

Programação Declarativa

Paradigmas de Programação Avançados

Programação Funcional

*Functional
Programming*

Tipos no OCaml

Tipos

Numa linguagem de programação, um **tipo** denota um **conjunto de valores**.

O Caml inclui tipos **base** e tipos **compostos**.

Tipos base:

- int
- float
- bool
- char
- string

Literais têm sempre um tipo, e os tipos base correspondem a literais

Tipos compostos

O Caml tem **construtores de tipos** que podem ser usados para gerar tipos, partindo dos tipos base.

- tuplo: denota-se um tipo $t_1 * t_2$ e um literal (v_1, v_2)

Tipos soma

Podemos “agrupar” várias formas sob um nome - trata-se de designar múltiplas variantes duma mesma coisa.

Por exemplo, supondo que queremos trabalhar com formas geométricas (quadrados, círculos, pontos, segmentos, ...), podemos:

1. Definir variáveis para cada grandeza relevante e agrupá-las p/ex em tuplos.

Pode funcionar, mas é algo conturbado e opaco (convenções não materializadas no programa)

2. Usar o construtor de tipo “soma”, que agrupa vários valores de tipos diferentes, em alternativa uns aos outros...

Tipos soma

Podemos fazer algo assim:

```
type forma = Quadrado of float  
            | Circulo of float  
            | Rectangulo of (float*float)  
            | Ponto  
            | Segmento of (float*float);;
```

Isto define **valores** que serão sempre do tipo **forma**.

Os valores começam sempre pele **seletor** seguido do valor do tipo associado.

É como um valor com uma etiqueta...

Tipos soma

Por exemplo:

```
# Circulo 10.0;;  
- : forma = Circulo 10.0  
# Ponto;;  
- : forma = Ponto  
# Rectangulo (10.0, 20.0);;  
- : forma = Rectangulo (10.0, 20.0)  
#
```

Tipos soma

Podemos fazer funções com vários casos “indexados” às alternativas dum tipo:

```
# let area = function
  Circulo x -> 3.14 *. x *. x
| Rectangulo (a, b) -> a *. b
| Quadrado q -> q *. q
| Ponto -> 0.0
| Segmento _ -> 0.0;;
val area : forma -> float = <fun>
```

Que podemos usar, por exemplo:

```
# area (Quadrado 2.0);;
- : float = 4.
# area (Circulo 1.0);;
- : float = 3.14
# area (Rectangulo (1.0,2.0));;
- : float = 2.
# area (Ponto);;
- : float = 0.
# area (Segmento (1.1,2.2));;
- : float = 0.
#
```


Tipos polimórficos

Um tipo pode incluir um **parâmetro**, o que faz dele um tipo polimórfico, i.e. que pode assumir várias formas.

Exemplo para uma lista de inteiros:

```
# type il = Nada | Par of (int*il);;  
type il = Nada | Par of (int * il)  
# Nada;;  
- : il = Nada  
# Par (10, Par (20, Nada));;  
- : il = Par (10, Par (20, Nada))  
# (10, (20, Nada));;  
- : int * (int * il) = (10, (20, Nada))  
#
```

Podemos generalizar, sobre o “**int**”, tornando-o um parâmetro:

Tipos polimórficos

O mesmo exemplo (lista de “coisas”):

```
# type 'a il = Nada | Par of ('a*'a il);;  
type 'a il = Nada | Par of ('a * 'a il)  
# Nada;;  
- : 'a il = Nada  
# Par (10, Nada);;  
- : int il = Par (10, Nada)  
# Par ("xpto", Nada);;  
- : string il = Par ("xpto", Nada)  
#
```

Listas e Arrays

Um tipo lista é dado por:

```
# [1; 2; 3];;  
- : int list = [1; 2; 3]
```

E partilha das propriedades das listas, i.e. é composto por pares ou uma lista vazia.

Um array denota-se pelo construtor de tipo “array” e os seus literais são delimitados por [| e |]:

```
# [| 1; 2; 3 |];;  
- : int array = [|1; 2; 3|]
```

E, para lhe aceder podemos usar o operador .(), por exemplo:

```
# let a = [| 10; 20; 30 |];;  
val a : int array = [|10; 20; 30|]  
# a .( 2 );;  
- : int = 30
```

Note-se que a indexação é a partir de zero

Tipos funcionais

Uma função é denotada por um tipo com o construtor “ \rightarrow ”.

- $T1 \rightarrow T2$

Em que **T1** e **T2** são tipos arbitrários, entende-se que **T1** é o tipo do argumento da função e **T2** o do resultado.

Os literais são realizados com o operador **function** ou com a sintaxe aumentada da declaração **let**.

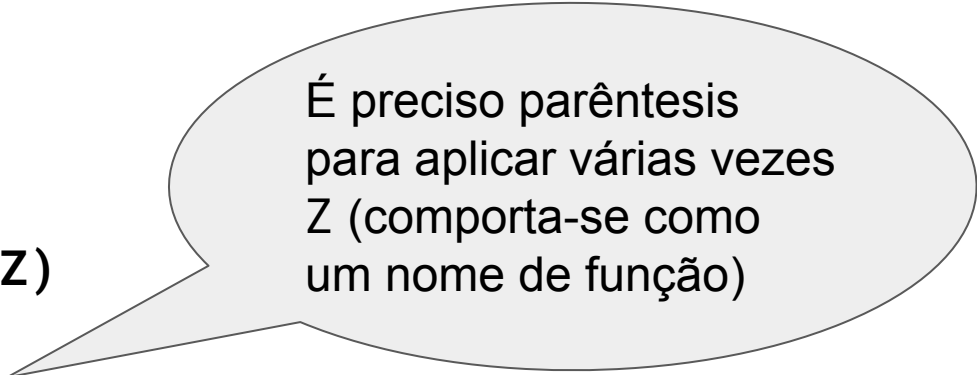
Exemplos:

```
# type ff = int -> int;;
type ff = int -> int
# function x -> x+1;;
- : int -> int = <fun>
# let inc : ff = function x -> x+1;;
val inc : ff = <fun>
# inc 3;;
- : int = 4
#
```

Numerais revisitados

Podemos fazer em Caml o mesmo que fazemos em Prolog:

```
# type numeral = Z | S of numeral;;  
type numeral = Z | S of numeral  
# Z;;  
- : numeral = Z  
# S Z;;  
- : numeral = S Z  
# S (S Z);;  
- : numeral = S (S Z)  
# S S Z;;  
Error: Syntax error  
# S (S (S Z));;  
- : numeral = S (S (S Z))  
#
```



É preciso parêntesis
para aplicar várias vezes
Z (comporta-se como
um nome de função)

Expressões de controle - match

Para avaliar uma expressão diferente consoante casos do valor doutra expressão, podemos usar a expressão “**match...with**”:

```
match expr with
  pat1 -> expr1
  | ...
  | patn -> exprn
```

Por exemplo

```
# let a = [1;2] in match a with [] -> 0 | x::_ -> x;;
- : int = 1
```

Note-se que isto é só “acúcar sintático” para:

```
# let a = [1;2] in (function [] -> 0 | x::_ -> x) a;;
- : int = 1
```

Expressões de controle - **while, for**

Para avaliar uma expressão repetidamente, enquanto outra expressão, for verdade, podemos usar o “**while...do...done**”:

```
while expr1 do expr2 done
```

Também podemos usar a instrução “for”:

```
for nome = expr1 to expr2 do expr3 done
```

Ou:

```
for nome = expr1 to expr2 downto expr3 done
```

Atenção que estas expressões fazem sentido se houver efeitos secundários (input/output ou afetação)

Exemplo de ciclo com I/O:

```
# for a = 1 to 10 do print_int a; print_string "\n"; done;;  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
- : unit = ()  
#
```