# Machines virtuelles fonctionnelles (suite)
# Compilation ML $\rightarrow$ Java

## Cours de Compilation Avancée (MI190)

Benjamin Canou
Université Pierre et Maire Curie

Année 2010/2011 – Semaine 3

# Machines virtuelles fonctionnelles (suite)

# La ZAM : machine fonctionnelle stricte

Schéma dérivé de la machine de Krivine :

- Le corps d'une instruction commence par `GRAB`,
- comme les fonctions ont plusieurs arguments, le code ressemble en fait à : `[GRAB;`$n_{args}$`;...;RETURN]`
- les arguments sont passés sur la pile par les instructions `APPLY{1,2,3}` + compteur `extra_args`
- `GRAB` applique la fonction (évaluation stricte) si elle trouve les arguments nécessaires, sinon, elle crée une fermeture.

# La ZAM : application générale

Comment s'exécute le programme suivant ?

```
# open Printf;;

# let separe sep =
  let rec aux i str =
    if i < String.length str then (
      printf "%c%c" str.[i] sep ;
      aux (i + 1) str
    )
  in
  aux 0;;
val separe : char -> string -> unit = <fun>

# separe ',';;
- : string -> unit = <fun>

# separe ',' "toto";;
t,o,t,o,
- : unit = ()
```

Grâce à `CLOSURE`, `APPLY`, `GRAB` et `RETURN`

# La ZAM : CLOSURE

```
Instruct(CLOSURE): {
      int nvars = *pc++;
      int i;
      if (nvars > 0) *--sp = accu;
      Alloc_small(accu, 1 + nvars, Closure_tag);
      Code_val(accu) = pc + *pc;
      pc++;
      for (i = 0; i < nvars; i++) Field(accu, i + 1) = sp[i];
      sp += nvars;
      Next;
    }
```

# La ZAM : APPLY

```
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;
      goto check_stacks;
    }
```

# La ZAM : GRAB

```
Instruct(GRAB): {
      int required = *pc++;
      if (extra_args >= required) {
        extra_args -= required;
      } else {
        mlsize_t num_args, i;
        num_args = 1 + extra_args; /* arg1 + extra args */
        Alloc_small(accu, num_args + 2, Closure_tag);
        Field(accu, 1) = env;
        for (i = 0; i < num_args; i++) Field(accu, i + 2) = sp[i];
       Code_val(accu) = pc - 3; /* Point to the preceding RESTART instr. */
        sp += num_args;
        pc = (code_t)(sp[0]);
        env = sp[1];
        extra_args = Long_val(sp[2]);
        sp += 3;
      }
      Next;
    }
```

RESTART effectue la copie environnement $\rightarrow$ pile.

Compilation : ...RESTART;[GRAB;$n_{args}$;...;RETURN]...

# La ZAM : RETURN

```
Instruct(RETURN): {
      sp += *pc++;
      if (extra_args > 0) {
        extra_args--;
        pc = Code_val(accu);
        env = accu;
      } else {
        pc = (code_t)(sp[0]);
        env = sp[1];
        extra_args = Long_val(sp[2]);
        sp += 3;
      }
      Next;
    }
```

# Compilation ML $\rightarrow$ Java : partie haute

# Exemple de code ML compilable

```
let f = function x -> function y -> x + y;;
let g = f 3;;
let a = g 2;;
let q = g, f 7;;
let r = ((fst q) 9);;

let rec map = function f -> function l ->
   if l = [] then [] else (f (hd l)) :: (map f (tl l));;

let l = 1 :: 2 :: 3 :: [];;
let k = map g l;;
let k2 = map f l;;
```

# Structure de `ml2java`

| | |
|---|---|
| util.ml | utilitaire |
| types.ml | définition des types de mini-ML |
| alex.mll | analyseur lexical |
| asyn.mly | analyseur syntaxique |
| typeur.ml | le typeur |
| env_typeur | environnement |
| intertypeur.ml | toplevel du typeur * |
| eval.ml | évaluateur * |
| env_eval | environnement de l'évaluateur * |
| intereval.ml | toplevel de l'évaluateur * |
| env_trans.ml | environnement du traducteur |
| lift.ml | un pseudo $\lambda$-lifting |
| trans.ml | le traducteur vers un LI |
| prod.ml | le traducteur LI vers java |
| comp.ml | le compilateur complet |
| maincomp.ml | l'analyseur de la ligne de commande |

# (Retour sur l')analyse lexicale

## Fichier → Suite de lexèmes

```
{
open Util ;; open Asyn ;;
let keyword_table = Hashtbl.create 100 ;;
do_list (fun (str,tok) -> Hashtbl.add keyword_table str tok) [
  "else", ELSE;  "function", FUNCTION;
  "if", IF;      "in", IN;
  "let", LET;    "rec", REC;
  "ref", REF;    "then", THEN
];;
}

rule main = parse
    [' ' '\010' '\013' '\009' '\012'] +
      { main lexbuf }
  | ['A'-'Z' 'a'-'z'  ] ( '_' ? ['A'-'Z' 'a'-'z'  '0'-'9' ] ) *
      { let s = get_lexeme lexbuf in
          try  Hashtbl.find keyword_table s
          with Not_found -> IDENT s }
  | ['0'-'9']+
    | '0' ['x' 'X'] ['0'-'9' 'A'-'F' 'a'-'f']+
    | '0' ['o' 'O'] ['0'-'7']+
    | '0' ['b' 'B'] ['0'-'1']+
      { INT (int_of_string(get_lexeme lexbuf)) }
```

# (Retour sur l')analyse lexicale

## Fichier → Suite de lexèmes

```
| "(" { LPAREN }
| ")" { RPAREN }
| "," { COMMA }
| "->" { MINUSGREATER }
| "::" { COLONCOLON }
| ":=" { COLONEQUAL }
| ";" { SEMI }
| ";;" { SEMISEMI }
| "=" { EQUAL }
| "[" { LBRACKET }
| "]" { RBRACKET }
```

# (Retour sur l')analyse syntaxique

## Suite de lexèmes → AST

### Défintion des tokens (utilsés par le lexeur)

```
%token <string> IDENT
%token <string> PREFIX
%token <string> INFIX

%tOoken <int> INT
%token <float> FLOAT
%token <string> STRING

%token EOF

%token EQUAL            /* "=" */
%token LPAREN           /* "(" */
/* ... */
```

# (Retour sur l')analyse syntaxique

## Suite de lexèmes → AST

### Deux niveaux d'expressions : simples

```
Simple_expr :
        Struct_constant
          { Const $1 }
      | IDENT
          {Var $1}
      | LPAREN RPAREN
          { Const  Unit }
      | LBRACKET RBRACKET
         { Const Emptylist}
      | LPAREN Expr RPAREN /* expr -> simple*/
          { $2 }
      | PREFIX Simple_expr
          { Unop ( $1,$2)}
```

# (Retour sur l')analyse syntaxique

## Suite de lexèmes → AST

### Deux niveaux d'expressions : construites

```
Expr :
        Simple_expr
          { $1 }
      | Simple_expr Simple_expr_list   %prec prec_app
          {make_apply $1 $2}
      | REF Expr
          {Ref $2}
      | Simple_expr COMMA Expr
          {Pair ($1,$3)}
      | Simple_expr  COLONCOLON Expr
          {Cons ($1,$3)}
      | Simple_expr  COLONEQUAL Expr
          {Binop (":=",$1,$3)}
      | IF Expr THEN Expr ELSE Expr
          { Cond($2, $4, $6) }
      | LET IDENT EQUAL Expr IN Expr
          { Letin (false,$2,$4,$6) }
      | FUNCTION IDENT MINUSGREATER Expr
          { Abs($2,$4) }
```

# (Retour sur l')analyse syntaxique

## Suite de lexèmes → AST

### Résolution des ambiguités avec des priorités

```
%right prec_app

Expr :
      /* ... */
      | Simple_expr Simple_expr_list   %prec prec_app
  {make_apply $1 $2}
      /* ... */

%right prec_if

Expr :
      /* ... */
      | IF Expr THEN Expr ELSE Expr  %prec prec_if
          { Cond($2, $4, $6) }
      | IF Expr THEN Expr  %prec prec_if
          { Cond($2, $4, Const Unit)}
      /* ... */
```

# (Retour sur l')analyse syntaxique

## Suite de lexèmes → AST

Programme :

```
/* ... */
%start implementation
%type <Types.ml_phrase> implementation

%%

implementation :
        Expr SEMISEMI
          { Expr $1 }
      | LET IDENT EQUAL Expr SEMISEMI
          { Decl (Let(false,$2,$4))}
      | LET REC IDENT EQUAL Expr SEMISEMI
          { Decl (Let(true,$3,$5))}
      | EOF
          { raise End_of_file }
;
/* ... */
```

# Arbre de Syntaxe Abstraite

```
type ml_expr =
 | Const of ml_const | Var of string
 | Unop of string  * ml_expr   | Binop of string  * ml_expr * ml_expr
 | Pair of ml_expr * ml_expr   | Cons of ml_expr * ml_expr
 | Cond of ml_expr * ml_expr * ml_expr
 | App of ml_expr * ml_expr    | Abs of string * ml_expr
 | Letin of bool * string * ml_expr * ml_expr
 | Ref of ml_expr
 | Straint of  ml_expr * ml_type (* <- après typage *)
and ml_const =
 | Int of int | Float of float   | Bool of bool
 | String of string | Emptylist
 | Unit

type ml_decl = Let of bool * string * ml_expr

type ml_phrase = Expr of ml_expr | Decl of ml_decl
```

# Typage

Types :

```
type vartype =
 | Unknown of int | Instanciated of ml_type
and consttype =
 | Int_type    | Float_type  | String_type
 | Bool_type   | Unit_type
and ml_type =
 | Var_type of vartype ref          | Const_type of consttype
 | Pair_type of ml_type * ml_type   | List_type of ml_type
 | Fun_type of ml_type * ml_type    | Ref_type of ml_type
```

# Typage

(ce n'est pas le propos de ce cours)

- `typeur.ml` :
  Algorithme de Hindley-Milner :
  1. On introduit des inconnues de types dans les expressions,
  2. on les spécifie par effet de bord en utilisant le contexte (unification),
  3. s'il reste des inconnues sous un let, on obtient une valeur polymorphe (généralisation).
  4. Erreur de type si l'unification ne marche pas.
- `env_typeur.ml` :
  Gestion de l'environnement et environnement initial (types des primitives).

# Suppression des fonction locales : $\lambda$-lifting

1. On part d'une déclaration locale,
```
let f x y =
  let g a = a + x in
  g y ;;
```
2. on la remonte au niveau au dessus,
```
let g a = a + x ;;
let f x y =
  g y ;;
```
3. et on passe l'environnement en paramètre explicitement.
```
let g x a = a + x ;;
let f x y =
  g x y ;;
```

Plus compliqué avec les fonctions mutuellement récursives.

# Langage intermédiaire

Instructions :

```
type li_const =
 | INT of int     | FLOAT of float
 | BOOL of bool   | STRING of string
 | EMPTYLIST      | UNIT
type li_instr =
 | CONST of li_const | VAR   of string * li_type
 | AFFECT of string * li_instr
 | IF  of li_instr * li_instr * li_instr
 | PRIM of (string * li_type) * li_instr list
 | APPLY of li_instr * li_instr
 | RETURN of li_instr
 | BLOCK of (string  * li_type * li_instr) list  * li_instr
 | FUNCTION of string * li_type * int * (string list * li_type) * li_instr
```

# Langage intermédiaire

Types :

```
type li_const_type =
 | INTTYPE    | FLOATTYPE | BOOLTYPE
 | STRINGTYPE | UNITTYPE
type li_type =
| ALPHA
| CONSTTYPE of li_const_type
| PAIRTYPE | LISTTYPE
| FUNTYPE
| REFTYPE
```

# Schéma de compilation vers LI

## format des règles

`[ e ]E -> INSTRUCTION(params)`

- e le motif de code à compiler
- E l'environnement de compilation
  - g : l'environnement des noms
  - r : vrai s'il faut retourner le résultat
  - d : nom d'une éventuelle variable à affecter
  - t : type LI
- INSTRUCTION le constructeur d'instruction
- params ses paramètres : constantes ou appels de règles

# Schéma de compilation vers LI

## constantes

```
[ c ](g,F,"",t)    -> CONST(c,g,t)
[ c ](g,T,"",t)    -> RETURN(CONST(c,g,t))
[ c ](g,F,D,t)     -> AFFECT(D,CONST(c,g,t))
```

# Schéma de compilation vers LI

## variables

```
[ v ](g,F,"",t)    -> VAR(v,g,t)
[ v ](g,T,"",t)    -> RETURN(VAR(v,g,t))
[ v ](g,F,D,t)     -> AFFECT(D,VAR(v,g,t))
```

# Schéma de compilation vers LI

## conditionnelle

```
[ if v1 then e2 else e3 ](g,r,d,t) ->
  IF([v1](g,F,"",bool),[e2](g,r,d,t),[e3](g,r,d,t))
[ if e1 then e2 else e3](g,r,d,t) ->
  [let v1 = e1 in if v1 then e2 else e3](g,r,d,t)
```

# Schéma de compilation vers LI

## application

```
[ v1 v2 ](g,F,"",t) -> APPLY([v1](g,F,"",_),[v2](g,F,"",_))
[ v1 v2 ](g,T,"",t) -> RETURN(APPLY([v1](g,F,"",_),[v2](g,F,"",_)))
[ v1 v2 ](g,F,D,t)  -> AFFECT(D,APPLY([v1](g,F,"",_),[v2](g,F,"",_)))
[ e1 e2 ](g,r,s,t)  -> [let v1 = e1 and v2 = e2 in v1 v2](g,r,s,t)
```

# Schéma de compilation vers LI

## déclaration locale

```
[let v1 = \x.e1 in e2](g,r,d,t) -> ERREUR!!!
[let v1 = e1 in e2](g,r,d,t)   ->
   BLOCK(w1=N(v1),t,[e1](g,F,(w1,t),e2((v1,w1)::g,r,d,t))
```

PLus de fonction locale ($\lambda$-lifting)

# Schéma de compilation vers LI

## déclaration globales

```
[let v1 = \x.e1 ](g,t)     ->
  FUNCTION(w1=N(v1),t,1,[x],[e1](g,T,"",_))
[let rec v1 = \x.e1](g,t) ->
  FUNCTION(w1=N(v1),t,1,[x],[e1]((x,N(x))::(v1,w1)::g,T,"",_))
[Let v1 = e1](g,t)         ->
  VAR(w1=N(v1),t,[e1](g,false,"w1",t))
[let rec v1 = e1](g,t)     ->
  ERREUR
```

# Compilation ML $\rightarrow$ Java : partie basse

# Compilation vers Java

On peut utiliser la bibliothèque d'exécution de Java pour implanter celle de ML :

- Gestion mémoire,
- types primitifs,
- bibliothèque,
- exceptions, etc.

En contrepartie : il faut coller au système de types.

# Définition des valeurs

Il faut un typeunique pour les valeurs, qui peut encapsuler différent types.

1. On définit un type abstrait avec les opérations communes à toutes la valeurs,
2. et on dérive une classe pour chaque type ML.

```
abstract class MLvalue extends Object {
  abstract void  print();
  /* on pourrait ajouter equals, serialise, etc. */
}
```

# Types primitifs

```
class MLunit extends MLvalue {
  MLunit(){}
  public void print(){System.out.print("()");}
}

class MLbool extends MLvalue {
  private boolean val;
  MLbool(boolean a){val=a;}
  public void print(){
    if (val) System.out.print("true");
    else System.out.print("false");
  }
  public boolean MLaccess(){return val;}
}
```

# Types primitifs

```
class MLint extends MLvalue {
   private int val;
   MLint(int a){val=a;}
   public void print(){System.out.print(val);}
   public int MLaccess(){return val;}
}

class MLdouble extends MLvalue {
   private double val;
   MLdouble(double a){val=a;}
   public void print(){System.out.print(val);}
   public double MLaccess(){return val;}
}

class MLstring extends MLvalue {
  private String val;
  MLstring(String a){val=a;}
  public void print(){System.out.print("\""+val+"\"");}
  public String MLaccess(){return val;}
}
```

# Constructions

```
class MLpair extends MLvalue
{
  private MLvalue MLfst;
  private MLvalue MLsnd;

  MLpair(MLvalue a, MLvalue b){MLfst=a; MLsnd=b;}

  public MLvalue MLaccess1(){return MLfst;}
  public MLvalue MLaccess2(){return MLsnd;}
  public void print(){System.out.print("(");
                      MLfst.print();
                      System.out.print(",");
                      MLsnd.print();
                      System.out.print(")");}

}
```

# Primitives

```
class MLruntime {
  public static MLbool MLtrue = new MLbool(true);
  public static MLbool MLfalse = new MLbool(false);
  public static MLunit MLlrp = new MLunit();
  public static MLlist MLnil = new MLlist(null,null);
  /* ... */
  public static MLint MLaddint(MLint x, MLint y) {
       return new MLint(x.MLaccess()+y.MLaccess());
  }
  /* ... */
}
```

# Compilation des fonctions

On ne peut pas utiliser le mécanisme de méthodes de Java :

- Les fonctions sont des valeurs.
- Application partielle.

On utilise alors une méthode Java pour le corps de la fonction, et on l'emballe dans une machinerie implantant ces propriétés.

# Compilation des fonctions

## Classe de base des fonctions :

```
abstract class MLfun extends MLvalue {
  public int MLcounter;
  protected MLvalue[] MLenv;

  MLfun(){MLcounter=0;}
  MLfun(int n){MLcounter=0;MLenv = new MLvalue[n];}

  public  void  MLaddenv(MLvalue []O_env,MLvalue a)
   { for (int i=0; i< MLcounter; i++) {MLenv[i]=O_env[i];}
     MLenv[MLcounter]=a;MLcounter++;}

  abstract public MLvalue invoke(MLvalue x);

  public void print(){
     System.out.print("<fun> [");
     for (int i=0; i< MLcounter; i++)
           MLenv[i].print();
     System.out.print("]");
  }
}
```

# Compilation des fonctions

Exemple : `let app = function fx -> function x -> fx x;;`

```
class MLfun_app___76 extends MLfun {
  private static int MAX = 2;
  MLfun_app___76() {super();}
  MLfun_app___76(int n) {super(n);}
  public MLvalue invoke(MLvalue MLparam){
    if (MLcounter == (MAX-1)) {
      return invoke_real(MLenv[0], MLparam);
    }
    else {
      MLfun_app___76 l = new MLfun_app___76(MLcounter+1);
      l.MLaddenv(MLenv,MLparam);
      return l;
    }
  }
  MLvalue invoke_real(MLvalue fx___77, MLvalue x___78) {
    return ((MLfun)fx___77).invoke(x___78);
  }
}
```