



universidade de aveiro

Real Time Systems 2011-2012

Objective
Categorical
Abstract
Machine
Language

Ricardo Lopes Almeida

Aveiro, 17/11/2011

What is OCaml?

- OCaml is the Objective (Object oriented) version of the CAML Language
- CAML is a ML language (*meta language*) that was developed around 1987 by a team of researchers at INRIA (*Institut National de Recherche en Informatique et en Automatique*), a public scientific french institution.

Meta-Language?

- Meta-language is a **functional** language, where functions are treated as first-class values. This means that a function can be nested, passed as a argument in an other function or even stored inside data structures.

How to use OCaml

- OCaml is distributed with a compiler, a *runtime* and a *toploop*.
- For simple programs or just to evaluate a expression, use the toplevel: type *ocaml* in a Linux terminal with OCaml packages installed to enter the prompt (#)
- To evaluate an expression use ';;' (double semicolon) at the end of it

How to use OCaml

- To exit the toplevel press CTRL-D or CTRL-Z.
- When writing complex programs, use a text editor and save the file with a .ml extension.
- To compile an .ml file, use the *ocamlc* command on a terminal.
- There are some IDE's available (Camelia).

How to use OCaml

- Insert comments between `(*` and `*)`, just like `/*` and `*/` in C/C++ and Java.
- Use the ***let*** keyword to attribute values in expressions or to setup functions:
- Example:
let x = 12;;
let add x y = x + y;;

OCaml Types

- Variable types used in OCaml:
- **int** (i = 1, x = 12)
- **float** (d = 1.7, r = 3.0, h = 5.)
- **char** (z = 'z', a = 'B', k = 'q')
- **string** (x = "Hello", y = "Ricardo")
- **bool** (true, false)
- **uint()** - the single element

OCaml Types

- `uint()` represents the unity element. Every function in OCaml must return at least one value. It works like a “default return value”, much like the **`void`** type in C/C++.
- In OCaml, a N bit integer uses $N - 1$ bits to set the value (in a two's complement representation). The remaining bit is used for garbage collection purposes.

OCaml basic operators

- `+`, `-`, `*`, `/`, `mod` → arithmetic operators (integers)
- `+.`, `-.`, `*.`, `/.`, `**` → arithmetic operators (floats)
- `Not`, `&&`, `||` → logic operators (bool)
- `=`, `<>`, `<`, `<=`, `>`, `>=` → logic operators
- NOTE: `mod` → remainder of int division
`**` → exponentiation (valid only on floats)
`<>` → different then

OCaml Main characteristics

- **Type inference** – The type of an variable, expression or result of a function is automatically determined at compile time. A function **may** be constructed without any type indication of the arguments or return value.
- Example: In an expression like $x +. y = z$, the compile infers that x , y and z are floats because the $+.$ operator is used only to add floating point values.

OCaml Main characteristics

- **Strongly typed** – At compile time, the type of every variable or expression is evaluated to determine if there are no type violations.
- Example: If in an expression tries to sum a int with a float or concatenate a string with a boolean, the compiler generates an error.

OCaml Main characteristics

- **Polymorphism** – It is possible to write certain types of generic functions which can be adapted to the argument type in run time.
- Some data types, like tuples, can have elements of different types.

OCaml Main characteristics

- Simple example: function identify

```
let identify x = x;;
```

```
val identify : 'a → 'a = <fun>
```

- The 'a symbol means that the function accepts any type of argument but it will return a value from the same type

Anatomy of a function

- Functions are one of the most important and useful aspects in OCaml
- A function is basically an attribution of a set of parameters (inputs) to an expression. The type of the output is determined by the compiler in an evaluation of the expression in compile time.

Anatomy of a function

- Function template:

```
let <function_name> = fun <input1> <input2>  
... <inputn> -> <expression>;
```

or more simply:

```
let <function_name> <input1> <input2> ...  
<inputn> = <expression>;
```

Anatomy of a function

- Example: **let** sum10 = **fun** x -> x + 10;;

val sum10 : int → int = <fun>

usage: sum10 5;;

- Example: **let** multiplyAll a b c d = a*b*c*d;;

val multiplyAll : int → int → int → int → int = <fun>

usage: multiplyAll 1 2 3 4;;

How to read the function output

- Every time a function is correctly written, the compiler return a summary of how the function must be used. This summary should be read as:

$(\text{type_input_1} \rightarrow \text{type_input_2} \rightarrow \dots \rightarrow \text{type_input_n}) \rightarrow \text{type_output}$

- Example: `let sft x y =`
`if x + y < 10 then "Bigger than ten" else "Lower than ten";;`
`val sft : int → int → string = <fun>`

If then else statement

- Like any other imperative languages, OCaml uses **if then else** statements to implement conditionality

Usage: **if** <boolean_condition> **then**
<result_if_true> **else** <result_if_false>

- In OCaml, the result can be a simple return value or another function call.

Lock function types

- Its possible to predefine the type of the arguments of a given function, if the definition makes her generic, using (argument:<type>) on the arguments
- Example: **let** compare x y =
if x < y **then** “Lower” **else** “Higher”;;
val compare : 'a → 'a → string = <fun>

Lock function types

```
let compare (x:float) (y:float) =  
  if x < y then "Lower" else "Higher";;  
val compare : float → float → string =  
<fun>
```

- In this case, the use of parenthesis to separate the arguments is imperative

Recursive functions

- Recursive functions are very common in OCaml, as one would expect from a functional language.
- To implement a recursive function it is necessary to use the **rec** keyword because the **let** declaration doesn't support recursive behavior.

Recursive functions

- Example: Implementing exponentiation using recursiveness

```
let rec power i x =
```

```
if i = 0 then 1.0
```

```
else x *. (power (i - 1) x);;
```

```
val power : int → float → float = <fun>
```

```
usage: power 3 10;;
```

Tuples

- Tuples are a basic data structure in OCaml. Unlike some other structures like lists or arrays, tuples are polymorphic, i.e, a single tuple can accommodate values of different types.
- Usage: A tuple must be defined between parenthesis and with commas separating its elements

Tuples

- Example: **let** tpe = (12, 'c', 3.4, true);;
*val tpe : int*char*float*bool = (12, 'c', 3.4, true)*
- Example: **let** coord x y = (x, y);;
*val coord : 'a → 'b → 'a * 'b = <fun>*
- Using a tuple creator function to define points in 2D space:
let p1 = coord 3 2;;

Lists

- Lists are data structures very similar to arrays on other languages.
- There are two ways of composing a list:
 - 1) Just like constructing a tuple but with [] instead of () and using semicolon instead of colon to separate the elements

Example: **let** nr = [2 ; 4 ; 8 ; 23];;

val nr : int list = [2 ; 4 ; 8; 23]

Lists

2) Using the constructor `::` between all of its elements. In this case, the end of the list must be the empty list constructor `[]`.

Example: **let** vowels = 'a' :: 'e' :: 'i' :: 'o' :: 'u' :: [];;

val vowels : char list = ['a' ; 'e' ; 'i' ; 'o' ; 'u']

- The constructor `::` acts also as a “deconstructor” when used on a existing list, inside a function.

Lists

- Example:

let rec sum = function

[] \rightarrow 0

| x :: l \rightarrow x + sum l

| _ \rightarrow raise (Invalid_argument “Error”);;

val sum : int list \rightarrow int = <fun>

The previous function uses the `::` operator to progressively disassemble a list in its individual components and add all of them in the end.

“Ready made” functions

- OCaml has quite a lot of “ready made” functions:

Example: `sqrt 16.0;;`

`cos 3.14;;`

`let` `lst = 1::4::6::9::[];;`

`List.hd lst;;`

`List.tl lst;;`

OCaml versus C

- Here's a simple example for the implementation of the Euclid's algorithm to determine the maximum common divider between two integers:

In C: **int** gcd (**int** a, **int** b)

```
{  
    while (a != b) {  
        if (a > b)  
            a -= b;  
        else  
            b -= a;  
    }  
    return a; }
```

- Example: gcd 10 6

10	6
$(10 - 6) = 4$	6
4	$(6 - 4) = 2$
$(4 - 2) = 2$	2
2	2

OCaml vs C

- In OCaml, although it possess while loops, the correct and optimized approach is through a recursive implementation:

```
let rec gcd a b =  
  if a = b then  
    a  
  else if a > b then  
    gcd (a - b) b  
  else  
    gcd a (b - a)
```

Conclusion

There are two quotes from some anonymous OCaml students that pretty much summarize, my experience with it:

*“It's hard to get it to compile,
but once it compiles, it works.” [1]*

Conclusion

*“Never have I spent
so much time
writing so little
that does so much.”[1]*

[1] – An Introduction to Objective Caml, Stephen A. Edwards, Columbia University, 2011

Bibliography

- *Introduction to Objective Caml*; Jason Hickey; 2008(<http://files.metaprl.org/doc/ocaml-book.pdf>)
- *Developing Applications with Objective Caml*; Emmanuel Chailloux, Pascal Manoury, Bruno Pagano; Éditions O'REILLY, 2000
- The Ocaml system – Documentation and User's manual; Xavier Leroy et al; 2011
- *An Introduction to Objective Caml*; Stephen A. Edwards; Columbia University; 2011
- <http://caml.inria.fr>