CSCI 2041: Introduction

Chris Kauffman

Last Updated:
Wed Sep 5 11:56:55 CDT 2018

Logistics

Reading

- OCaml System Manual: 1.1 - 1.3
- Practical OCaml: Ch 1-2

Goals

- History
- Tour of OCaml Language
- First Programs
- Course Mechanics

Origins of ML

- 1930s: Alonzo Church invents the **Lambda Calculus**, a notation to succinctly describe computable functions.
- 1958: John McCarthy and others create **Lisp**, a programming language modeled after the lambda calculus. Lisp is the second oldest programming language still widely used.
- 1972: Robin Milner and others at Edinburgh/Stanford develop the Logic For Computable Functions (LCF) Theorem Prover to do Math stuff
- To tell LCF how to go about its proofs, they invent a **Meta Language (ML)** which is like **Lisp with a type system** (Hindley-Milner type system)
- Folks soon realize that ML is a damn fine general purpose programming language and start doing things with it besides programming theorem provers
- 2007: Rich Hickey creates **Clojure**, a dialect of Lisp with a focus on concurrency and Java interoperability *CSCI 2041 is sometimes taught in Clojure*

Exercise: Origins of OCaml

- Circa 1990[1], Xavier Leroy at France's INRIA looks at the variety of ML implementations and declares "They suck"
  - No command line compiler: only top level Read-Eval-Print Loops (REPL)
  - Only run on Main Frames, not Personal Computers
  - Hard to experiment with adding new features
- Leroy develops the ZINC system for INRIA's flavor of ML: Categorical Abstract Machine Language (CAML)
  - `ocamlrun` Byte-code runtime interpreter
  - `ocamlc` Byte-code compiler
  - Allows separate compilation, linking, running
- Later work introduces
  - Object system: **Objective Caml**, shortened to OCaml
  - Native code compiler `ocamlopt`
  - Various other tools sweet tools like a **time traveling debugger**
- Question: Byte Code? Native Code? What common systems use them? What else is there?

[1] Xavier Leroy. The ZINC Experiment. Technical report 117, INRIA, 1990

Byte Code versus Native Code Compilation

Native Code Compilation

Convert source code to a form directly understandable by a CPU (an executable program)

Byte Code Compilation

Convert source code to an intermediate form (byte code) that is must be further converted to native code by an interpreter.

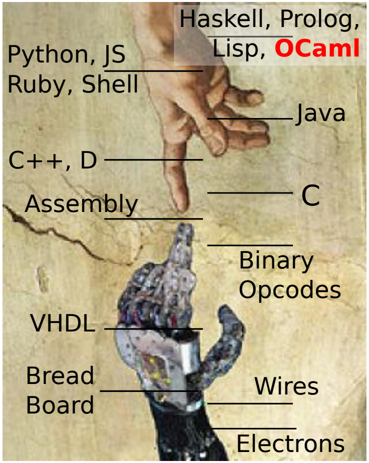
Source Code Interpreter

Directly execute source code as it is read by doing on-the-fly conversions to native code.

System	Compilation/Execution Model
Java	Byte Code Compilation: `javac` and `java`
C / C++	Native Code Compilation (mostly)
Python	Source Code Interpreter (with some byte code)
OCaml	Byte Code and Native Code Compilation
	And interactive REPL with on-the-fly compilation

Motivation to Use OCaml

Pure Abstraction



Bare Metal

Source

- Expressing programs at a high level is usually easier
  - Fewer lines of code
  - Closer to natural language statement of problem
  - Can effectively reason about program correctness
- OCaml translates high-level code to reasonably efficient low-level instruction

  OCaml delivers at least 50% of the performance of a decent C compiler –Xavier Leroy, on Caml Mailing List
- In **many** cases, trading runtime slow down to get development speedup is totally worth it

"If Programming Languages were Cars..."

Courtesy of Mike Vanier

- C is a racing car that goes incredibly fast but breaks down every fifty miles.
- C++ is a souped-up version of the C racing car with dozens of extra features that only breaks down every 250 miles, but when it does, nobody can figure out what went wrong.
- Java is a family station wagon. It's easy to drive, it's not too fast, and you can't hurt yourself.
- Python is a great beginner's car; you can drive it without a license. Unless you want to drive really fast or on really treacherous terrain, you may never need another car.
- **Ocaml** is a very sexy European car. It's not quite as fast as C, but it never breaks down, so you end up going further in less time. However, *because it's French, none of the controls are in the usual places.*

Aspects of OCaml that Set it Apart

A quick tour of ML features that make it **a joy** to program include

- Type Inference: compiler figures out types for you
- Type Checking: compiler checks functions called correctly
- Built-in Aggregate Types: Tuples, Linked Lists, Arrays
- Algebraic Types: Easy to create new types
- Pattern Matching: Easy to create code dealing with algebraic types
- First-Class Functions: Functions can be arguments to other functions
- Map-Reduce Functionality: Easy to work with aggregate data
- Module System: Enables separate, safe compilation with control over exported namespace

What follows is a quick tour of these features **to be discussed in more detail in coming weeks**

Tour: Type Inference

```
(* TYPE INFERENCE *)
> ocaml                                         (* start the REPL *)
      OCaml version 4.06.0

# let x = 7;;                                   (* bind x to 7 *)
val x : int = 7                                 (* x must be an integer *)
# let doubler i = 2*i;;                         (* bind doubler to a function *)
val doubler : int -> int = <fun>                (* int argument, int returns *)
(*            arg    return *)

(* TYPE CHECKING *)
# doubler 9;;                                   (* call doubler on 9 *)
- : int = 18                                    (* result is an integer *)
# doubler x;;                                   (* call on x *)
- : int = 14                                    (* ok - x is an integer *)
# doubler "hello";;                             (* call doubler "hello" *)
Characters 8-15:                                (* Type Checker says: *)
  doubler "hello";;                             (* NO SOUP FOR YOU *)
          ^^^^^^^

Error: This expression has type string but an
       expression was expected of type int
```

Tour: Built-in Aggregate Types

```
(* BUILT-IN AGGREGATE DATA TYPES *)
# let pair = (1.23, "hi there");;       (* tuple *)
val pair : float * string = (1.23, "hi there")

# let str_list = ["a"; "b"; "c"];;      (* linked list *)
val str_list : string list = ["a"; "b"; "c"]

# let float_arr = [|1.23; 4.56;|];;     (* array *)
val float_arr : float array = [|1.23; 4.56|]
```

Tour: Creating Types Types

```
(* Record Types: like C structs / Java classes *)
# type grade =                                 (* new record type *)
    {name : string; score : int; max : int };;
type grade = { name : string; score : int; max : int; }

# let a1 =                                     (* bind a1 to record value *)
    {name="Assgn 1"; score=86; max=100};;
val a1 : grade = {name = "Assgn 1"; score = 86; max = 100}

(* QUICK 'N EASY ALGEBRAIC TYPES *)
# type fruit =                                 (* create a new type *)
    Apple | Orange | Grapes of int;;           (* 3 value kinds possible *)
type fruit = Apple | Orange | Grapes of int

# let a = Apple;;                              (* bind a to Apple *)
val a : fruit = Apple
# let g = Grapes(7);;                          (* bind g to Grapes *)
val g : fruit = Grapes 7
```

Tour: Pattern Matching

```
# let a = Apple;;                       (* bind a to Apple *)
# let g = Grapes(7);;                   (* bind g to Grapes *)

(* PATTERN MATCHING *)
(* On Algebraic Types *)
# let count_fruit f =                   (* function of fruit *)
    match f with                        (* pattern match f *)
    | Apple ->  1                       (* case of Apple *)
    | Orange -> 1                       (* case of Orange *)
    | Grapes(n) -> n                    (* case of Grapes *)
;;
val count_fruit : fruit -> int = <fun>

# count_fruit a;;                       (* call on a = Apple *)
- : int = 1
# count_fruit g;;                       (* call on g = Grapes(7) *)
- : int = 7
```

Tour: First Class Functions and Map/Reduce

```
(* FIRST-CLASS FUNCTIONS + MAP-REDUCE PARADIGM *)
# let fruit_basket =                           (* Create a list of fruits *)
    [Apple; Apple; Grapes(2); Orange; Grapes(5);];;
val fruit_basket : fruit list = [...]

# let fruit_counts =                           (* Generate list by applying *)
    List.map count_fruit fruit_basket;; (* count_fruit to each el *)
val fruit_counts : int list = [1; 1; 2; 1; 5]

# let total =                                  (* apply + to each el of *)
    List.fold_left (+) 0 fruit_counts;; (* fruit_counts list *)
val total : int = 10
```

Influence of Functional Programming and ML

You may never use OCaml for a job, but you will definitely feel its effects elsewhere, particularly

- Functional Programming
- ML-inspired type systems

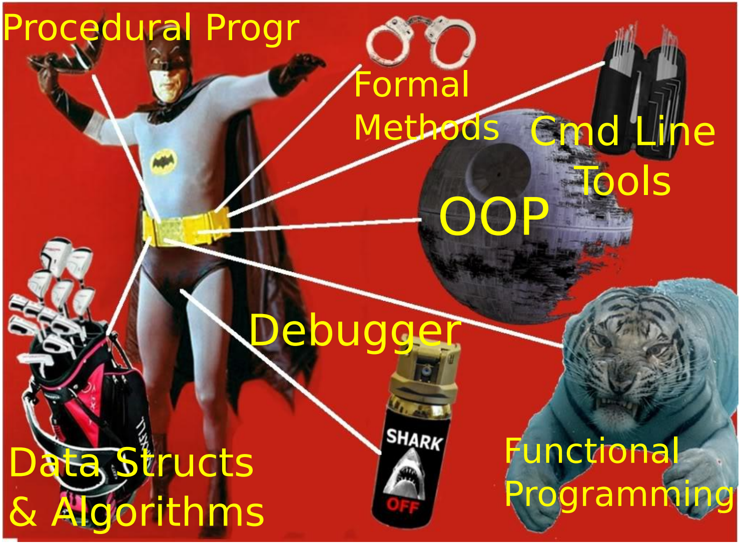
"Main Stream" programming languages have been strongly influenced by functional programming and ML dialects

- Java 8
  - Added anonymous functions to allow more functional programming like Map/Reduce
  - Generics system to allow type-safe containers with limited type inference
- F# (Microsoft) : OCaml + .NET framework
- Swift (Apple) : ML + Objective-C library access
- Scala : JVM language with type inference, algebraic data types, functional features, OO features, every lang feature known and unknown

Thus **OCaml is fun to program in and informative about other modern languages**

Programming Tools

Functional Programming (FP) is an integral part of any utility belt



OCaml Tools

OCaml comes with a fairly complete set of tools

- `ocamlc`: Byte Code Compiler & Interpreter
- `ocamlopt`: Native Code Compiler
- `ocaml`: Read-Eval-Print Loop (REPL)
- `ocamldebug`: Debugger (step code forwards or backwards (!))
- `ocamlprof`: Profiler to examine Performance
- Fairly Complete Standard Library including Hash Tables, Sets, Maps, Unix System Calls, and Threads (sort of)
- Documentation Generator

Interactive REPL versus Batch Compilation

REPL: Interactive Exploration

- Will sometimes demonstrate concepts in the REPL
- Type top-level statements directly or `#use` a file to load it
- Programs compiled on the fly and results displayed interactively
- GREAT for learning and exploration
- BAD for medium- to large-scale program development

Compiling Code

- Convert a source `.ml` file into an executable version (byte code or native code)
- Not interactive: compiler may find errors at compile time which cannot be resolved (syntax, types, missing modules)
- Can split program into many pieces and control their **interfaces**
- Typically several **library** files providing function and a **main** file which runs them

Exercise: Interactive REPL

Interactive session in `ocaml` REPL loading source files

```
> ocaml                                     (* start ocaml REPL *)
        OCaml version 4.06.0

# #use "basics.ml";;                        (* run basics.ml *)
val x : int = 15                            (* loads bindings for *)
val y : string = "hi there"                 (* x, y, repeat_print *)
val repeat_print :
  int -> string -> unit = <fun>

# repeat_print 2 y;;                        (* call repeat_print *)
hi there                                    (* interactively *)
hi there
- : unit = ()

# let msg = "adios";;                       (* bind msg interactively *)
val msg : string = "adios"

# repeat_print x msg;;                      (* call function again *)
adios                                       (* Q: How many times  *)
adios                                       (*    is msg printed? *)
...
```

Answers: Interactive REPL

```
# #use "basics.ml";;
val x : int = 15                                (* x has value 15 *)
...
# let msg = "adios";;

# repeat_print x msg;;                          (* call function again *)
adios                                           (* Q: How many times  *)
adios                                           (*    is msg printed? *)
adios
adios                                           (* 15 times *)
adios
adios
adios
adios
adios
adios
adios
adios
adios
adios
adios
- : unit = ()
```

Exercise: Batch Compilation

Batch compilation with `ocamlc` in a Unix shell

```
> ls                                # list files in the working directory
basics_main.ml  basics.ml           # two OCaml source files

> cat basics_main.ml                # show contents of basics_main.ml
(* main actions *)
Basics.repeat_print 4 Basics.y;;
Basics.repeat_print 2 "bye!";;

> ocamlc basics.ml basics_main.ml   # compile both files together

> ls                                # list files again, many more now
a.out       basics.cmo       basics_main.cmo  basics.ml
basics.cmi  basics_main.cmi  basics_main.ml

> file basics.cmo                   # what is basics.cmo?
basics.cmo: OCaml object file (.cmo) (Version 022)

> file a.out                        # what is a.out?
a.out: a /usr/bin/ocamlrun script executable (binary data)

> ./a.out                           # run a.out as an executable
hi there
...
# What is the remainder of the output?
```

Answers: Batch Compilation

```
> cat basics.ml                     # show contents of file
let x = 15;;
let y = "hi there";;                # Basics.y is "hi there"
...

> cat basics_main.ml                # show contents of file
(* main actions *)
Basics.repeat_print 4 Basics.y;;    # print "hi there" 4 times
Basics.repeat_print 2 "bye!";;      # print "bye" two times

> ./a.out
hi there                            # 4 times
hi there
hi there
hi there
bye!                                # 2 times
bye!
```

Byte-Code versus Native-Code Compilation

```
# BYTE CODE COMPILER : ocamlc
> ocamlc speedtest.ml    # compile to bytecode
> file a.out             # show file type
a.out: a /usr/bin/ocamlrun script executable (binary data)

> time ./a.out           # time execution
33554432

real  0m0.277s           # about a quarter second passed
user  0m0.276s           # full debug features available
sys   0m0.000s

# NATIVE CODE COMPILER: ocamlopt
> ocamlopt speedtest.ml  # compile to native code
> file a.out             # show file type
a.out: ELF 64-bit LSB pie executable x86-64

> time ./a.out
33554432

real  0m0.022s           # about 1/10th the time: WAY FASTER
user  0m0.022s           # BIG BUT: can't use native code with
sys   0m0.000s           #          OCaml's debugger
```

CSCI 2041: Course Goals

- Basic proficiency in a high-level programming language that facilitates functional programming (OCaml)
- An understanding of the advantages and disadvantages of types in programming languages
- An understanding of the importance of polymorphism in programming language types
- The ability to employ functions as first-class values in common higher-order computation patterns such as mapping, reducing, and filtering.
- An understanding of the advantages and disadvantages of side-effect free "pure" functions in computation versus computations based on explicit mutation and state change.
- A strong knowledge of how recursive functions perform and solve problems on data structures and search problems.
- Familiarity with program organization mechanisms such as modules, namespaces, private internals, classes, and nesting.