Cours 2 : machine de Krivine (compilation du λ-calcul)

La principale difficulté de la mise en œuvre d'un langage fonctionnel vient de la traduction du contrôle implicite de l'exécution dans un modèle de calcul où il devient explicite. Comme le modèle sous-jacent des langages fonctionnels est le λ-calcul, nous nous intéressserons tout d'abord à la compilation du λ-calcul en étudiant une machine abstraite très simple appelée la machine de Krivine.

1 - Compilation du λ-calcul

Cette section décrit une machine à environnement pour l'évaluation paresseuse, appelée machine de Krivine, qui est le modèle le plus simple pour la compilation du λ-calcul.

1.1 Évaluateur de λ-calcul

Le schéma d'évaluation d'un λ-terme est le suivant :

env	terme	pile	env	terme	pile
e	c	S	e	c	S
e	MN	S	e	M	$(N.e :: S)$
e_ρ	$\lambda x.M$	$s :: S$	$(e,s)_{\rho,x}$	M	S
$(e,(M.e2))_{\rho,x}$	x	S	$e2_\rho$	M	S
$(e,a)_{\rho,y}$	x	S	e_ρ	x	S

où l'environnement contient des couples (termes,environnement), c'est-à-dire des valeurs calculables.
Le type Caml suivant correspond aux termes ou expressions de ce λ-calcul étendu :

```
type expr =
    Kapp of expr * expr
|   Kid of string
|   Klambda of string * expr
;;
```

1.2 Instructions de la machine

La machine suivante (dérivée de la machine de Krivine) comporte 3 instructions la compilation du λ-calcul et un étiquetage.

- Grab : met le sommet de pile dans l'environnement
- Push label : met le label et l'environnement dans la pile
- Acces n : accès à la n-ième variable de l'environnement
- Label label : étiquetage

Les types Caml suivants :

```
type instr =
     Grab           (* met le sommet de la pile dans l'environnement *)
|    Push of int    (*  *)
|    Access of int  (* acces \'a la n-ieme variable de l'environnement *)
|    Label of int   (* etiquetage d'une partie de code *)
;;
type instrI =
     GrabI          (* met le sommet de la pile dans l'environnement *)
|    PushI of int   (*  *)
|    AccessI of int (* acces \'a la n-ieme variable de l'environnement *)
;;
```

correspondent respectivement aux instructions de l'assembleur et aux instructions machine (après assemblage).

1.3 Schéma de compilation

On note $[M]_\rho$ la compilation du terme M dans l'environnement ρ.

- $[MN]_\rho$ = Push l; $[M]_\rho$; Label l; $[N]_\rho$
- $[\lambda x.M]_\rho$ = Grab; $[M]_{\rho,x}$
- $[x]_\rho$ = Acces n (où n est la position de x dans ρ)

Les deux fonction `new_label` et `reset_label` manipulent le compteur des nouveaux numéros de label.

```
let new_label,reset_label  =
  let c = ref 0 in
      ( function () -> (incr c);!c),(function () -> c:=0;!c)
;;
```

La variable globale `aCompiler` contient des morceaux de code à compiler par la suite.

```
let aCompiler = ref ([]:instr list);;
```

Les deux fonctions principales de ce compilateur sont `mk_lab` pour la création et le placement de nouveaux labels et la fonction `compiler` qui traduit les λ-termes en instructions machine.

```
let rec mk_lab env term = let lab = new_label() in
        (aCompiler := ( Label lab)::((compiler env term)@(!aCompiler));
         lab)
and
  compiler env = function
  Kid v         -> [ Access ((index v env)+1) ]
| Klambda (v,e) ->  Grab::(compiler (v::env) e)
| Kapp (f,a)    ->  let u = (mk_lab env a) in
                    Push u :: (compiler env f)
;;
```

Enfin la fonction `cc` initialise les variables globales de compilation et compile un terme à partir d'un environnement vide.

```
let cc e = reset_label();
           aCompiler :=[];
           let sub_code = (compiler [] e)
           in sub_code @ (!aCompiler)
;;
```

1.4 Schéma d'évaluation

Une fois les λ-termes traduits en instructions machine, il est nécessaire d'avoir un interprète de cette machine virtuelle. Voici le schéma d'évaluation des instructions de cette machine :

env	code	pile	env	code	pile
e	Push l;C	S	e	C	$(l.e) :: S$
e	Grab; C	$s :: S$	(e, s)	C	S
e	(*)Grab;C	()	arrêt	sur	$(e, *)$
$(e, (l.e2))$	Acc 0;C;Label l; C2	S	$e2$	C2	S

Un des points importants est le critère d'arrêt. L'instruction *Grab* correspond à une abstraction. S'il n'y a pas d'argument sur la pile, cette abstraction ne trouve donc pas d'argument et le calcul s'arrête. Il n'y a pas de réduction sous le λ.

Le programme Caml suivant effectue une passe d'assemblage des instructions de type `instr` et évalue le résultat, une liste d'instructions `instrI`.

Pour faire disparaître les labels, il est nécessaire de connaître la longueur de chaque instruction :

```
let longueur_code = function
     Grab     -> 1
|    Push _   -> 1
|    Access _ -> 1
|    Label _  -> 0
;;
```

La fonction `pass_one` associe à chaque label l'adresse de l'instruction étiquetée.

```
let pass_one =
  let rec pass_rec pc = function
     [] -> []
  |   Label n :: rest -> (n,pc):: pass_rec pc rest
  |  instr :: rest  -> pass_rec (pc+(longueur_code instr)) rest
  in  pass_rec 1
;;
```

La fonction `assemble` traduit du code assembleur (`intsr list`) en code machine (`instrI list`).

```
let assemble code =
  let list_label = pass_one code
  in
    let rec asm = function
      [] -> []
    | Label n :: rest  -> asm rest
    | Push  l:: rest ->  PushI  (assoc l list_label) :: asm rest
    | Grab :: rest   ->  GrabI :: asm rest
    | Access n :: rest ->  AccessI n :: asm rest
    in
      asm code
;;

type closure = C of int * (closure list) ref
;;
let rec nth l a = match l with
  [] -> failwith "nth"
| h::t -> if a=1 then h else nth t (a-1)
;;
```

La fonction `interprete` exécute le code machine passé comme argument et retourne une fermeture (`closure`).

```
let interprete code =
  let rec interp env pc stack = match (nth code pc) with
     AccessI n -> (try
                     let u = nth env  n
                     in
                       match u with
                         (C  (n,e)) -> interp !e n stack
                    with
                      x -> (C (pc, ref env)))
  | PushI n  -> interp env (pc+1) ((C (n,ref env))::stack)
  | GrabI    -> (match  stack with
                   [] -> (C  (pc,ref env))
                | so::s -> interp (so::env) (pc+1) s)
  in
    interp   [] 1  []
;;
```

Enfin la fonction `kall` interprète le code compilé et assemblé d'un λ-terme.

```
let kall exp =    interprete (assemble (cc exp))
;;
```

1.4 Exemples

```
let ex1 =
(Kapp
 ((Kapp
    ((Klambda ("f",(Klambda ("x",(Kapp ((Kid "f"),(Kid "x")))))),
        (Klambda ("x",(Kid "x")))),
          (Klambda ("x",(Kid "x")))
          ;;
cc ex1;;
assemble (cc ex1);;
interprete (assemble(cc ex1));;
```

Bibliographie

- S. L. Peyton Jones. “Mise en œuvre des langages fonctionnels” Dunod, 1997.