Introdução ao Ocaml

Alexsandro Santos Soares

Universidade Federal de Uberlândia
Faculdade de Computação

Ocaml

- Ocaml pertence à família das linguagens funcionais fortemente tipadas, com tipagem estática e inferência de tipos.
  - Essa família começou com a linguagem ML de Robin Milner.
- Também admite a programação imperativa e orientada a objetos.
- Criada em 1980 no INRIA da França, com a publicação da primeira implementação em 1987.

Características da linguagem

- tipagem estática forte: qualquer expressão da linguagem rigorosamente possui um itpo, verificado em tempo de compilação.
- inferência de tipos: o compilador sabe como calcular o tipo sem a necessidade de definição prévia.
- Casamento de padrões: é possível examinar a estrutura de tipo e usá-la.

Gestor de pacotes

Existe um conjunto abrangente de ferramentas de desenvolvimento:

- opam: um gestor de pacotes de software para a instalação de ferramentas e bibliotecas:
  - Ver `http://opam.ocaml.org`
  - Centenas de bibliotecas de ferramentas disponíveis em `http://opam.ocaml.org/packages`

Ferramentas de compilação

O Ocaml possui:

- um intérprete para o desenvolvimento rápido.
- um compilador para bytecode, para código portável.
- um compilador nativo, `ocamlopt`, para várias plataformas e que gera executáveis otimizados.
- um compilador para JavaScritp, `js_of_ocaml`, para criar aplicações web.

Instalação do OCaml via opam no Ubuntu

Instalação do opam e de suas dependências

```
sudo apt install wget
sudo apt install git aspcud m4 mercurial darcs
wget https://raw.github.com/ocaml/opam/master/shell/
  opam_installer.sh -O - | sh -s /usr/local/bin
opam init
eval `opam config env`
opam repository add git git+https://github.com/ocaml/
    opam-repository
opam update
```

Para saber qual é a última versão do OCaml disponível no repositório:

```
opam switch
```

No momento, a versão `4.07.0` é a mais autal e para instalá-la digite:

```
opam switch 4.07.0
eval `opam config env`
```

Primeiro experimento como o OCaml

Para chamar o intérprete com:

```
rlwrap ocaml
```

Podemos digitar e avaliar expressões sempre terminadas por ;; e seguidas por `enter`:

```
# print_string "Olá Mundo!\n";;
Olá Mundo!
- : unit = ()

# 1 + 4;;
- : int = 5

# List.map (fun x -> x + 1) [1; 2; 3; 4; 5];;
- : int list = [2; 3; 4; 5; 6]

# let x = 3 * 9;;
val x : int = 27
```

Primeiro experimento como o OCaml

```
# x;;
- : int = 27

# let rec fat n = if n < 1 then 1 else n * fat (n-1);;
val fat : int -> int = <fun>

# fat 5;;
- : int = 120

# #quit;;
```

- A diretiva `#quit` encerra a sessão.
- Comentários são da forma `(* ... *)`

Tipos básicos

Tipo	Função	Literais
`bool`	Valores booleanos	`true`, `false`
`int`	Números inteiros	`-34`
`float`	Números em ponto flutuante com precisão dupla	`-0.34e2`
`char`	caracteres	`'a'`
`string`	cadeias	`"abcd"`
`unit`	Tipo com valor único	`()`

Usando `_` é possível escrever números mais legíveis: `9_067_567`

Expressões

- As expressões são similares às de outras linguagens: `2+3`, `5 * (3 + 4)`.
- Os operadores básicos em ponto flutuante possuem um ponto: `+.`, `-.`, `*.` e `/.`

```
# 2. /. 3.;;
- : float = 0.6666666666666663

# 2.0 / 3.0;;
Error: This expression has type float but an expression was expected
  of type int

# 2 / 3.;;
Error: This expression has type float but an expression was expected
  of type int
```

- O operador `^` concatena strings:

```
# "mini" ^ "linguagem";;
- : string = "minilinguagem"
```

Definições

A expressão `let nome = exp` define um nome para uma expressão cujo escopo é o envolve todo o módulo:

```
# let x = 2;;
val x : int = 2

# let s = "abc";;
val s : string = "abc"

# x;;
- : int = 2

# s;;
- : string = "abc"

# y;;
Error: Unbound value y
```

Definições locais

`let nome = expr in exp2` define localmente um nome para uso na expressão `exp2`:

```
# let a = 7 in 3 + a;;
- : int = 10

# a;;
Error: Unbound value a

# let a = 1 in
    let b = 2 in
    let c = 3 in a + b + c;;
- : int = 6

# let a,b,c = 1,2,3 in a + b + c;;
- : int = 6
```

Funções

- Funções são valores de primeira classe em Ocaml.
- A expressão `function args -> corpo` serve para definir funções:

```
# function x -> 2 * x;;
- : int -> int = <fun>

# (function x -> 2 * x) 4;;
- : int = 8
```

- Podemos dar um nome à função definida:

```
# let dobro = function x -> 2 * x;;
val dobro : int -> int = <fun>

# dobro 4;;
- : int = 8

# let dobro2 x = 2 * x;;
val dobro2 : int -> int = <fun>

# dobro2 4;;
- : int = 8
```

Funções com definições locais

- Podemos usar `let` para criar expressões locais em uma função:

```
# let f a b =
    let x = a +. b in
    x +. x ** 2.;;
val f : float -> float -> float = <fun>
```

- Também podemos definir funções aninhadas em uma outra:

```
# let f a b =
      let quadrado z = z *. z in
      quadrado a +. quadrado b;;
val f : float -> float -> float = <fun>

# f 3.0 4.0;;
- : float = 25.
```

Tuplas

- Tuplas, também chamadas de tipos produtos, combinam um número fixo de outros tipos.
- Os elementos são separados por vírgula e a tupla pode ser envolvida por parênteses:

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

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

# 2, "dois" ;;
- : int * string = (2, "dois")
```

- Para pares, `fst` e `snd` extraem o primeiro e o segundo componentes:

```
# fst (2, "dois");;
- : int = 2

# snd (2, "dois");;
- : string = "dois"
```

Listas

- Os construtores para listas são `[]` para lista vazia e `::` para inserir um elemento na cabeça

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

- Uma forma abreviada de escrever uma lista:

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

- O módulo `List` contém funções para a manipulação de listas, tais como, `List.hd` para o primeiro elemento e `List.tl` para o resto:

```
# let ls = [1; 2; 3];;
val l : int list = [1; 2; 3]
# List.hd ls ;;
- : int = 1
# List.tl ls ;;
- : int list = [2; 3]
```

Tipos definidos pelo usuário

- Podemos definir tipos sinônimos usando `type`:

```
# type cadeia = string;;
type cadeia = string

# ("abc" : cadeia);;
- : cadeia = "abc"
```

- No tipo soma, cada valor possui um construtor diferente:

```
# type sinal = Verde | Amarelo | Vermelho;;
type sinal = Verde | Amarelo | Vermelho

# Verde;;
- : sinal = Verde
```

Tipos definidos pelo usuário

- Os construtores podem ter argumentos:

```
# type lista = Vazio | Cons of inteiro * lista ;;
type lista = Vazio | Cons of inteiro * lista

# Cons (1, Cons (2, Vazio));;
- : lista = Cons (1, Cons (2, Vazio))
```

- O nome de um construtor deve sempre começar com maiúscula.

Casamento de padrões

- Qualquer tipo em Ocaml pode ser analisado usando casamento de padrões.
- Um casamento de padrões possui a forma

```
match <exp> with p1 -> e1 | p2 -> e2 | ...
```

- Exemplo:

```
# let rec soma lista =
    match lista with
    Vazio -> 0
    | Cons (n, l) -> n + soma l;;
val soma : lista -> int = <fun>

# let ls = Cons (1, Cons (2, Vazio));;
val ls : lista = Cons (1, Cons (2, Vazio))

# soma ls;;
- : int = 3
```

Funções recursivas

- Usando `let rec` podemos definir uma função recursiva:

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

# fat 5;;
- : int = 120

# let rec tamanho xs =
  match xs with
  | [] -> 0
  | x::xs -> 1 + tamanho xs;;
val tamanho : 'a list -> int = <fun>

# tamanho [5; 6; 7];;
- : int = 3
```

- A função *tamanho* também poderia ser definida assim:

```
# let rec tamanho = function
    | [] -> 0
    | x::xs -> 1 + tamanho xs;;
```

Exceções

- Em Ocaml podemos definir novas exceções com `exception`:

```
# exception Numero_negativo;;
exception Numero_negativo

# Numero_negativo;;
- : exn = Numero_negativo
```

- Lançamos as exceções com `raise`:

```
      # let rec fat n =
      if n < 0
      then raise Numero_negativo
      else begin
        if n = 0 then 1 else n * fat (n - 1)
      end;;
val fat : int -> int = <fun>

# fat (-1);;
Exception: Numero_negativo.
```

Captura de exceções

- Capturamos exceções com `try ... with`

```
# try fat (−1) with Numero_negativo −> 0;;
− : int = 0

# try fat 5 with Numero_negativo −> 0;;
− : int = 120
```

Básico de entradas e saídas

- O Ocaml possui várias funções para leitura ou escrita de valores.
- Aqui estão algumas funções de leitura:

```
# read_int ;;
- : unit -> int = <fun>
# read_float ;;
- : unit -> float = <fun>
# read_line ;;
- : unit -> string = <fun>
```

- Abaixo encontram-se algumas funções de escrita:

```
# print_string ;;
- : string -> unit = <fun>
# print_newline ;;
- : unit -> unit = <fun>
# print_endline ;;
- : string -> unit = <fun>
```

```
# print_endline ;;
- : string -> unit = <fun>
# print_int ;;
- : int -> unit = <fun>
# print_float ;;
- : float -> unit = <fun>
```

Entrada e saída no estilo do C

- Existem duas bibliotecas do OCaml que disponibilizam funções de entrada e saída ao estilo da linguagem C, como o `printf` e o `scanf`.
- Algums exemplos:

```
# Printf.printf ;;
- : ('a, out_channel, unit) format -> 'a = <fun>

# Printf.eprintf ;;
- : ('a, out_channel, unit) format -> 'a = <fun>

# Scanf.scanf ;;
- : ('a, 'b, 'c, 'd) Scanf.scanner = <fun>

# Printf.printf "x = %d e y = %d\n" 1 2;;
x = 1 e y = 2
- : unit = ()
```

Abertura de módulos

- Para acessar as funções de um módulo devemos prefixá-la como o nome módulo, como fizemos com `printf`.
- Também podemos abrir um módulo e importar todos as suas funções e, nesse caso, não precisaremos prefixá-las como o nome do módulo.
- Fazeros isso usando a diretiva `open`:

```
# printf "Olá!\n";;
Error: Unbound value printf

# Printf.printf "Olá!\n";;
Olá!
- : unit = ()

# open Printf;;

# printf "Olá!\n";;
Olá!
- : unit = ()
```

Um intérprete de expressões em Ocaml

- Vamos escrever um pequeno intérprete para expressões e comandos em Ocaml.
- Vamos começar definindo o formato da árvore sintática para expressões e comandos.
- Depois definiremos um estado que permite ligar um identificador a um número.

Expressões aritméticas

```
type num = int;;

type ident = string;;

type expA = Num of num
                  | Ident of ident
                  | Soma of expA * expA
                  | Sub of expA * expA
                  | Mult of expA * expA;;
```

Expressões booleanas

```
type expB = Bool of bool
          | Eq of expA * expA
          | Menor of expA * expA
          | Not of expB
          | And of expB * expB
          | Or of expB * expB;;
```

Comandos

```
type local = ident;;

type comando = Nada
             | Atrib of local * expA
             | Seq of comando * comando
             | Cond of expB * comando * comando
             | While of expB * comando;;
```

Estado

```
(* o estado é uma função de identificadores para inteiros *)
type sigma = ident -> num;;

(* estado inicial *)
let vazio : sigma = function x -> 0;;

(* um estado modificado: sigma[l -> n] *)
let ligado f l n a = if a = l then n else f a;;
```

Alguns exemplos de uso

```
# ligado vazio "x" 3 "x";;
- : num = 3

# ligado (fun "y" -> 1) "x" 3 "x";;
- : int = 3

# ligado (fun "y" -> 1) "x" 3 "y";;
- : int = 1

# ligado (fun "y" -> 1) "y" 3 "y";;
- : int = 3
```

Avaliação de expressões aritméticas

```
let rec avalia_expA (e, sigma) =
   match e with
  | Num n -> n
  | Ident x -> sigma x
  | Soma (e1, e2) ->
      avalia_expA (e1, sigma) + avalia_expA (e2, sigma)
  | Sub (e1, e2) ->
      avalia_expA (e1, sigma) - avalia_expA (e2, sigma)
  | Mult (e1, e2) ->
      avalia_expA (e1, sigma) * avalia_expA (e2, sigma);;
```

Avaliação de expressões aritméticas

Vamos avaliar algumas expressões:

```
# avalia_expA (Num 2, vazio);;
- : num = 2

# avalia_expA (Ident "x", vazio);;
- : num = 0

# avalia_expA (Ident "x", ligado vazio "x" 3);;
- : num = 3
```

Avaliação de expressões booleanas

```
let rec avalia_expB (e, sigma) =
   match e with
  | Bool b -> b
  | Eq (e1, e2) ->
      avalia_expA (e1, sigma) = avalia_expA (e2, sigma)
  | Menor (e1, e2) ->
       avalia_expA (e1, sigma) < avalia_expA (e2, sigma)
  | Not e1 -> not (avalia_expB (e1, sigma))
  | And (e1, e2) ->
       avalia_expB (e1, sigma) && avalia_expB (e2, sigma)
  | Or (e1, e2) ->
       avalia_expB (e1, sigma) || avalia_expB(e2, sigma);;
```

Avaliação de expressões booleanas

Vamos avaliar algumas expressões:

```
# avalia_expB (Bool false, vazio) ;;
- : bool = false

# avalia_expB (Menor (Num 3, Num 4), vazio);;
- : bool = true

# avalia_expB (Menor (Num 5, Num 4), vazio);;
- : bool = false
```

Avaliação de comandos

```
let rec avalia_cmd (c, sigma) =
   match c with
   | Nada -> sigma
   | Atrib (i, e) -> ligado sigma i (avalia_expA (e, sigma))
   | Seq (c1, c2) -> avalia_cmd (c2, avalia_cmd (c1, sigma))
   | Cond (e, c1, c2) ->
        if (avalia_expB (e,sigma))
        then avalia_cmd (c1, sigma)
        else avalia_cmd (c2, sigma)
   | While (e, c1) ->
         if (avalia_expB (e, sigma))
        then let novo_sigma = avalia_cmd (c1, sigma) in
                  avalia_cmd (c, novo_sigma)
        else sigma
```

Programas de teste

```
let prog1 =
  (Seq (Atrib ("x", Soma (Num 1, Num 2)),
        Cond (Menor (Ident "x", Num 5),
              Atrib ("x", Num 10),
              Atrib ("x", Num 20))));;

# let sigma1 = avalia_cmd (prog1, vazio);;
val sigma1 : sigma = <fun>

# sigma1 "x";;
- : num = 10
```

A árvore acima equivale ao seguinte trecho em C:

```
x = 1 + 2;
if (x < 5)
   x = 10;
else
   x = 20;
```

Programas de teste

```
let prog2 =
  Seq (Atrib ("x", Soma (Num 1, Num 2)),
       While (Menor (Ident "x", Num 5),
              Atrib ("x", Soma (Ident "x", Num 1))));;

# let sigma1 = avalia_cmd (prog2, vazio);;
val sigma1 : sigma = <fun>

# sigma1 "x";;
- : num = 5
```

A árvore acima equivale ao seguinte trecho em C:

```
x = 1 + 2;
while (x < 5)
   x = x + 1;
```

Uso do compilador do Ocaml

- Vamos usar o compilador do OCaml.
- Para isso, coloque o conteúdo abaixo em um arquivo de nome `alo.ml`:

```
print_string "Alô mundo!\n"
```

- Para compilar este arquivo usamos

```
> ocamlc alo.ml -o alo.exe
```

- Esse comando cria um arquivo executável de nome `alo.exe` no mesmo diretório onde foi chamado.

```
> ./alo.exe
Alô mundo!
```

Uso do compilador do Ocaml

- Ao compilar um programa o OCaml gera dois arquivos binários, além do executável:
  - Um arquivo de interface contendo todas as funções e expressões exportadas e com extensão `.cmi`.
  - Um arquivo objeto contendo todas as funções e expressões compiladas e com extensão `.cmo`.
- Podemos gerar apenas compilar um programa, gerando os dois arquivos acima sem gerar o executável final:

```
> ocamlc -c alo.ml
```

- Depois, se quisermos, podemos juntar arquivos objeto no executável final:

```
> ocamlc alo.cmo -o alo.exe
```

Carregando código fonte ou compilado no intérprete

- Podemos digitar um programa em um arquivo e depois usar o intérprete para executá-lo, sem precisar compilá-lo:

```
> rlwrap ocaml
```

```
# #use "alo.ml";;
Alô mundo!
- : unit = ()
```

- Podemos também carregar um módulo que tenha sido previamente compilado:

```
# #load "alo.ml";;
Alô mundo!
```

Referência

- Em Ocaml uma variável que pode mudar de valor é uma referência.
- Declaração:

```
# let v = ref 0;;
val v : int ref = {contents = 0}
```

- Acesso:

```
# v;;
- : int ref = {contents = 0}

# !v;;
- : int = 0
```

- Alteração:

```
# v := !v + 4;;
- : unit = ()

# !v;;
- : int = 4
```

-
-

Referências

- Pereira, M. e Sousa, S. M. *Introdução à programação funcional em OCaml*. 2012. Disponível em `https://www.di.ubi.pt/~desousa/2015-2016/TC/intro_ocaml.pdf`