Langages Fonctionnels Stricts

comparaison et convergence de Scheme et ML

Emmanuel Chailloux
LIENS - LITP

Université de Liège
29 octobre 1993

SOMMAIRE

SCHEME

- Scheme (1975) dialecte Lisp (1960)
- Norme IEEE Std 1178-1990
- LF strict muni de traits impératifs
- typé dynamiquement
- portée statique (déclaration locale)
- portée dynamique (déclaration locale)
- pas de différence données/programmes
- call_cc

CAML

- Caml (1985) dialecte ML (1980)
- LF strict muni de traits impératifs
- typé statiquement
- polymorphe paramétrique
- liaison statique
- exceptions
- déclarations de types

Scheme

```
4
;Value: 4

(define x 3)
;Value: x

(define id (lambda (x) x))
;Value: id

(id id)
;Value: #[compound-procedure 4 id]
```

Caml

```
4;;
- : int = 4

let x = 3;;
x : int = 3

let id = function x -> x;;
id : 'a -> 'a = <fun>

id id;;
- : 'a -> 'a = <fun>
```

Scheme	Caml
(*define* K (*lambda* (x y) x)) ;Value: k	*let* K = *fun* x y -> x;; K : 'a -> 'b -> 'a = <fun>
(K id K) ;Value: #[compound-procedure 4 id]	K id K;; - : 'a -> 'a = <fun>
(K id) ;Wrong number of arguments 1 ;within procedure #[compound-procedure 5 k] ;minimum/maximum number of arguments: 2 2 ;Abort!	K id ;; - : 'a -> 'b -> 'b = <fun>

(*define* K2
  (*lambda* (x)
  (*lambda* (y) x)))
;Value: k2	

(K2 id) ;Value: #[compound-procedure 6]	

Scheme	Caml
statique	statique

(*define* g
  (*let* ((x 3))
  (*lambda* (y) (+ x y))))
;Value: g
 	
(g 2)
;Value: 5
 	
(*define* (h a)
  (*let* ((x 10)) (g a)))
;Value: h
 	
(h 2)
;;Value: 5
	let g =
	  let x = 3 in
	  function y -> x + y;;
	g : int -> int = <fun>
	
	g 2;;
	- : int = 5
	
	let h a =
	  let x = 10 *in* g a;;
	h : int -> int = <fun>
	
	h 2;;
	- : int = 5

Scheme	Caml
dynamique	statique
(*define* x 3) ;Value: x	*let* x = 3;; x : int = 3
(*define* (f y) (+ x y)) ;Value: f	*let* f y = x + y;; f : int -> int = <fun>
(f 0) ;Value: 3	f 0;; - : int = 3
(*define* x 0) ;Value: x	*let* x = 0;; x : int = 0
(f 0) ;Value: 0	f 0;; - : int = 3

Scheme : variables globales mutables,
Caml : variables globales non mutables sauf précision (*ref*) .

Scheme	Caml
dynamique	statique
(*number?* 3) ;Value: #T	
(*define* (add x) (+ x ())) ;Value: add	*let* add x = x + [];; >let add x = x + [];; > ^^ > Expression of type 'a list > cannot be used with type int
(add 0) ;Illegal datum in second argument ;Abort!	

En Scheme : tous les types ont un prédicat explicite (number?) ($\Rightarrow$ information de type dans les valeurs) un seul type somme extensible ($\Rightarrow$ représentation uniforme des données)

En Caml : pas d'erreur de typage à l'exécution ($\Rightarrow$ l'information de typage peut être perdue à l'exécution).

Scheme	Caml
dynamique	statique
(*define* x 3) ;Value: x	*let* x = ref 3;; x : int ref = ref 3
(*define* (addx y) (+ x y)) ;Value: addx	*let* addx y = !x + y;; addx : int -> int = <fun>
(addx 0) ;Value: 3	addx 0;; - : int = 3
(*set!* x #T) ;Value: 3 x ;Value: #T	x := true;; > ^^^^ > Expression of type bool > cannot be used with type int
(addx 0) ;Illegal datum in first argument	

Paires

Scheme	Caml
(*define* p (cons 1 #T)) ;Value: p ; (1 . #T) (pair? p) ;Value: #T (define l '(1 . (#T . ()))) ;Value: l ; (1 #T) (pair? l) ;Value: #T (list? l) ;Value: #T	*let* p = (1,3);; p : int * bool = 1, true let l = (1,(true,()));; l : int * (bool * unit) = 1, (true, ())

En Scheme les listes sont des paires spéciales ou nil
En ML c'est un autre type de données :

```
type 'a LIST = CONS of 'a * 'a LIST
             | NIL;;
Type LIST defined.
```

Elles sont homogènes en ML.

En Scheme toute valeur est mutable (RW).

Scheme

```
(define m  (cons 1 #T))
;Value: m ; (1 . #T)
(set-car! m "X25")
;No value
m
;Value: ("X25" . #T)
```

En ML seules les valeurs de types mutables le sont (RW), les autres ne le sont pas (RO).

```
let m = ref [];;
>let m = ref [];;
>^^^^^^^^^^^^^^^
> Cannot generalize 'a in 'a list ref
```

Heureusement, sinon x serait de type : $\forall\alpha.\alpha\ list\ ref$, et on pourrait écrire :

`x:= cons 1 !x;;` et `x` aurait toujours le type général

`x:= cons true !x;;` marcherait aussi

$\Rightarrow$ ce qui casserait le système de types de ML.

En Caml, dans le langage :

```
let rec count1  e =
match e with
   NIL ->  0
 | CONS (1,t) -> 1 + count1 t
 | CONS (_,t) -> count1 t;;
```

```
count1 : int LIST -> int = <fun>
```

Pas de cas non spécifié, motif linéaire ($\Rightarrow$ comment tester des fermetures?).

En Scheme, vu comme une bibliothèque.
Exemple en bigloo : `eq?` est utilisé comme prédicat d'égalité.

```
(match-case (list (K 1) (K 1))
   ((?x ?x) 'OK)
   (( ?f ?g) (if (eqv? (f 2) (g 2)) 'GOOD 'BAD)))
```

```
GOOD
```

Motifs non linéaires, mais problème avec l'égalité physique sur des fermetures.

Produit des éléments d'une liste en Caml

```
let rec list_mult l =
  match l with [] -> 1
    |  h::t -> h * list_mult t
;;
```

En utilisant les exceptions :

```
exception Zero;;
```

```
Exception Zero defined
```

```
let rec list_mult_aux l =
  match l with [] -> 1
    | 0::_ -> raise Zero
    | h::t -> h * list_mult_aux t
;;
```

```
list_mult_aux : int list -> int = <fun>
```

```
let list_mult l =
  try list_mult_aux l
  with Zero  -> 0
;;
```

```
list_mult : int list -> int = <fun>
```

Produit des éléments d'une liste en Scheme

```
(define (list_mult_aux return l)
  (letrec ((r (lambda (l) (print l)
              (if (null? l) 1
              (if (= (car l) 0) (return 0)
               (print(* (car l) (r (cdr l))))))))))
     (r l)))

(define (list_mult l)
   (call/cc (lambda (c) ((list_mult_aux c l)))))

(list_mult '( 1 2 3))
```

```
(1 2 3)
(2 3)
(3)
()
3
6
6
6
```

```
(list_mult '( 1 2 3 0 4 5))
```

```
(1 2 3 0 4 5)
(2 3 0 4 5)
(3 0 4 5)
(0 4 5)
0
```

```
(define (main x)
    (+ (call/cc (lambda (k0) (f1 k0 x))) 3) )

(define (f1 k0 x)
    (+ (call/cc (lambda (k1) (f2 k0 k1 x))) 2) )

(define (f2 k0 k1  x)
    (+ (call/cc (lambda (k2) (f3 k0 k1 k2 x))) 1) )

(define (f3 k0 k1 k2 x)
   (cond ((< x 10) (k2 x))
         ((< x 100) (k1 x))
         (#t  (k0 x))))
```

```
(main 0)
6
(main 10)
15
(main 100)
103
```

Goto fonctionnel en séquentiel, intéressant pour décrir e des co-routines.
Problèmes de typage (comme les références) : n'existe pas en standard dans Caml.
Mise en œuvre coûteuse (duplication de la pile pour chaque nouvelle continuation).

Représentation des données et GC

En Scheme chaque "type" possède un prédicat vérifiant si une valeur est de ce type.

```
(number? 314)

;Value: #T
```

⇒ une information de type dans l'objet, ou des zones d'allocation des objets d'un certain type.

Application

vérification d'arité (Scheme)
application partielle (Caml)
fermeture (idem en Scheme et Caml)

⇒ langage intermédiaire (structures de contrôle et données) commun pour la compilation de Scheme et ML.

Interprète

En Scheme : tous les programmes syntaxiquement corrects sont interprétables et compilables.

```
(car '1)
```

est interprétable et compilable.

En Caml il n'y a pas d'interprète (boucle interactive des compilateurs).

Compilateur

En Scheme le typage est vu comme une optimisation de compilation. Le but est d'enlever le maximum de tests de type (enlever une partie du code utilisateur).

En Caml, l'information de typage peut être perdue à l'exécution. Le typage est vu comme une garantie d'éviter une erreur de typage à l'exécution. Interessant aussi pour les preuves de programmes.

- Noyau Fonctionnel (à la curryfication près)
- liaison lexicale
- variable RW ou RO/RW : pb uniquement pour les valeurs immédiates
- listes (vecteurs) homogènes et n-uplets/records
- tag/exit (try with) (call_cc : pb de typage (comme les références) et mise en œuvre peu efficace (séquentiel))
- filtrage linéaire/ non linéraire
- kit de mise en œuvre : contrainte sur le GC

Grande discussion : typage fort ou souple ????

```
(define (lastn i l)
  (if (pair? l)   (let  ((r (lastn i (cdr l))) )
    (cond
      ((not (integer? r)) r)
      ((= r i) l)
      (#t (+ r 1)) ))
   0))
```

```
int -> list -> {int | list}
```

```
type 'a int_or_alpha_list = I of int | L of 'a list;;
```

```
Type int_or_alpha_list defined.
```

```
let lastn i l =
  let rec lastn_aux  l =
  match l with
    [] -> I 0
  | _::t ->
    let r = lastn_aux  t in
     (match r with
        L al -> r
      | I ir -> if ir =i then( L l )
                else (I (ir+1)))
  in  match (lastn_aux l) with L al -> al
                             | _ -> [];;
```

```
lastn : int -> 'a list -> 'a list = <fun>
```