CS 320 : Functional Programming in Ocaml

based on slides by

David Walker (Princeton),
Lukasz Ziarek (Buffalo),
and *Mark Gardner* (BU)

Assaf Kfoury

MSC 118

kfoury@bu.edu

This class – CS 320

We will study different components of **programming languages** (PL's), such as:

- *design* issues of PL's and features they support (not all PL's are the same – why? – some are better for some purposes, no single PL is best for all tasks)
- *formal reasoning* about programmable functions
- how PL's are *compiled*
- *runtime environment* of compiled programs
- *functional programming* as a new paradigm

What is a Functional Language

A functional language:

- defines programs in a way similar to the way we define *mathematical functions*,
- avoids the use of *mutable states* (states that can change) in the execution of a program.

In *pure* functional languages, all information pertaining to the computation is *transparent*: there is *"no hidden information", "no side effects", "what you code/see is all what you get".*

Why study functional programming?

1. Learning a different style of programming teach you that programming transcends programming in a language
2. Functional languages predict the future
3. Functional languages are more and more used in industry
4. Functional languages help writing correct code

A good functional language part of the ML family.

- Small core language,
- Supports first-class higher order functions,
- Lexically scoped
- Statically strongly typed
- Type inference
- It has a good community: ocalm.org

TopHat Q1-Q6

Learning Goals for today

- Basic usage of Ocaml
- Expressions vs values
- Type checking basics
- Type Safety

A First OCaml Program

hello.ml:

```
print_string "Hello COS 326!!\n";;
```

A First OCaml Program

hello.ml:

```
print_string "Hello COS 326!!\n"
```


a function

its string argument enclosed in `"..."`

a program can be nothing more than just a single expression (but that is uncommon)

no parens. normally call a function f like this:

```
f arg
```

(parens are used for grouping, precedence only when necessary)

A First OCaml Program

hello.ml:

```
print_string "Hello COS 326!!\n"
```


compiling and running hello.ml:


```
$ ocamlbuild hello.d.byte
$ ./hello.d.byte
Hello COS 326!!
$
```

.d for debugging
(other choices .p for profiled; or none)

.byte for interpreted bytecode
(other choices .native for machine code)

A First OCaml Program

hello.ml:

```
print_string "Hello COS 326!!\n";;
```


interpreting and playing with hello.ml:

```
$ ocaml
        Objective Caml Version 3.12.0
#
```

A First OCaml Program

hello.ml:

```
print_string "Hello COS 326!!\n"
```


interpreting and playing with hello.ml:


```
$ ocaml
        Objective Caml Version 3.12.0
# 3 + 1;;
- : int = 4
#
```

A First OCaml Program

hello.ml:

```
print_string "Hello COS 326!!\n"
```


interpreting and playing with hello.ml:


```
$ ocaml
        Objective Caml Version 3.12.0
# 3 + 1;;
- : int = 4
# #use "hello.ml";;
hello cos326!!
- : unit = ()
#
```

A First OCaml Program

hello.ml:

```
print_string "Hello COS 326!!\n";;
```


interpreting and playing with hello.ml:


```
$ ocaml
        Objective Caml Version 3.12.0
# 3 + 1;;
- : int = 4
# #use "hello.ml";;
hello cos326!!
- : unit = ()
# #quit;;
$
```

Functions: so confusing!

- What is the difference between calling a function in these two ways:

  f arg1 arg2 f (arg1, arg2)

  The type of the first is:
  type_of_arg1 -> type_of_arg2 -> return_type_of_f
  The type of the second is:
  type_of_arg1 * type_of_arg2 -> return_
  For now just consider both as a way to
  is going on under the hood
  Which one you use depends on how you **define your function**!

Challenge question: How can both be true?

- Functions only take one argument (wait … but what is happening above?!?)

- Functions always return a value
  No return statement
  Body of a function is an **Expression** NOT a **sequence of statements**

How would you write a program summing the first n numbers?

A Second OCaml Program

sumTo8.ml:

a comment
(* ... *)

```
(* sum the numbers from 0 to n
   precondition: n must be a natural number
*)
let rec sumTo (n:int) : int =
  match n with
    0 -> 0
  | n -> n + sumTo (n-1)

let _ =
  print_int (sumTo 8);
  print_newline()
```

A Second OCaml Program

the name of the function being defined

sumTo8.ml:

```
(* sum the numbers from 0 to n
   precondition: n must be a natural number
*)
let rec sumTo (n:int) : int =
  match n with
     0 -> 0
   | n -> n + sumTo (n-1)

let _ =
  print_int (sumTo 8);
  print_newline()
```


the keyword “let” begins a definition; keyword “rec” indicates recursion

A Second OCaml Program

sumTo8.ml:

```
(* sum the numbers from 0 to n
   precondition: n must be a natural number
*)
let rec sumTo (n:int) : int =
  match n with
    0 -> 0
  | n -> n + sumTo (n-1)

let _ =
  print_int (sumTo 8);
  print_newline()
```


result type int

argument named n with type int

A Second OCaml Program

deconstruct the value n
using pattern matching

sumTo8.ml:

```
(* sum the numbers from 0 to n
   precondition: n must be a natural number
*)
let rec sumTo (n:int) : int =
  match n with
    0 -> 0
  | n -> n + sumTo (n-1)

let _ =
  print_int (sumTo 8);
  print_newline()
```

data to be deconstructed appears between key words "match" and "with"

A Second OCaml Program

vertical bar "|" separates the alternative patterns

sumTo8.ml:

```
(* sum the numbers from 0 to n
   precondition: n must be a natural number
*)
let rec sumTo (n:int) : int =
  match n with
    0 -> 0
  | n -> n + sumTo (n-1)

let _ =
  print_int (sumTo 8);
  print_newline()

_
```


deconstructed data matches one of 2 cases:

(i) the data matches the pattern 0, or (ii) the data matches the variable pattern n

A Second OCaml Program

Each branch of the match statement constructs a result

sumTo8.ml:

```
(* sum the numbers from 0 to n
   precondition: n must be a natural number
*)
let rec sumTo (n:int) : int =
  match n with
    0 -> 0
  | n -> n + sumTo (n-1)

let _ =
  print_int (sumTo 8);
  print_newline()
```

construct the result 0

construct a result using a recursive call to sumTo

A Second OCaml Program

sumTo8.ml:

```
(* sum the numbers from 0 to n
   precondition: n must be a natural number
*)
let rec sumTo (n:int) : int =
  match n with
    0 -> 0
  | n -> n + sumTo (n-1)

let _ =
  print_int (sumTo 8);
  print_newline()
```


print the result of calling sumTo on 8

print a new line

Expressions, Values, Simple Types

Terminology: Expressions, Values, Types

- Expressions : specify computations
  - 2 + 3 is a computation

- Values are the results of computations
  - 5 is a value

- Types describe collections of values and the computations that generate those values
  - int is a type

  - values of type int include
    - 0, 1, 2, 3, …, max_int
    - -1, -2, …, min_int

Some simple types, values, expressions

Type:	Values:	Expressions:
`int`	`-2, 0, 42`	`42 * (13 + 1)`
`float`	`3.14, -1., 2e12`	`(3.14 +. 12.0) *. 10e6`
`char`	`'a', 'b', '&'`	`int_of_char 'a'`
`string`	`"moo", "cow"`	`"moo" ^ "cow"`
`bool`	`true, false`	`if true then 3 else 4`
`unit`	`()`	`print_int 3`

For more primitive types and functions over them,
see the OCaml Reference Manual here:

http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html

Not every expression has a value

Expression:

```
42 * (13 + 1)          evaluates to  588
(3.14 +. 12.0) *. 10e6      ↦        151400000.
int_of_char 'a'             ↦        97
"moo" ^ "cow"               ↦        "moocow"
if true then 3 else 4       ↦        3
print_int 3                 ↦        ()

1 + "hello"    does not evaluate!
```

Core Expression Syntax

The simplest OCaml expressions e are:

- values — *numbers, strings, bools, ...*
- id — *variables (x, foo, ...)*
- e_1 op e_2 — *operators (x+3, ...)*
- id e_1 e_2 … e_n — *function call (foo 3 42)*
- **let** id = e_1 **in** e_2 — *local variable decl.*
- **if** e_1 **then** e_2 **else** e_3 — *a conditional*
- (e) — *a parenthesized expression*
- (e : t) — *an expression with its type*

A note on parentheses

In most languages, arguments are parenthesized & separated by commas:

```
f(x,y,z)        sum(3,4,5)
```

Both are valid in OCaml, but different

In OCaml, ~~we don't write the parentheses or the commas:~~


```
f x y z         sum 3 4 5
```


But we do have to worry about *grouping*. For example,


```
f x y z
f x (y z)
```

The first one passes three arguments to f (x, y, and z)

The second passes two arguments to f (x, and the result of applying the function y to z.)

TopHat Q7-Q9

Type Checking Basis

Type Checking

- Every value has a type and so does every expression
- We write (e : t) to say that expression e has type t. eg:

2 : int

"hello" : string

2 + 2 : int

"I say " ^ "hello" : string

Type Checking Rules

- You can always start up the OCaml interpreter to find out a type of a simple expression:

```
$ ocaml
        Objective Caml Version 3.12.0
#
```

Type Checking Rules

- You can always start up the OCaml interpreter to find out a type of a simple expression:

```
$ ocaml
        Objective Caml Version 3.12.0
# 3 + 1;;
```

Type Checking Rules

- You can always start up the OCaml interpreter to find out a type of a simple expression:

```
$ ocaml
        Objective Caml Version 3.12.0
# 3 + 1;;
- : int = 4
#
```

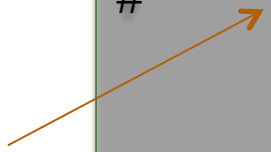

press return and you find out the type and the value

Type Checking Rules

- You can always start up the OCaml interpreter to find out a type of a simple expression:

```
$ ocaml
        Objective Caml Version 3.12.0
# 3 + 1;;
- : int = 4
# "hello " ^ "world";;
- : string = "hello world"
#
```


press return and you find out the type and the value



Type Checking Rules

- You can always start up the OCaml interpreter to find out a type of a simple expression:

```
$ ocaml
        Objective Caml Version 3.12.0
# 3 + 1;;
- : int = 4
# “hello ” ^ “world”;;
- : string = “hello world”
# #quit;;
$
```

Type Checking Rules

- There are a set of simple rules that govern type checking
  - programs that do not follow the rules will not type check and O'Caml will refuse to compile them for you (the nerve!)
  - at first you may find this to be a pain …

- But types are a great thing:
  - they *help us think* about *how to construct* our programs
  - they help us *find stupid programming errors*
  - they help us track down compatibility errors quickly when we edit and *maintain our code*
  - they allow us to *enforce powerful invariants* about our data structures

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "…")

Suppose we have an expression of the shape:

```
e1 + e2
```


How can we check if it has the type `int`?

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "...")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

Suppose we have an expression of the shape:

```
e1 ^ e2
```


How can we check if it has the type `string`?

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "...")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

(6) if e : int
then string_of_int e : string

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "...")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

(6) if e : int
then string_of_int e : string

- Using the rules:

2 : int and 3 : int. (By rule 1)

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "...")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

(6) if e : int
then string_of_int e : string

- Using the rules:

2 : int and 3 : int. (By rule 1)
Therefore, (2 + 3) : int (By rule 3)

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "...")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

(6) if e : int
then string_of_int e : string

- Using the rules:

2 : int and 3 : int. (By rule 1)
Therefore, (2 + 3) : int (By rule 3)
5 : int (By rule 1)

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for a ")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

tring_of_int e : string

FYI: This is a ***formal proof*** that the expression is well-typed!

- Using the rules:

2 : int and 3 : int. (By rule 1)
Therefore, (2 + 3) : int (By rule 3)
5 : int (By rule 1)
Therefore, (2 + 3) * 5 : int (By rule 4 and our previous work)