
- un compilateur classique génère du *code objet* ou *natif* pour une architecture physique donnée (x86, PPC, MIPS...)
- n architectures prises en charge n executables à distribuer.



L'avantage majeur: des programmes portables

Avec machine virtuelle

- Ocamlc/Javac génèrent du *code-octet* pour une MV (Ocamlrun/Java), qui l'interprète ou le traduit en code natif
- Un seul executable distribué, *n portages* de la MV



Interprétation et compilation

La machine virtuelle comme un compromis entre interprétation et compilation:

Interprétation L'exécution se fait au fur et à mesure de l'analyse/sans pré-traitement du code source

Compilation Traduction du code source d'un programme en «langage machine» (instructions processeur)

Schéma avec machine virtuelle Compilation du code source en un «langage machine» (de plus haut niveau), puis interprétation par une MV

Remarque

La distinction n'est pas si nette:

- *même les interprètes travaillent sur une forme pré-traitée du code source (arbre de syntaxe abstraite, voir* AC6*)*
- *les «langages interprétés» (*Python, Javascript*) sont souvent à MV*
- *les instructions processeur sont compilées en un langage de plus bas niveau*

Comment implémenter une machine virtuelle?

- Le code source est *compilé* vers du code-octet
  non-divulgation du source, optimisations possibles
- Le code-octet est *interprété* par la MV potentiels problèmes d'efficacité
- Il est une donnée, et peut être analysé pour garantir des propriétés sur l'execution
- L'implémentation de MVs simples, efficaces et sûres est difficile

C'est l'objectif de ce cours

Dans ce cours

On apprendra à concevoir et implémenter des machines virtuelles:

- coder/décoder des instructions en code-octet (assembler/désassembler)
- comprendre les *machines à pile*
- savoir compiler des expressions vers du code-octet
- traiter les appels de fonctions et de méthodes

Deux études de cas:

- OCamlrun, La machine virtuelle de OCaml
- JVM, la machine virtuelle de Java

Dans ce cours

On apprendra à concevoir et implémenter des machines virtuelles:

- coder/décoder des instructions en code-octet (assembler/désassembler)
- comprendre les *machines à pile*
- savoir compiler des expressions vers du code-octet
- traiter les appels de fonctions et de méthodes

Deux études de cas:

- OCamlrun, La machine virtuelle de OCaml
- JVM, la machine virtuelle de Java

Spoiler alert L'étude des machines virtuelles n'est qu'une excuse pour introduire la compilation dans un cadre simple

Insertion dans le cursus de Paris 7

PF1	Notion de machine binaire, de code
PF5	Premiers pas avec OCaml
AC6	Front-end d'un compilateur, interprétation
MV6	Vous êtes ici
Compil-M1	Back-end d'un compilateur
...	

Prérequis

- Notions d'assembleur et d'architecture des machines
- Codage et décodage de/vers format binaire
- Manipulation d'expressions en OCaml

Informations pratiques

Horaire	jeudi de 14h à 16h
Cours et TD	alternés chaque semaine, assurés par *Matthias Puech* puech@pps.univ-paris-diderot.fr
Projet	implémenter une machine virtuelle directement lié au projet d'AC6
Évaluation	Session 1 : 1/3 projet + 2/3 examen Session 2 : max(examen, 1/3 projet + 2/3 examen)
Page du cours	http://www.pps.univ-paris-diderot.fr/~puech/ens/mv6.html
Liste de diffusion	important: s'abonner à mv6-forum@listes.sc.univ-paris-diderot.fr

Bibliographie

Cours atypique, pas de livre «tout-en-un»

- *Virtual Machines: Versatile Platforms for Systems and Processes* (Smith, Nair, 2005)
- *Java and the Java Virtual Machine: Definition, Verification, Validation* (Stärk, Schmid, Börger, 2001)
- *The Java Virtual Machine* (Meyer, Downing, Shulmann, 1997)
- *Développement d'applications avec Objective Caml* (Chailloux, Manoury, Pagano, 2000)[2]
- *Caml Virtual Machine – File and data format* (Clerc, 2007)[3]
- *Caml Virtual Machine — Instruction set* (Clerc, 2010)[4]

[2] http://www.pps.univ-paris-diderot.fr/Livres/ora/DA-OCAML/
[3] http://cadmium.x9c.fr/distrib/caml-formats.pdf
[4] http://cadmium.x9c.fr/distrib/caml-instructions.pdf

Plan du cours

Plan du cours

Qu'est-ce qu'une expression?

- Par commodité, les programmes sont souvent représentés comme/composés d' expressions.
- Une expression est la représentation textuelle d'une structure arborescente.
- L'*analyse syntaxique* convertit la représentation textuelle en l'*arbre de syntaxe* (donnée)
- Une expression est *évaluée* en une *valeur*

Exemple

La chaîne de caractères

```
let x = 10. /. 2. ** 2. in x +. 1.
```

est une représentation textuelle de l'expression qui a pour valeur 3.5

Qu'est-ce que du code natif?

- Le code natif est une *liste contigue d'instructions* pour le processeur, stockée en RAM
- Chaque instruction est un *code binaire* qui modifie l'état de la mémoire (RAM + registres)
- L'*assembleur* est une syntaxe compréhensible pour ce code
- Un registre spécial, PC (program count) stocke l'adresse de la prochaine instruction à executer
- Le processeur implémente le cycle *fetch-decode-execute*:

  fetch charge le code de la prochaine instruction, incrémente PC

  decode décode l'instruction chargée (e.g. `0x43AB` → "ajouter 2 au contenu du registre 3")

  execute réalise l'opération décodée

- Le jeu d'instruction et leur codage dépend du processeur et constitue l'*ISA* (*instruction set architecture*)

Qu’est-ce que du code natif?

```
$ objdump -d /bin/ls
/bin/ls:     file format elf64-x86-64
Disassembly of section .init:
0000000000402148 <_init>:
402148: 48 83 ec 08    sub    $0x8,%rsp
40214c: e8 af 25 00 00 callq  404700 <__ctype_b_loc@plt+0x1f40>
402151: e8 3a 26 00 00 callq  404790 <__ctype_b_loc@plt+0x1fd0>
402156: e8 65 04 01 00 callq  4125c0 <__ctype_b_loc@plt+0xfe00>
40215b: 48 83 c4 08    add    $0x8,%rsp
40215f: c3             retq
Disassembly of section .plt:
0000000000402160 <__ctype_toupper_loc@plt-0x10>:
402160: ff 35 aa 80 21 00 pushq  0x2180aa(%rip)
402166: ff 25 ac 80 21 00 jmpq   *0x2180ac(%rip)
40216c: 0f 1f 40 00       nopl   0x0(%rax)
0000000000402170 <__ctype_toupper_loc@plt>:
402170: ff 25 aa 80 21 00 jmpq   *0x2180aa(%rip)
402176: 68 00 00 00 00    pushq  $0x0
40217b: e9 e0 ff ff ff    jmpq   402160 <_init+0x18>
0000000000402180 <getenv@plt>:
402180: ff 25 a2 80 21 00 jmpq   *0x2180a2(%rip)
402186: 68 01 00 00 00    pushq  $0x1
40218b: e9 d0 ff ff ff    jmpq   402160 <_init+0x18>
...
```

Le code-octet: un code binaire portable

Le code-octet des MV est une approximation du code natif. Sa proximité avec le code natif réduit le travail de la MV (la couche d'interprétation):

- c'est une liste d'instructions
- elles codent des opération sur la mémoire
- le jeu d'instruction constitue la MV
- elles sont souvent moins nombreuses mais de plus haut niveau que sur un processeur
- le modèle de mémoire peut être de plus haut niveau (pile)

Interpréter du code-octet est plus efficace qu'interpréter le code source directement. La MV peut même *compiler* le code-octet en code machine à la volée (compilation *just-in-time*)

Le code-octet: un code binaire portable

Le code-octet des MV est une approximation du code natif. Sa proximité avec le code natif réduit le travail de la MV (la couche d'interprétation):

- c'est une liste d'instructions
- elles codent des opération sur la mémoire
- le jeu d'instruction constitue la MV
- elles sont souvent moins nombreuses mais de plus haut niveau que sur un processeur
- le modèle de mémoire peut être de plus haut niveau (pile)

Interpréter du code-octet est plus efficace qu'interpréter le code source directement. La MV peut même *compiler* le code-octet en code machine à la volée (compilation *just-in-time*)

Comment aller de l'expression au code-octet? Comment interpréter le code-octet?

Exemple: le langage Myrte

On considère le langage des expressions formées par:

- les constantes **true**, **false**, 0, 1, 2...
- les opérations binaires $+$, $=$, $\wedge$
- les parenthèses

et sa sémantique habituelle:

Exemple

- $2 + 2$ *vaut* 4
- $(1 = 0 + 1) \wedge (1 = 1) \wedge$ **true** *vaut* **true**
- **true** $+ 1$ *ne vaut rien (erreur)*

Exemple: le langage Myrte

On considère le langage des expressions formées par:

- les constantes **true**, **false**, 0, 1, 2...
- les opérations binaires $+$, $=$, $\wedge$
- les parenthèses

et sa sémantique habituelle:

Exemple

- $2 + 2$ *vaut* 4
- $(1 = 0 + 1) \wedge (1 = 1) \wedge$ **true** *vaut* **true**
- **true** $+ 1$ *ne vaut rien (erreur)*

Exercice

Écrire un interprète pour ce langage

Un interprète de Myrte

On définit d'abord le type des valeurs et des expressions (AST):

```
type value =
  | Int of int
  | Bool of bool

type binop = Add | Eq | And
type expr =
  | Const of value
  | Binop of binop × expr × expr
```

Un interprète de Myrte

On définit d'abord le type des valeurs et des expressions (AST):

```
type value =
  | Int of int
  | Bool of bool

type binop = Add | Eq | And
type expr =
  | Const of value
  | Binop of binop × expr × expr


let ex1 = let deux = Const (Int 2) in
  Binop (Eq, Binop (Add, deux, deux), Const (Int 4))
let ex2 = Binop (Eq, Const (Int 2), Const (Bool true))
```

Un interprète de Myrte

Puis le code de l'interprète:

```
let rec interp : expr → value = function
  | Const v  → v
  | Binop (b, e1, e2) → match b, interp e1, interp e2 with
    | Add, Int i, Int j → Int (i + j)
    | Eq, Int i, Int j → Bool (i = j)
    | And, Bool i, Bool j → Bool (i && j)
    | _ → failwith "ill-formed expression"
```

Un interprète de Myrte

Puis le code de l'interprète:

```
let rec interp : expr → value = function
  | Const v  → v
  | Binop (b, e1, e2) → match b, interp e1, interp e2 with
    | Add, Int i, Int j → Int (i + j)
    | Eq, Int i, Int j → Bool (i = j)
    | And, Bool i, Bool j → Bool (i && j)
    | _ → failwith "ill-formed expression"
```

```
# interp ex1;;
- : value = Bool true
# interp ex2;;
Exception: Failure "ill-formed expression".
```

Typage de Myrte

Problème

L'évaluation de certaines expressions peut échouer (e.g. $2 + \mathbf{true}$)

Typage de Myrte

Problème

L'évaluation de certaines expressions peut échouer (e.g. $2 + \mathbf{true}$)

Question

Peut-on analyser une expression pour détecter les cas d'échec *sans évaluer l'expression* (*statiquement*, i.e. sans calculer sa valeur)?

Typage de Myrte

Problème

L'évaluation de certaines expressions peut échouer (e.g. $2 + \textbf{true}$)

Question

Peut-on analyser une expression pour détecter les cas d'échec *sans évaluer l'expression* (*statiquement*, i.e. sans calculer sa valeur)?

Exercice

Écrire une fonction de typage `check` : `expr` $\rightarrow$ `bool` telle que si `check e` $=$ **true** alors `interp e` $=$ `v`

> *"Well-typed programs can't go wrong"* —*Milner (1978)*

Typage de Myrte

```
type tp = TInt | TBool

let rec infer : expr → tp = function
  | Const (Int _) → TInt
  | Const (Bool _) → TBool
  | Binop (b, e1, e2) → match b, infer e1, infer e2 with
    | Add, TInt, TInt → TInt
    | Eq, TInt, TInt → TBool
    | And, TBool, TBool → TBool
    | _ → failwith "expression mal typee"

let check e = try ignore (infer e); true with Failure _ → false
```

De l’interprète au code

- Sur un exemple comme celui-ci, un interprète fait l’affaire
- Pour un langage plus riche, il devient beaucoup trop lent
- Il faut “linéariser” l’expression en code, et au besoin l’optimiser

Machines à *a-pile*

La mémoire stocke des *mots mémoire* (64 bits). Un *état* de la machine est constitué de:

- une *pile* S
- un registre A (l'*accumulateur*)
- un tableau d'*instructions* C, et un pointeur vers l'instruction courante

Jeu d'instructions constituant C:

push	empile le contenu de A sur S
consti n	remplace le contenu de A par n
addi	dépile un mot n de S, remplace A par $A+n$
andi	dépile un mot n de S, remplace A par 0 si $A=n=0$, par une valeur quelconque $\neq 0$ sinon
eqi	dépile un mot n de S, remplace A par 1 si $n=A$, 0 sinon

Une machine virtelle pour Myrte

Exercice

Écrire un interprète pour ce langage, i.e. une fonction
`machine` : `state` → `state`

Une machine virtelle pour Myrte

Exercice

Écrire un interprète pour ce langage, i.e. une fonction
`machine` : `state` → `state`

Coup de pouce

```
type instr = Push | Consti of int | Addi | Eqi | Andi

type state = {
  mutable acc: int;
  code: instr array;
  mutable pc: int;   (* indice de l'instruction courante dans code *)
  stack: int array;
  mutable sp: int;   (* indice du sommet de la pile dans stack *)
}
```

Une machine virtelle pour Myrte

```
let machine m = while m.pc < Array.length m.code do
  begin match m.code.(m.pc) with
  | Consti n →
      m.acc ← n
  | Push →
      m.sp ← m.sp + 1;
      m.stack.(m.sp) ← m.acc
  | Addi →
      m.acc ← m.stack.(m.sp) + m.acc;
      m.sp ← m.sp-1
  | Andi →
      m.acc ← m.stack.(m.sp) × m.acc;
      m.sp ← m.sp-1
  | Eqi →
      m.acc ← if m.stack.(m.sp) = m.acc then 1 else 0;
      m.sp ← m.sp-1 end;
   m.pc ← m.pc + 1
 done; m
```

Une machine virtelle pour Myrte

Exemple

```
let init c =
  { code = c; stack = Array.make 1000 42;
    pc = 0; sp = -1; acc = 52 }

# machine (init [|Consti 2; Push|]);;
# machine (init [|Consti 2; Push; Consti 2; Addi|]);;
# machine (init [|Consti 2; Push; Addi|];;
# machine (init [|Push; Addi|]);;
# machine (init [|Addi|]);;
# machine (init (Array.make 1001 Push));;
```

Compilation d'expressions

On doit d'abord fixer des conventions d'encodage:

Étape 1: Encodage des valeurs en états

- Quand la machine s'arrête, le résultat est dans A
- Un entier n est codé par sa représentation sur un mot $\overline{n}$ (complément par 2)
- Les booléens **true** et **false** sont codés resp. par $\overline{0}$ et $\overline{n} \neq \overline{0}$

Compilation d'expressions

On doit d'abord fixer des conventions d'encodage:

Étape 1: Encodage des valeurs en états

- Quand la machine s'arrête, le résultat est dans A
- Un entier n est codé par sa représentation sur un mot $\overline{n}$ (complément par 2)
- Les booléens **true** et **false** sont codés resp. par $\overline{0}$ et $\overline{n} \neq \overline{0}$

Remarque

L'encodage est non injectif: $|\textbf{false}| = |0| = \overline{0}$.

Compilation d'expressions

```
let repr : value → int = function
  | Bool true → 1
  | Bool false → 0
  | Int i → i

let eval c =
  let s = machine
    { code = Array.of_list c;
      stack = Array.make 1000 42;
      pc = 0;
      sp = -1;
      acc = 52 } in
  s.acc
```

Compilation d'expressions Myrte

Étape 2: Compilation des expressions en instructions

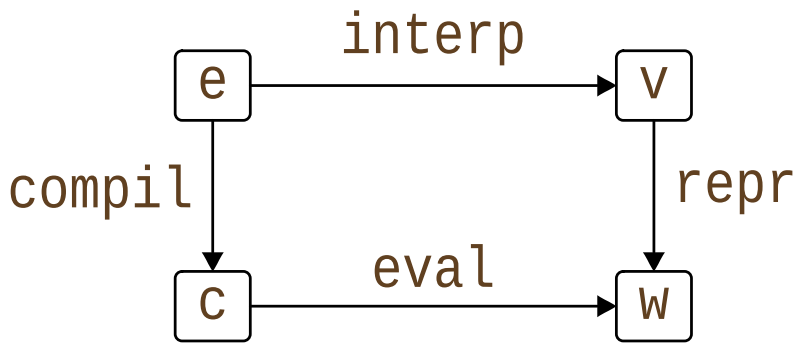
Écrire une fonction

```
val compil : expr → instr list
```

telle que pour toute expression `e`,

```
eval (compil e) = repr (interp e)
```

i.e. telle que ce diagramme commute:



Compilation d'expressions Myrte

Indice

« La notation polonaise inversée d'une expression est le code pour la machine à pile le calculant »

Compilation d'expressions Myrte

```
let op = function Add → Addi | And → Andi | Eq → Eqi

(* compil met la repr. de la valeur dans l'accum. et restaure la pile *)
let rec compil : expr → instr list = function
  | Const v → [Consti (repr v)]
  | Binop (o, e1, e2) →
      compil e1 @ [Push] @
      compil e2 @ [op o]
```

Compilation d'expressions Myrte

Compilation d'expressions Myrte

Parenthèse

Cette version est *beaucoup* plus efficace:

```
(* version par passage d'accumulateur: liste renvoyee renversee *)
let rec compil l : expr → instr list = function
  | Const v → Consti (repr v) :: l
  | Binop (o, e1, e2) →
      let l = Push :: compil l e1 in
      op o :: compil l e2

let compil e = List.rev (compil [] e)
```

Optimisation

La fonction de compilation peut être plus maline et optimiser le code généré.

Factorisation de sous-expressions communes

Une opération dont les deux sous-expressions sont syntaxiquement égales $e + e$ peut être factorisée et e calculée une seule fois

Exemple

L'expression $(1 + 2) + (1 + 2)$ *est compilée en*

```
[|Consti 1; Push; Consti 2; Addi; Push; Consti 1; Push; Consti 2; Addi; Addi|]
```

Elle pourrait être compilée simplement en

```
[|Consti 1; Push; Consti 2; Addi; Push; Addi|]
```

Optimisation

La fonction de compilation peut être plus maline et optimiser le code généré.

Factorisation de sous-expressions communes

Une opération dont les deux sous-expressions sont syntaxiquement égales $e + e$ peut être factorisée et e calculée une seule fois

Exemple

L'expression $(1+2)+(1+2)$ *est compilée en*

```
[|Consti 1; Push; Consti 2; Addi; Push; Consti 1; Push; Consti 2; Addi; Addi|]
```

Elle pourrait être compilée simplement en

```
[|Consti 1; Push; Consti 2; Addi; Push; Addi|]
```

Exercice

- Implémenter cette optimisation dans la fonction `compil`
- Sauriez-vous la généraliser à toutes réutilisation de sous-expressions (e.g. dans $((1+2)+3)+(1+2)$)?

Code-octet

Pour l'instant, un *programme* est une *valeur* OCaml,
(e.g. [|Consti 2; Push; Addi|])

Or un programme est la représentation efficace du calcul d'une expression Myrte. C'est lui qu'on veut *stocker* et *distribuer*

Donc il faut un *format de fichier* pour ces programmes:
il faut un code-octet.

Code-octet

Propositions

1. La représentation ASCII de la valeur OCaml

Code-octet

Propositions

1. La représentation ASCII de la valeur OCaml
   pas compacte
2. La représentation ASCII *gzippée* de la valeur OCaml

Code-octet

Propositions

1. La représentation ASCII de la valeur OCaml
   pas compacte
2. La représentation ASCII *gzippée* de la valeur OCaml
   pas efficace
3. La représentation *mémoire* de la valeur OCaml
   (*cf.* Marshal.`to_string`)

Code-octet

Propositions

1. La représentation ASCII de la valeur OCaml
   pas compacte
2. La représentation ASCII *gzippée* de la valeur OCaml
   pas efficace
3. La représentation *mémoire* de la valeur OCaml
   (*cf.* Marshal.`to_string`)
   pas compacte non plus
4. Une représentation ad-hoc

Code-octet

Propositions

1. La représentation ASCII de la valeur OCaml
   pas compacte
2. La représentation ASCII *gzippée* de la valeur OCaml
   pas efficace
3. La représentation *mémoire* de la valeur OCaml
   (*cf.* Marshal.`to_string`)
   pas compacte non plus
4. Une représentation ad-hoc ∅

Un code-octet pour Myrte

Codage des instructions

- une instruction par octet
- pour chaque octet:

  bit 0-2 *opcode* de l'instruction

  bit 3-7 vide, sauf si l'instruction est Consti n, auquel cas c'est n
- les opcodes sont: *(i)* Push: 0 *(ii)* Addi: 1 *(iii)* Eqi: 2 *(iv)* Andi: 3 *(v)* Consti: 4

Un code-octet pour Myrte

Codage des instructions

- une instruction par octet
- pour chaque octet:

  bit 0-2 *opcode* de l'instruction

  bit 3-7 vide, sauf si l'instruction est Consti n, auquel cas c'est n
- les opcodes sont: *(i)* Push: 0 *(ii)* Addi: 1 *(iii)* Eqi: 2 *(iv)* Andi: 3 *(v)* Consti: 4

Limitation

On ne peut coder que les constantes entre 0 et 31 (!)

Exemple

Le programme [|Consti *2;* Push*;* Addi|] *a pour code-octet*

$\overline{20}$	$\overline{0}$	$\overline{1}$

Notion d'assembleur

Définition

L'assembleur `assemble` *est la fonction qui encode un programme en code-octet*

Le désassembleur `disassemble` *est la fonction qui décode du code-octet en programme*

Remarque

Par abus de langage, on appelle souvent assembleur la syntaxe prise en entrée par l'assembleur.

Propriété

Ces deux fonctions forment une bijection:

$$\texttt{disassemble}\ (\texttt{assemble}\ \texttt{p}) = \texttt{p}$$

Un (dés)assembleur pour Myrte

```
let assemble (p : instr array) : string =
  let s = String.make (Array.length p) 'z' in
  for i = 0 to Array.length p - 1 do
    s.[i] ← match p.(i) with
    | Push → Char.chr 0
    | Addi → Char.chr 1
    | Eqi → Char.chr 2
    | Andi → Char.chr 3
    | Consti n → assert (n < 32); Char.chr (4 + n lsl 3);
  done; s
;;
```

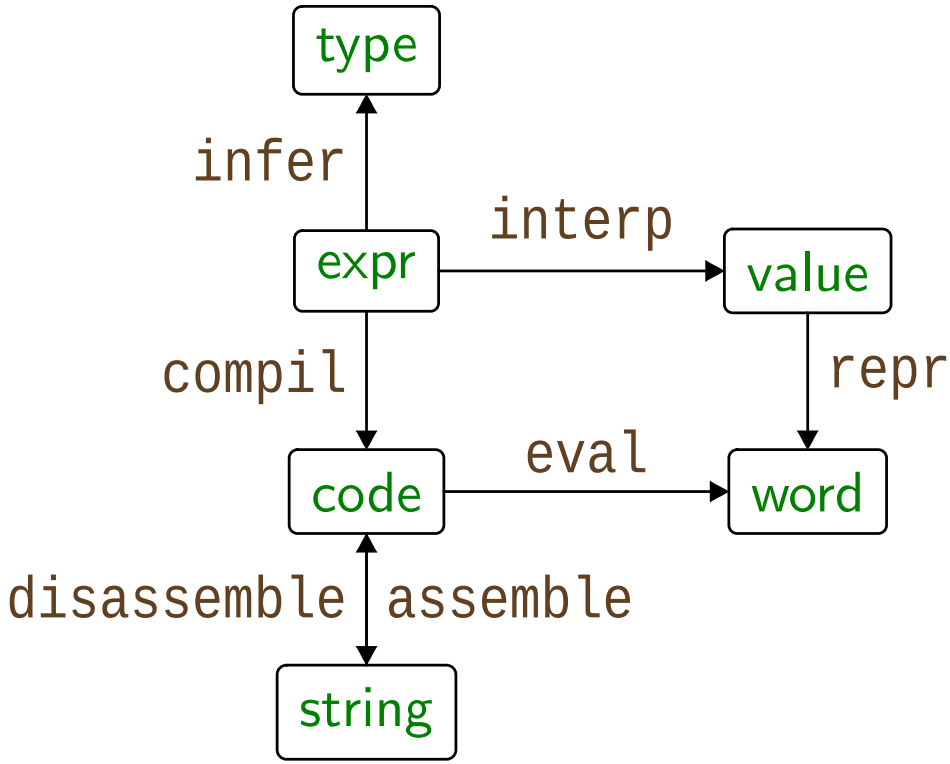
Un (dés)assembleur pour Myrte

```
let disassemble (s : string) : instr array =
  let p = Array.make (String.length s) Push in
  for i = 0 to String.length s - 1 do
    p.(i) ← match Char.code s.[i] with
    | 0 → Push
    | 1 → Addi
    | 2 → Eqi
    | 3 → Andi
    | n when (n mod 8 = 4) → Consti (n lsr 3)
    | _ → failwith "invalid byte-code"
  done; p
```

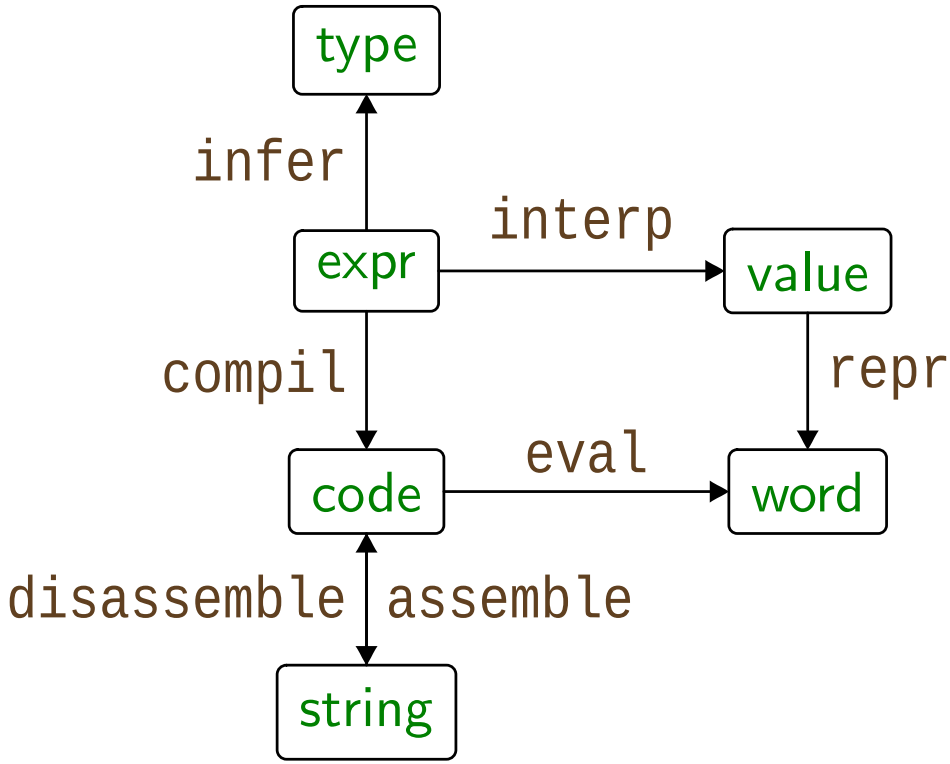
Un (dés)assembleur pour Myrte

```
# let p = Array.of_list (compil ex1);;
val p : instr array =
  [|Consti 2; Push; Consti 2; Addi; Push; Consti 4; Eqi|]

# let s = assemble p;;
val s : string = "\020\000\020\001\000$\002"

# let p' = disassemble s;;
val p' : instr array =
  [|Consti 2; Push; Consti 2; Addi; Push; Consti 4; Eqi|]
```

Résumé


type
infer
interp
expr
value
compil
repr
eval
code
word
disassemble
assemble
string

Résumé



Exercice

Décomposer les étapes de la compilation d'un programme C avec `cpp`, `gcc -S`, `as` et `ld`. Faire la même chose sur un programme OCaml avec `ocamlopt -S`.

Alternatives

Pour le besoin de ce cours, on a fait des choix arbitraires

Machine à a-pile

Il existe d'autre modèles de calcul/machines abstraites

- machine à pile (e.g. Postscript)
- machines à registres (RAM, RASP, pointeurs)
- machines de Turing
- systèmes de réecriture de termes

Alternatives

Pour le besoin de ce cours, on a fait des choix arbitraires

Machine à a-pile

Il existe d'autre modèles de calcul/machines abstraites

- machine à pile (e.g. Postscript)
- machines à registres (RAM, RASP, pointeurs)
- machines de Turing
- systèmes de réecriture de termes

Interprétation du code-octet

La VM peut aussi le compiler en code natif
(compilation *just-in-time*)

Alternatives

Pour le besoin de ce cours, on a fait des choix arbitraires

Machine à a-pile

Il existe d'autre modèles de calcul/machines abstraites

- machine à pile (e.g. Postscript)
- machines à registres (RAM, RASP, pointeurs)
- machines de Turing
- systèmes de réecriture de termes

Interprétation du code-octet

La VM peut aussi le compiler en code natif
(compilation *just-in-time*)

Conception du code-octet

Le codage peut aussi être de taille variable, avoir des instruction composées (e.g. Constpush n)

Exercice

On souhaite étendre le langage Myrte avec une construction conditionnelle **if** ... **then** ... **else** .... Sa sémantique est:

> *"la valeur de l'expression* **if** E_1 **then** E_2 **else** E_3 *est la valeur de* E_2 *si la valeur de* E_1 *est* **true**, *ou la valeur de* E_3 *si la valeur de* E_1 *est* **false**."

Exercice

On souhaite étendre le langage Myrte avec une construction conditionnelle **if** ... **then** ... **else** .... Sa sémantique est:

> *"la valeur de l'expression* **if** E_1 **then** E_2 **else** E_3 *est la valeur de* E_2 *si la valeur de* E_1 *est* **true**, *ou la valeur de* E_3 *si la valeur de* E_1 *est* **false**."

1. Étendre le type des expressions avec un constructeur If
2. Étendre et tester la fonction `interp` avec ce nouveau cas
3. Étendre la fonction de typage `check` (on considère que E_2 et E_3 s'évaluent toujours vers des entiers)
4. Ajouter l'instruction suivante à la MV:

   branchif n saute n instruction en avant si $A \neq 0$; n'a pas d'effet sinon
5. Étendre la fonction de compilation `compil` et de (dés)assemblage de façon à compiler la nouvelle construction

Plan du cours

Plan du cours

Présentation

Il existe (au moins) deux compilateurs pour le langage OCaml

`ocamlopt` compile vers du code natif
(IA-32, IA-64, AMD64, HP/PA, PowerPC, SPARC, Alpha, MIPS, ARM)

`ocamlc` compile vers du bytecode (`.cmo`)
(`ocamlrun`)

OCaml-Java compile vers du bytecode Java (`.class`)

OCamlIl compile vers du bytecode CIL (.NET)

Présentation

Il existe (au moins) deux compilateurs pour le langage OCaml

`ocamlopt` compile vers du code natif
(IA-32, IA-64, AMD64, HP/PA, PowerPC, SPARC, Alpha, MIPS, ARM)

`ocamlc` compile vers du bytecode (`.cmo`)
(`ocamlrun`)

OCaml-Java compile vers du bytecode Java (`.class`)

OCamlI compile vers du bytecode CIL (.NET)

La machine virtuelle `ocamlrun`

- machine à a-pile + environnement + tas
- représentation mémoire homogène des données
- les fonctions sont des données

Le format de code-octet `.cmo`

Observons le contenu d'un fichier `.cmo` avec `xxd`...

Le format de code-octet `.cmo`

Observons le contenu d'un fichier `.cmo` avec `xxd`...
On y retrouve:

- "`Caml1999`": un numéro magique
- "`Test`": le nom du module créé
- le reste est incompréhensible → il faut le désassembler

Le format de code-octet `.cmo`

- Il existe une spécification du format `.cmo`:
  `http://cadmium.x9c.fr/distrib/caml-formats.pdf`
- Il existe aussi des outils pour afficher leur contenu de façon lisible:
  - `ocamldumpobj` affiche la description textuelle des instructions
  - `ocamlobjinfo` affiche des informations supplémentaires
- On peut même demander au compilateur d'afficher le code produit:

```
$ ocamlc -c -dinstr test.ml
$ ocaml -dinstr
```

`ocamlrun`, la machine virtuelle d'OCaml

Elle est programmée en C, et implémente le même cycle *fetch-decode-execute* que notre MV Myrte

- voir fichier `byterun/interp.c` du code source OCaml.
- un executable produit avec `ocamlc` commence par `#!/usr/bin/ocaml`

`ocamlrun`, la machine virtuelle d'OCaml

Son état est constitué de:

- un programme (liste de bytecode)
- un registre PC (adresse de la prochaine instruction)
- un registre A (l'accumulateur)
- une pile S
- un tas H (heap) où placer les données allouées
- un environnement global E
- un registre $extra\_args$

Plan d'attaque

Décrivons le jeu d'instruction de `ocamlrun`

On va découper sa présentation en trois jeux d'instructions:

- le fragment arithmétique, commun à Myrte
  (valeurs *immédiates*: `int`, `char`, `bool`, `unit`)
- la représentation et la manipulation des données structurées
  ($\alpha \times \beta$, `list`, `option`, `float`, types définis par l'utilisateur)
- la manipulation des fonctions de première classe
  ($\alpha \to \beta$)

Plan du cours

Représentation mémoire des valeurs

Toute valeur OCaml est représentée par un *mot machine* (32/64 bits). Son premier bit détermine son utilisation:

- si c'est 1, il représente une valeur immédiate
  (codée sur 31/63 bits)
- si c'est 0, il représente un pointeur vers un *bloc*
  (données structurées, fonctions, *cf.* plus loin)

Représentation mémoire des valeurs

Toute valeur OCaml est représentée par un *mot machine* (32/64 bits). Son premier bit détermine son utilisation:

- si c'est 1, il représente une valeur immédiate (codée sur 31/63 bits)
- si c'est 0, il représente un pointeur vers un *bloc* (données structurées, fonctions, *cf.* plus loin)

Représentation des valeurs immédiates

par des mots 31/63 bits

int	entier signé
char	entier correpondant au code ASCII
bool	`false` est codé par `0`, `true` par `1`
unit	`()` est codé par `0`

Gestion de la pile

push empile A

pop n dépile n éléments du sommet de S (c'est tout)

acc n copie dans A la valeur du n-ème élément de S

Variantes:

acc0 ... acc7 instructions spécialisées pour $n \leq 7$

pushacc n même effet que "push; acc n"

pushacc0 ... pushacc7 *cf.* acc0...acc7

assign n déplace A dans la n-ème case de S

(il vaut "( )" après)

Arithmétique

constint n	met n dans A Variantes:
	const0 . . . const3 constint spécialisés pour $n \leq 3$
	pushconstint n même effet que “push; constint n”
	pushconstint0. . . 3 *cf.* “pushconstint n”
offsetint n	incrémente A de n
negint	A reçoit l’opposé de sa valeur
addint	A reçoit A + sommet de S (qui est dépilé)
subint, mulint, divint, modint, andint, orint, (u)ltint, leint, (n)eqint. . .	sont des opérateurs binaires qui fonctionnent de la même manière que “addint”

Aiguillage

Le saut se fait par *décalage* (offset)
(alternative: par *adresse*)

branch n saute n instructions en avant (ou en arrière si $n < 0$)

branchif n saute n instructions si $A = 1$

branchifnot n saute n instructions si $A = 0$

beq m n saute n instructions si $A = m$

bneq, b(u)ltint, bleint, bgtint, b(u)geint sont des sauts conditionnels binaires qui fonctionnent de la même façon que “beq”

Plan du cours

Rappel

En OCaml, on peut déclarer des types “maison”

```
type t = A | B of int | C | D of t ;;
A ;;
- : t = A
B 52 ;;
- : t = B 52
D (D (D (D C))) ;;
- : t = D (D (D (D C)))
```

Comme pour toute valeur, ces valeurs doivent avoir une représentation en mémoire

Rappel

En OCaml, on peut déclarer des types “maison”

```
type t = A | B of int | C | D of t ;;
A ;;
- : t = A
B 52 ;;
- : t = B 52
D (D (D (D C))) ;;
- : t = D (D (D (D C)))
```

Comme pour toute valeur, ces valeurs doivent avoir une représentation en mémoire

sur le *tas* (*heap*)

Qu'est-ce que le tas (déjà)?

À un nouveau processus est assigné par l'OS

- un espace mémoire constant pour stocker son code
- un espace pour ses données constantes (string etc.)
- un espace de pile extensible (mais monotone)
- à la demande, de l'espace utilisateur (`malloc`, `free`)

La zone mémoire pointée par le retour de `malloc` fait partie du *tas*. Le processus est chargé de la libérer après utilisation.

Qu'est-ce que le tas (déjà)?

À un nouveau processus est assigné par l'OS

- un espace mémoire constant pour stocker son code
- un espace pour ses données constantes (string etc.)
- un espace de pile extensible (mais monotone)
- à la demande, de l'espace utilisateur (`malloc`, `free`)

La zone mémoire pointée par le retour de `malloc` fait partie du *tas*. Le processus est chargé de la libérer après utilisation.

on mime cela dans la MV

Représentation mémoire des valeurs

Toute valeur OCaml est représentée par un *mot machine* (32/64 bits). Son premier bit détermine son utilisation:

- si c'est 1, il représente une valeur immédiate
  (codée sur 31/63 bits)
- si c'est 0, il représente un pointeur vers un *bloc*
  (données structurées, fonctions)

Les blocs sont alloués par la MV par `malloc`

Représentation mémoire des valeurs

Représentation des données structurées

par des pointeurs vers des *blocs*. Un bloc est une zone contiguë de $n+1$ mots composé de:

- **en-tête** (1 mot) composé de:
  - **tag** (8 bits) identifie le type de bloc
  - **color** (2 bits) *cf.* plus loin
  - **wosize** (22 bits) taille n de la zone de donnée
- **donnée** (n mots)

Quelques tags (*cf.* `byterun/mlvalue.h`)

- un `float`: 253
- une `string`: 252
- une fonction (fermeture): 247
- un constructeur non constant: $[0 \ldots 245]$

Représentation des valeurs types utilisateurs

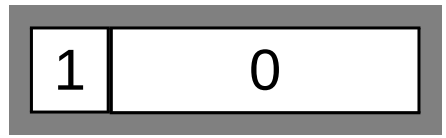
- Les constructeurs *constants* sont représentés par des entiers
  constant = sans argument (**of**); chaque constructeur est numéroté à partir de 0
- Les constructeurs *paramétrés* sont représentés par des blocs
  paramétré = avec argument

Représentation des valeurs types utilisateurs

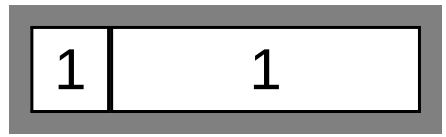
Exemple

```
type t = A | B of int | C | D of t ;;
```

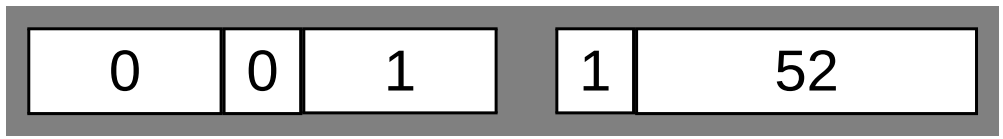
- A *la valeur immédiate 0:*



- C *la valeur immédiate 1:*

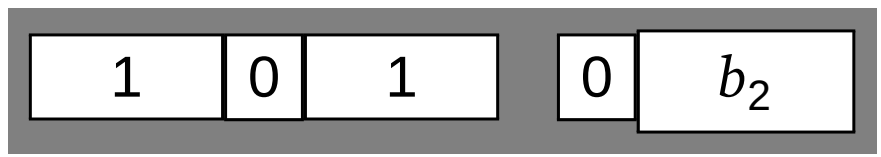


- B *52* *le bloc de taille 1 (2 mots):*



- D *(D C)* *deux blocs de taille 1:*

b_1 :



b_2 :

Représentation des valeurs des types de base

tableau `[|1;2;3;4|]`
un bloc de taille 4, de tag 0, dont chaque donnée est une valeur immédiate représentant un entier

n-uplet (1, 2, 3, 4)
même représentation!

chaîne `"foobar"`
un bloc de taille 2 (8/4), de tag 252: le tableau d'octet + '\0' + padding

liste `[1;2;3;4]`
représenté comme si c'était une valeur du type
type α `list` = Nil | Cons **of** $\alpha \times \alpha$ `list`
(*cf.* tableau)

Gestion des blocs

Instructions

makeblock n k alloue un bloc de taille n et de tag k, y copie les n éléments de A + sommet de S; A reçoit un pointeur vers ce bloc

getfield n A doit pointer vers un bloc; A reçoit la n-ème donnée du bloc

setfield n A doit pointer vers un bloc; la n-ème donnée du bloc reçoit le sommet de S

vectlength A reçoit la taille du bloc pointé par A

Gestion des blocs

Variantes

makeblock1,2,3, get/setfield0,1,2,3 *cf.* ce qui précède

getvectitem, setvectitem *cf.* get/setfield, mais avec tous les arguments dans A et S

atom k comme makeblock 0 k; il existe aussi atom0, pushatom k et pushatom0

makefloatblock, get/setfloatfield (cas particuler pour les `float`)

get/setstringchar (cas particuler pour les `string`)

offsetref n ajoute n au premier champ d'un bloc

Gestion des blocs

Switch

Permet de compiler le **match** de OCaml

switch $c_0 \ldots c_m$ $d_0 \ldots d_n$ On examine le contenu de A:

- si c'est une valeur directe i (le i-ème constructeur constant), on saute c_i instructions
- si c'est un pointeur vers un bloc de tag i (le i-ème constructeur *non* constant), on saute de d_i instructions

Le ramasse-miettes (*garbage collector* (GC))

Exemple

```
match [1;2;3] with
| x :: _ → x
```

- *crée sur le tas la liste* [*1;2;3*]
- *A est un pointeur vers cette liste*
- *parcourt ces blocs avec* “getfield”
- *A devient* 1

On n’a pas besoin de garder sur le tas les blocs de la liste.
Qui est en charge de les effacer?

Le ramasse-miettes (*garbage collector* (GC))

- Le ramasse-miettes parcourt le tas régulièrement à la recherche de blocs “orphelins”
  - soit qui ne sont plus référencé par aucun bloc
  - soit qui sont seulement référencé par des blocs eux même orphelins
- C’est la partie la plus importante du *runtime* OCaml
- L’algorithme doit savoir *à run time* si un mot est une valeur immédiate ou un pointeur: c’est la raison de l’annotation sur le premier bit des mots
- Il utilise un marquage des blocs selon leur ancienneté (color)

Plan du cours

L'essence de la programmation fonctionnelle

Les fonctions sont des valeurs comme les autres (de *première classe*), que l'on peut introduire et éliminer. Chaque genre de valeur est identifié par une construction de type.

Type	**Introduction**	**Elimination**
$\alpha \times \beta$	E_1, E_2	**let** (x, y) = E_1 **in** E_2
α `list`	:: ou []	**match** E **with** ...
type déclaré	constructeurs	pattern-matching
$\alpha \to \beta$	**fun** x $\to$ E	E_1 E_2

L'essence de la programmation fonctionnelle

Les fonctions sont des valeurs comme les autres (de *première classe*), que l'on peut introduire et éliminer. Chaque genre de valeur est identifié par une construction de type.

Type	**Introduction**	**Elimination**
$\alpha \times \beta$	E_1, E_2	**let** (x, y) = E_1 **in** E_2
α `list`	:: ou []	**match** E **with** ...
type déclaré	constructeurs	pattern-matching
$\alpha \to \beta$	**fun** x $\to$ E	E_1 E_2

Particularités de cette vision:

- Les fonctions sont toujours anonymes
- Elles n'ont qu'un argument
- Elles peuvent intervenir n'importe où dans le code

L'essence de la programmation fonctionnelle

Comment déclare-t-on des fonctions n-aires?

L'essence de la programmation fonctionnelle

Option 1

L'unique argument peut être un n-uplet:

Exemple

Tous ces programmes sont équivalents:

- **let** f *(*x, y*)* = *(*x × x*)* + y **in** f *(2, 1)*

L'essence de la programmation fonctionnelle

Option 1

L'unique argument peut être un n-uplet:

Exemple

Tous ces programmes sont équivalents:

- **let** f *(*x*,* y*)* = *(*x × x*)* + y **in** f *(2, 1)*
- **let** f p = **let** *(*x*,*y*)* = p **in** *(*x × x*)* + y **in** f *(2, 1)*

L’essence de la programmation fonctionnelle

Option 1

L’unique argument peut être un n-uplet:

Option 1

L'unique argument peut être un n-uplet:

Exemple

Tous ces programmes sont équivalents:

- **let** f *(*x, y*)* = *(*x × x*)* + y **in** f *(2, 1)*
- **let** f p = **let** *(*x,y*)* = p **in** *(*x × x*)* + y **in** f *(2, 1)*
- **let** f = *(***fun** *(*x, y*)* → *(*x × x*)* + y*)* **in** f *(2, 1)*
- *(***fun** *(*x, y*)* → *(*x × x*)* + y*)* *(2, 1)*

L’essence de la programmation fonctionnelle

Option 2

Par *curryfication*, c’est à dire en créant des fonctions qui renvoie des fonctions. OCaml fournit du sucre syntaxique pour cela:

L'essence de la programmation fonctionnelle

Option 2

Par *curryfication*, c'est à dire en créant des fonctions qui renvoie des fonctions. OCaml fournit du sucre syntaxique pour cela:

Exemple

Tous ces programmes sont équivalents:

- **let** f x y = (x × x) + y **in** f *2 1*
- **let** f = (**fun** x y → (x × x) + y) **in** f *2 1*

L'essence de la programmation fonctionnelle

Option 2

Par *curryfication*, c'est à dire en créant des fonctions qui renvoie des fonctions. OCaml fournit du sucre syntaxique pour cela:

Exemple

Tous ces programmes sont équivalents:

- **let** f x y = (x × x) + y **in** f *2 1*
- **let** f = (**fun** x y → (x × x) + y) **in** f *2 1*
- **let** f = (**fun** x → **fun** y → (x × x) + y) **in** f *2 1*

L'essence de la programmation fonctionnelle

Option 2

Par *curryfication*, c'est à dire en créant des fonctions qui renvoie des fonctions. OCaml fournit du sucre syntaxique pour cela:

Exemple

Tous ces programmes sont équivalents:

- **let** f x y = (x × x) + y **in** f *2 1*
- **let** f = (**fun** x y → (x × x) + y) **in** f *2 1*
- **let** f = (**fun** x → **fun** y → (x × x) + y) **in** f *2 1*
- **let** f = (**fun** x → **fun** y → (x × x) + y) **in** (f *2*) *1*

L'essence de la programmation fonctionnelle

Option 2

Par *curryfication*, c'est à dire en créant des fonctions qui renvoie des fonctions. OCaml fournit du sucre syntaxique pour cela:

Exemple

Tous ces programmes sont équivalents:

- **let** f x y = (x × x) + y **in** f 2 1
- **let** f = (**fun** x y → (x × x) + y) **in** f 2 1
- **let** f = (**fun** x → **fun** y → (x × x) + y) **in** f 2 1
- **let** f = (**fun** x → **fun** y → (x × x) + y) **in** (f 2) 1
- ((**fun** x → **fun** y → (x × x) + y) 2) 1

L'essence de la programmation fonctionnelle

Option 2

Par *curryfication*, c'est à dire en créant des fonctions qui renvoie des fonctions. OCaml fournit du sucre syntaxique pour cela:

Exemple

Tous ces programmes sont équivalents:

- `let f x y = (x * x) + y in f 2 1`
- `let f = (fun x y -> (x * x) + y) in f 2 1`
- `let f = (fun x -> fun y -> (x * x) + y) in f 2 1`
- `let f = (fun x -> fun y -> (x * x) + y) in (f 2) 1`
- `((fun x -> fun y -> (x * x) + y) 2) 1`

Conséquence

Application partielle:

```
let g = f 2 in g 1 + g 2  (* g est une version specialisee de f *)
```

Fonctions de première classe: points clés

- Fonctions anonymes
  fun x → 1+x
- Fonctions en argument et/ou en résultat
  let appi f = **fun** x → 1 + f x
- Accès hors des “frontières” de la fonction
  let a = 10 **in** (**fun** x → a + x)
  fun x → **fun** y → x + y
- Récusivité (mutuelle)
  let rec f x = **if** x = 0 **then** 0 **else** f (x/2) + 1 **in** f 32
- Application partielle
  let add10 = (+) 10

L'appel par valeur

Quiz

Quelle est la valeur de, et qu'affiche l'expression suivante?

```
let succ x =
  printf "succ %d; " x;
  x+1 in
let plus x y =
  printf "plus %d %d; " x y;
  x + y in
plus (succ 1) (succ (succ 2)) ;;
```

L'appel par valeur

Quiz

Quelle est la valeur de, et qu'affiche l'expression suivante?

```
let succ x =
  printf "succ %d; " x;
  x+1 in
let plus x y =
  printf "plus %d %d; " x y;
  x + y in
plus (succ 1) (succ (succ 2)) ;;
```

Réponse

```
succ 2; succ 3; succ 1; plus 2 4; - : int = 6
```

L'appel par valeur

Quiz

Quelle est la valeur de, et qu'affiche l'expression suivante?

```
let succ x =
  printf "succ %d; " x;
  x+1 in
let plus x y =
  printf "plus %d %d; " x y;
  x + y in
plus (succ 1) (succ (succ 2)) ;;
```

Réponse

```
succ 2; succ 3; succ 1; plus 2 4; - : int = 6
```

Moralité

OCaml évalue les arguments de fonction avant de les passer à la fonction, "de droite à gauche"

Représentation mémoire des fonctions

Comme toutes les valeurs, les fonctions doivent être représentées en mémoire. La représentation mémoire d'une fonction est appelée une *fermeture*.

Approximation 1

Une fermeture est un bloc de tag 247 et de taille 1 dont la donnée est l'offset d'un saut vers le code de la fonction

Représentation mémoire des fonctions

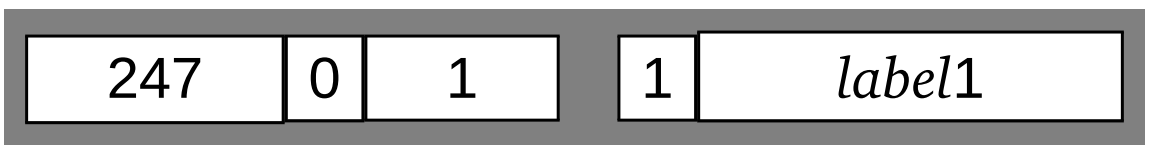
Comme toutes les valeurs, les fonctions doivent être représentées en mémoire. La représentation mémoire d'une fonction est appelée une *fermeture*.

Approximation 1

Une fermeture est un bloc de tag 247 et de taille 1 dont la donnée est l'offset d'un saut vers le code de la fonction

Remarque

On représentera cet offset par un *label* (un nom fixe donné à position dans le code)



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
```

Description informelle

Instructions

`closure L1, 0` crée une fermeture de label L1, puis met son adresse dans A (le code de `f`)

`apply 1` lance l'execution de `f` (A contient la fermeture); son argument est au sommet de S

`acc 0` `f` accède à son premier argument

`return 1` `f` "rend la main", en dépilant 1 case de la pile; son retour, 5, est dans A

Description informelle

Instructions

closure L1, 0	crée une fermeture de label L1, puis met son adresse dans A (le code de f)
apply 1	lance l'execution de f (A contient la fermeture); son argument est au sommet de S
acc 0	f accède à son premier argument
return 1	f "rend la main", en dépilant 1 case de la pile; son retour, 5, est dans A

Où est-ce que l'on retourne quand on fait return?

Sauvegardes/restaurations lors des appels/retours

Où repartir à la fin d'un appel de fonction?

Besoin de mémoriser PC lors de l'appel

- apply 1 sauve le PC sur la pile
  (en dessous de l'argument)
- return 1 s'attend à trouver sur la pile un PC à restaurer
  (après avoir enlevé un élément de la pile)

Sauvegardes/restaurations lors des appels/retours

Où repartir à la fin d'un appel de fonction?

Besoin de mémoriser PC lors de l'appel

- apply 1 sauve le PC sur la pile
  (en dessous de l'argument)
- return 1 s'attend à trouver sur la pile un PC à restaurer
  (après avoir enlevé un élément de la pile)

Remarque

En fait, deux autres éléments sont sauvegardés et restaurés:
env et extra_args (cf. plus loin)

Exemple

Quiz

Compiler à la main la fonction
fun x $\rightarrow$ (**fun** y $\rightarrow$ x + y)

Exemple

Quiz

Compiler à la main la fonction
fun x → (**fun** y → x + y)

Tentative

closure **L1**, 0
return 0
L1: closure **L2**, 0
return 1
L2: acc 0
??? *comment accéder à* x*?*
addint
return 1

Exemple

Environnements

Solution

Stocker dans un *environnement* toutes les valeurs définies dans la fonction courante

- on étend la MV avec un registre *env* qui pointe toujours sur la fermeture de la fonction courante
- la fermeture enregistre label + éléments de l'environnement
- on accède à l'environnement grâce à une instruction spéciale

Environnements

Solution

Stocker dans un *environnement* toutes les valeurs définies dans la fonction courante

- on étend la MV avec un registre env qui pointe toujours sur la fermeture de la fonction courante
- la fermeture enregistre label + éléments de l'environnement
- on accède à l'environnement grâce à une instruction spéciale

Approximation 2

Une fermeture est un bloc de tag 247 et de taille $n+1$, dont le premier champ est le label du code de la fonction, et les n suivants forment l'environnement

Exemple

La fonction `f` *précédente a pour fermeture*

header: size 3, tag 247	*label de* `f`	*val de* `c`	*val de* `g`

Environnements

Instructions

closure l, n	met dans A un pointeur vers une fermeture avec n éléments dans son environnement, ôtés de A et du sommet de S
envacc n	met dans A le contenu de la n-ème case de l'environnement courant

Exemple

- *accès à la valeur de* `c`*:* envacc 1
  (le registre env pointe vers une fermeture dont la case 1 vaut 10)
- *accès à la valeur de* `g`*:* envacc 2

Exemple

Quiz

Compiler à la main la fonction
fun x $\rightarrow$ (**fun** y $\rightarrow$ x + y)

Exemple

Quiz

Compiler à la main la fonction
fun x → (**fun** y → x + y)

Réponse

closure **L1**, 0

return 0
L1: acc 0
closure **L2**, 1
return 1
L2: envacc 1
addint
return 0

Exemple

Quiz

Compiler à la main la fonction
((**fun** x → (**fun** y → x + y)) 1) 2

Réponse

const 2; push; const 1; push
closure **L1**, 0
apply 1; apply 1
return 0
L1: acc 0
closure **L2**, 1
return 1
L2: envacc 1
addint
return 0

Fonctions récursives

Un appel récursif se fait:

- en chargeant dans A la fermeture courante contenue dans env
- en l'appliquant comme précédemment

Instructions

offsetclosure 0 copie le contenu de env dans A

closurerec l, n comme "closure l, n; push"

Fonctions récursives

Exemple

let rec f x = **if** x = *0* **then** *0* **else** f *(x/2) + 1* **in** f *32*

```
const 32
push
closurerec L1, 0
apply 1
return 1
L1: const 0
eqint
branchifnot L2
const 0
return 0
```

```
L2: const 2
push
acc 1
divint
push
offsetclosure 0
apply 1
offsetint 1
return 0
```

Fonctions récursives mutuelles

Dans le cas de fonctions récursives mutuelles, OCaml utilise une même fermeture pour tout un paquet mutuel

Exemple

```
let rec f x = ... f ... g ...
and g y = ... f ... g ...
```

(cf. tableau)

Fonctions récursives mutuelles

Dans le cas de fonctions récursives mutuelles, OCaml utilise une même fermeture pour tout un paquet mutuel

Exemple

let rec f x = ... f ... g ...
and g y = ... f ... g ...
(cf. tableau)

Approximation 3

Une fermeture est un bloc de tag 247 et de taille $n + m$ dont les n premiers champs sont les labels des fonctions, et les m suivants forment l'environnement

Fonctions récursives mutuelles

Dans le cas de fonctions récursives mutuelles, OCaml utilise une même fermeture pour tout un paquet mutuel

Exemple

let rec f x = ... f ... g ...
and g y = ... f ... g ...
(cf. tableau)

Approximation 3

Une fermeture est un bloc de tag 247 et de taille $n + m$ dont les n premiers champs sont les labels des fonctions, et les m suivants forment l'environnement

Instructions

offsetclosure n copie le contenu de $env + n$ dans A

closurerec $l_1, \ldots, l_n, m$ met dans A la fermeture de labels $l_1 \ldots l_n$ avec m éléments dans son environnement, ôtés de A et du sommet de S

Inefficacité 1: fonctions n-aires

Compiler une fonction n-aire crée n fermetures:

Exemple

fun x → (**fun** y → x+y)

```
closure L1, 0
return 0
L1: acc 0
closure L2, 1          copie l'argument dans l'environnement
return 1
L2: envacc 1           restaure l'environnement dans A
addint
return 0
```

Inefficacité 1: fonctions n-aires

Compiler une fonction n-aire crée n fermetures:

Exemple

fun x → (**fun** y → x+y)

```
closure L1, 0
return 0
L1: acc 0
closure L2, 1          copie l'argument dans l'environnement
return 1
L2: envacc 1           restaure l'environnement dans A
addint
return 0
```

Or, le plus souvent, elle sera appliquée totalement à n arguments (en temps $O(n^2)$)

Optimisation 1: applications n-aires

On permet à une application de s'effectuer sur n arguments. On ajoute un registre $extra\_args$ qui stocke le nombre d'arguments restants à appliquer.

- le code de la fonction n'est executé que quand $extra\_args$ est 0
- sinon, l'execution est *gelée* (une nouvelle fermeture est retournée)

Optimisation 1: applications n-aires

Instructions

apply n A contient une fermeture, le sommet de S contient n arguments puis une zone où sont sauvés $PC/env/extra\_args$. env reçoit A, $extra\_args$ reçoit $n-1$, et on saute au label de la fermeture

push_retaddr empile successivement les contenus de $PC/env/extra\_args$

apply 1...3 idem mais se charge de sauver $PC/env/extra\_args$

grab n si $extra\_args \geq n$, continue avec $extra\_args$ décrémenté de n; sinon, crée une fermeture avec $extra\_args$ arguments (*cf.* tableau) et retourne

restart env pointe vers une fermeture générée par grab; on met ses arguments sur la pile et on incrémente $extra\_args$ d'autant

Exemple

let f x y = x+y **in** (f 1 2) + 1

closure **L1**, 0
push
const 2
push
const 1
push
acc 2
apply 2
offsetint 1
return 2
restart

L1: grab 1
acc 1
push
acc 1
addint
return 2

Inefficacité 2: appels terminaux

Souvent, un appel *via* “apply” se trouve juste avant un “return”. Cela arrive quand l’appel à une fonction est la “dernière chose à faire”:

Exemple

let f x = x + 1 **in** f (2 + 2)

const 2; offsetint 2; push
closure **L1**, 0
apply 1 *sauvegarde le contexte et appelle* f
return 3 *on revient de* f*: on le jette*
L1: acc 0; offsetint 1; return 1

Inefficacité 2: appels terminaux

Souvent, un appel *via* "apply" se trouve juste avant un "return". Cela arrive quand l'appel à une fonction est la "dernière chose à faire":

Exemple

let f x = x + 1 **in** f (2 + 2)

const 2; offsetint 2; push
closure **L1**, 0
apply 1 *sauvegarde le contexte et appelle* f
return 3 *on revient de* f*: on le jette*
L1: acc 0; offsetint 1; return 1

On perd temps et espace à sauver des informations dont on a pas besoin. Pire pour les fonctions récursives:

Exemple

let rec f x = **if** x = 0 **then true else** f (x/2)

Optimistion 2: appels terminaux (*tail calls*)

Instructions

appterm m, n fait moralement comme apply m; return $n - m$, mais sans sauvegarder le contexte entre les deux

Remarque

Une fonction récursive dont tous les appels sont terminaux s'effectuera en pile constante (comme une boucle)

Exemple

let rec f x = **if** x = 0 **then true else** f (x/2) **in** f 32

```
    closurerec L1, 0
    const 32
    push
    acc 1
    appterm 1, 3
L1: acc 0
    push
    const 0
    eqint
    branchifnot L2
    const 1a
    return 1
L2: const 2
    push
    acc 1
    divint
    push
    offsetclosure 0
    appterm 1, 2
```

Quiz: Récursif terminal ou pas?

```
let rec rev_append l1 l2 =
  match l1 with
    [] → l2
  | a :: l → rev_append l (a :: l2)
```

Quiz: Récursif terminal ou pas?

```
let rec flatten = function
    [] → []
  | l::r → l @ flatten r
```

Quiz: Récursif terminal ou pas?

```
let rec map f = function
    [] → []
  | a::l → let r = f a in r :: map f l
```

Quiz: Récursif terminal ou pas?

```
let rec fold_left f accu l =
  match l with
    [] → accu
  | a::l → fold_left f (f accu a) l
```

Quiz: Récursif terminal ou pas?

```
let rec fold_right f l accu =
  match l with
    [] → accu
  | a::l → f a (fold_right f l accu)
```

Quiz: Récursif terminal ou pas?

```
let rec mem x = function
    [] → false
  | a::l → compare a x = 0 || mem x l
```

Quiz: Récursif terminal ou pas?

```
let rec find p = function
  | [] → raise Not_found
  | x :: l → if p x then x else find p l
```

Description précise des instructions

closure n, l	Alloue une fermeture de label l et de n éléments d'environnement pris dans A et au sommet de S. Met un pointeur vers cette fermeture dans A.
apply n	Met *extra_args* à $n-1$; met PC à la valeur du premier champ du bloc pointé par A; met *env* à la valeur de A.
push_retaddr n	empile *extra_args*, puis *env*, puis $PC+n$.
apply 1...3	comme **apply** n mais se charge de faire **push_retaddr**.
return n	dépile n éléments. Si *extra_args* >0, met *extra_args* à *extra_args* $-n$, met PC à la valeur du premier champ du bloc pointé par A, puis met *env* au contenu A. Sinon, trois valeurs sont dépilées et mises respectivement dans PC, *env* et *extra_args*

Description précise des instructions

appterm m, n Enlève les éléments de la pile du m-ème au n-ème. Puis met PC à la valeur du premier champ du bloc pointé par A, met env au contenu de A, et incrémente $extra\_args$ de $n - 1$

grab n Si $extra\_args \geq n$, alors décrémente $extra\_args$ de n. Sinon, alloue dans A une fermeture de label $PC - 3$ (un **restart**) et de $extra\_args + 1$ éléments d'environnement: le contenu de env, puis $extra\_args$ arguments dépilés. Puis, PC, env et $extra\_args$ sont dépilés et restaurés.

restart Soit n la taille de la fermeture pointée par env. Empile les $n - 2$ d'environnement de cette fermeture, puis met env au contenu du premier champ pointé par env.

Description précise des instructions

envacc n met dans A la valeur du n-ème champ du bloc pointé par env

offsetclosure n met dans A un pointeur vers le n-ème bloc pointé par env

closurerec $l_1, \ldots, l_n, m$ Alloue n fermetures contiguës de labels *resp.* $l_1, \ldots, l_n$. La dernière reçoit m éléments d'environnement, pris dans A et au sommet de S. Empile n pointeurs vers ces fermetures.

Plan du cours

Présentation