Introduction à CAML

Solange Coupet Grimal

Cours Logique, Déduction, Programmation
2002–2004

1 CAML : Un langage fonctionnel

1.1 Programmation "impérative" ou "procedurale"

La notion d'état y est essentielle. Un état représente l'ensemble des variables d'un programme. L'exécution d'un programme consiste, a partir d'un état initial, a exécuter une suite finie de commandes d'affectation dont chacune modifie l'état courant. Elles sont exécutées séquentiellement et peuvent être itérées par des boucles "while" et exécutées conditionnellement a l'aide de "if ... then ... else".
Ce style de programmation a été conçu par un procède d'abstraction a partir de l'architecture des machines, depuis le langage machine, les assembleurs, les macro-assembleurs, FORTRAN, Pascal, C, etc ...

1.2 Programmation "fonctionnelle"

Au lieu d'une approche "par le bas", qui consiste a partir de l'architecture de l'ordinateur et a abstraire vers un langage, il s'agit ici d'une approche "par le haut" dans laquelle le langage est conçu comme un outil pour spécifier les algorithmes.
Il n'y a plus de notion d'état, plus d'affectation, plus de commande, plus de boucle. Un programme est une expression, exécuter un programme signifie évaluer l'expression. Plus précisément, un programme est l'expression de la fonction transformant l'entrée du programme en sa sortie.
Par exemple, la fonction C

```
int f (int n)
{
  int s;
  s = n*n;
  return(s);
};
```

s'écrit simplement :

```
fun x-> x*x;;
```

De plus, dans un langage fonctionnel, il n'y a pas de différence de nature entre une fonction et un objet simple comme un entier par exemple. Une fonction est une expression comme une autre et a ce titre pourra elle-même être l'argument ou le résultat d'une autre fonction (ordre supérieur).
La notion de boucle est remplacée par la récursivité.
Ce cours s'appuie sur le langage Caml, version 0.74, développe et distribue par l'INRIA.

2 Syntaxe et sémantique

Syntaxe concrète : les caractères ASCII composant l'expression.
Syntaxe abstraite : la structure profonde.
Type : représente un ensemble de valeurs.
Sémantique dynamique : expression —-¿ valeur Sémantique statique : expression —-¿ type
Adéquation entre les 2 : le type de l'expression est le môme que celui de sa valeur.

3 Expressions

3.1 Expressions

Évaluation :

```
<nom>:<type>=<valeur>

#1+2;;
- : int = 3

#4/(2*2);;
- : int = 1

#fun x->x*x;;
- : int -> int = <fun>
```

L'évaluation se fait en 3 temps :
– analyse syntaxique avec production de l'arbre syntaxique si l'expression est syntaxiquement correcte,
– évaluation statique avec la production du type si l'expression est typable,
– évaluation dynamique avec production de la valeur si le programme ne boucle pas.

3.2 Définitions : "let"

Correspond a l'expression "soit x le nombre ..."

```
#let x = 2*5+1;;
x : int = 11

#let pi = 3.1416;;
pi : float = 3.1416

#let square = fun x ->x*x;;
square : int -> int = <fun>
```

3.3 Définitions locales : "let ... in ..."

La portée d'une telle définition est limitée a ce qui suit :

```
#let x= 2*5+1;;
x : int = 11

#let x= 2 in x*x;;
- : int = 4

#x;;
- : int = 11

#let a = 1 and b = 2 in 2*a+b;;
- : int = 4

#let a=1 in let b=2*a in b+a;;
- : int = 3

#let a=1 and b=2*a in b+2+a;;
Toplevel input:
>let a=1 and b=2*a in b+2+a;;
>    ^
The value identifier a is unbound.
```

3.4 Autre syntaxe : "... where ..."

```
#x*x where x=2;;
- : int = 4

# 2*a+b where a=1 and b=2;;
- : int = 4

#b where b=2*a where a=1;;
```

```
- : int = 2

#a+b where b=2*a where a=1;;
Toplevel input:
>a+b where b=2*a where a=1;;
>^
The value identifier a is unbound.

#(a+b where b=2*a) where a=1;;
- : int = 3
```

4 Types

4.1 int et float

int : + - / * mod
float : +. -. *. /. sqrt cos sin tan acos asin atan log exp

```
#3.45;;
- : float = 3.45
#3.45 E 10;;
Toplevel input:
>3.45 E 10;;
>^^^^
This expression is not a function, it cannot be applied.
#3.45E10;;
- : float = 34500000000.0
#3.45e10;;
- : float = 34500000000.0

#int_of_float 1.;;
- : int = 1
#int_of_float 1.1;;
- : int = 1
#float_of_int 76;;
- : float = 76.0

#1.1+2.2;;
Toplevel input:
>1.1+2.2;;
>^^^
This expression has type float,
but is used with type int.
#1.1 +. 2.2;;
- : float = 3.3
```

Mode flottant :

```
##open float;;
Toplevel input:
>#open float;;
>      ^^^^^
Syntax error

##open "float";;
#1.1 + 2.2;;
- : float = 3.3

# 1+2;;
Toplevel input:
> 1+2;;
> ^
This expression has type int, but is used with type float.

##close "float";;
```

4.2 bool

Opérateurs : `true, false, &, or, not` Construction d'expressions booléennes : `<, <=, >=, =, if..then...else`

```
#1<2;;
- : bool = true

#1<=2;;
- : bool = true

#0<1 & 3<9;;
- : bool = true

#true & false;;
- : bool = false
```

Une expression "if" est de la forme :

```
--------------
if <expr bool>
then
        e1
else
        e2;;
--------------
```

Pour que cette expression soit typable, e1 et e2 doivent être de même type. Son type est alors le type de e1 et e2. Lors de l'évaluation dynamique, seule une des 2 expressions e1 ou e2 est évaluée, selon la valeur de l'expression booléenne.

```
#let x=3 in if x=0 then 0 else 7/x;;
- : int = 2

#if true then 1 else 0.;;
Toplevel input:
>if true then 1 else 0.;;
>                      ^^
This expression has type float,
but is used with type int.
```

4.3 String

Entre : `" "` Opérateur : `^`

```
#"ceci est une"^" chaine de caracteres";;
- : string = "ceci est une chaine de caracteres"

#"ceci est une"  ^  "chaine de caracteres";;
- : string = "ceci est unechaine de caracteres"
```

4.4 Char

Entre : ‘ ‘ Conversion ASCII : `int_of_char` Réciproque : `char_of_int`

```
#'a';;
Toplevel input:
>'a';;
>^
Syntax error.

#"a";;
- : string = "a"

#‘a‘;;
- : char = ‘a‘

#int_of_char ‘a‘;;
- : int = 97

#char_of_int 97;;
- : char = ‘a‘
```

4.5 Produit cartésien

```
#1,2,3;;
- : int * int * int = 1, 2, 3

#1,2;;
- : int * int = 1, 2

#1, (2,3);;
- : int * (int * int) = 1, (2, 3)

#("COUCOU" , 3.1, (‘A‘,2));;
- : string * float * (char * int) = "COUCOU", 3.1, (‘A‘, 2)

#(TRUE & FALSE, 3+3);;
Toplevel input:
>(TRUE & FALSE, 3+3);;
> ^^^^
The value identifier TRUE is unbound.

#true & false, 3+3;;
- : bool * int = false, 6

#fst;;
- : 'a * 'b -> 'a = <fun>

#snd;;
- : 'a * 'b -> 'b = <fun>

#let x=(1,"coucou") and y=("hello",2.1) in (snd x, fst y);;
- : string * string = "coucou", "hello"
```

5 Expressions fonctionnelles

5.1 Application d'une fonction a un élément

```
#(fun x -> x*x) 4;;
- : int = 16

#square 4;;
- : int = 16
```

L'opérateur d'application est note simplement par un nombre ¿0 d'espaces. C'est l'opérateur de plus forte priorité.

```
#square 2+3;;
- : int = 7
```

```
#square (2+3);;
- : int = 25
```

5.2 Ordre supérieur

5.2.1 Curryfication

Toute fonction Caml est une fonction a un seul argument. Considérons la fonction mathématique :

$$\begin{aligned} f : ZxZ &\longrightarrow Z \\ (x, y) &\longmapsto f(x, y) = x * y \end{aligned}$$

Cette fonction a 2 arguments peut se représenter a l'aide de la fonction unaire F définie par :

$$\begin{aligned} F : Z &\longrightarrow (Z \Rightarrow Z) \\ x &\longmapsto F(x) = fx \end{aligned}$$

ou la fonction fx est définie par

$$\begin{aligned} fx : Z &\longrightarrow Z \\ y &\longmapsto fx(y) = x * y \end{aligned}$$

La fonction F est une fonction d'ordre supérieur puisque pour tout x, F(x) est une fonction. F définit en fait une famille de fonctions (fx), indexée sur Z. Elle est appelée "curryfiee" de f (du nom du mathématicien Curry).

```
#let F = fun x -> fun y-> x*y;;
F : int -> int -> int = <fun>
```

Autres écritures strictement équivalentes a la précédente :

```
#let F = fun x y -> x*y;;
F : int -> int -> int = <fun>

#let F x = fun y -> x*y;;
F : int -> int -> int = <fun>

#let F x y = x*y;;
F : int -> int -> int = <fun>
```

Le type `int -> int -> int` est par defaut parenthèse de droite a gauche :

```
int -> (int -> int)
```

Applications successives de F a 2 arguments :

```
#F 2 34;;
- : int = 68
```

Dans les applications successives le parenthesage se fait par defaut de gauche a droite. L'expression calculée ci-dessus est en fait ainsi parenthésée : `((F 2)34)`. On peut utiliser F pour définir par exemple la fonction "triple".

```
#let triple = F 3;;
triple : int -> int = <fun>

#triple 21;;
- : int = 63
```

Remarque. La curryfication introduit une dissymétrie dans les arguments. On aurait pu curryfier de la façon suivante :

$$\begin{aligned} G : Z &\longrightarrow (Z \Rightarrow Z) \\ y &\longmapsto (x \longmapsto f(x,y)) \end{aligned}$$

5.2.2 Fonctions en argument

Nous venons de voir que la fonction F renvoie une valeur fonctionnelle. De façon analogue une expression fonctionnelle peut être l'argument d'une fonction. Il est ainsi possible de définir la fonction "comp" qui prend en arguments 2 fonctions et qui renvoie leur composée.

```
#let comp f g x = f (g x);;
comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

#comp square triple 4;;
- : int = 144

#comp square;;
- : ('_a -> int) -> '_a -> int = <fun>

#let f = comp square int_of_float;;
f : float -> int = <fun>

# f 4.1;;
- : int = 16
```

5.3 Eta-réduction

La fonction "triple" aurait pu aussi être ainsi définie :

```
#let triple = fun x -> F 3 x;;
triple : int -> int = <fun>
```

Les 2 formes sont équivalentes. En fait, si f est l'expression d'une fonction,

`(fun x -> f x)` et `f`

sont des expressions équivalentes. On dit que la seconde est obtenue a partir de la première par eta-réduction.

6 Opérateurs infixes

`&` est un opérateur infixe. L'opérateur préfixe correspondant est note prefix `&`.

```
#prefix &;;
- : bool -> bool -> bool = <fun>

#prefix & true false;;
- : bool = false

#prefix & true;;
- : bool -> bool = <fun>


#& true false;;
Toplevel input:
>& true false;;
>^
Syntax error.


#prefix <;;
- : 'a -> 'a -> bool = <fun>
```

L'utilisateur peut définir ses propres opérateurs infixes par la commande `#infix "..."`.
Exemple.

```
#let o = fun f g x->f(g(x));;
o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

l'opérateur o est préfixe.

```
#o sq sq 2.;;
- : float = 16.0

##infix "o";;
```

A partir de ce moment la, il devient infixe et ne peut plus être utilise sous forme préfixe :

```
#(sq o sq) 2.;;
- : float = 16.0
```

```
#sq o sq 2.;;
Toplevel input:
>sq o sq 2.;;
>     ^^^^^
This expression has type float, but is used with type 'a -> float.
```

car l'application a la plus forte priorité.

```
#o sq sq 2.;;
Toplevel input:
>o sq sq 2.;;
>^
Syntax error
```

car il n'est plus préfixe.

```
##uninfix "o";;
```

Il redevient préfixe :

```
#o sq sq 2.;;
- : float = 16.0
```

7 Fonctions récursives

Dans une définition

```
let x = e
```

les variables libres figurant dans e doivent avoir été définies précédemment (ou localement). Si x lui-même apparait dans e, x doit avoir été défini auparavant. Il n'y a plus aucun lien ensuite entre le nouvel x et l'ancien qui est écrase mais utilise pour calculer la nouvelle valeur.

```
#let x =1;;
x : int = 1

#let x = x+2;;
x : int = 3

#let x = let x=1 in x+2;;
x : int = 3

#let x = x+2 where x=1;;
x : int = 3
```

Les définitions récursives sont introduites par "let rec".

```
#let rec fact n = if n=0 then 1 else n*fact(n-1);;
fact : int -> int = <fun>

#(fact 4);;
- : int = 24
```

Attention !!

```
#let rec fact n= if n=0 then 1 else n*(fact n-1);;
fact : int -> int = <fun>

#(fact 4);;
Uncaught exception: Out_of_memory
```

car (f n-1) est en fait (f n)-1.

```
#let rec f = (fun n-> (if n=0 then 1 else n*f(n-1)));;
f : int -> int = <fun>

#(f 4);;
- : int = 24
```

Exemple. Définissons une fonction puissance nième (ou n est entier) d'un flottant x
– classiquement
– a l'égyptienne

7.0.1 Puissance classique

```
#let rec puiss n x= if n=0 then 1. else x*.(puiss (n-1) x);;
puiss : int -> float -> float = <fun>

#puiss 2 3.;;
- : float = 9.0

#(puiss 200000 1.0);;
Uncaught exception: Out_of_memory
```

7.0.2 Puissance égyptienne

```
#let rec puiss n x = if n=0 then 1.
                          else
                             let p =(puiss (n/2) x) in
                               if (n mod 2)=0 then
                                  p*.p
                               else
                                  p*.p*.x;;
```

```
puiss : int -> float -> float = <fun>

#puiss 10 2.;;
- : float = 1024.0

#puiss 2000000 1.0;;
- : float = 1.0

#puiss 20000000000 1.;;
- : float = 1.0
```

Résultats immédiats!!!
Mauvaise solution :

```
#let rec puiss_bad n x =
        if n=0 then 1.
        else
         if (n mod 2)=0 then
            (puiss_bad (n/2) x)*.(puiss_bad (n/2) x)
         else x*.(puiss_bad ((n-1)/2) x)*.(puiss_bad ((n-1)/2) x);;

#puiss_bad 20000000 1.;;
- : float = 1.0
```

6 minutes sur protis!!!

7.1 Fonctions mutuellement récursives

2 types de récursion mutuelle : soit les fonctions s'appelant mutuellement sont déclarées de façon globale. C'est le cas dans l'exemple suivant :

```
#let rec even n = if n=0 then true
                         else odd(n-1)
    and
        odd n = if n=0 then false
                        else even(n-1);;
even : int -> bool = <fun>
odd : int -> bool = <fun>
```

Soit l'une d'entre elle peut être déclarée localement :

```
#let rec even n = if n=0 then true
                         else odd(n-1)
    where rec
        odd n = if n=0 then false
                        else even(n-1);;
even : int -> bool = <fun>
```