Types and Programming Languages

Lecture 2. Introduction to OCaml

Xiaojuan Cai

cxj@sjtu.edu.cn

BASICS Lab, Shanghai Jiao Tong University

Fall, 2016
Resources

- For quick start:
  https://try.ocamlpro.com/

- A concise introduction:
  http://www.csc.villanova.edu/~dmatusze/resources/ocaml/ocaml.html

- For Java/C/C++ programmer:
  http://ocaml.org/learn/tutorials/

- For MLers:
  http://www2.lib.uchicago.edu/keith/ocaml-class/class-01.html

- Official documentations:
  http://caml.inria.fr/pub/docs/manual-ocaml/

- OCaml install:
  http://ocaml.org

Tuareg mode: https://github.com/ocaml/tuareg
OCaml

- OCaml is one of the implementations of “Caml” language, which is a descendant of ML, *Meta Language*.
- **Paradigm**: Multi-paradigm (functional, OO and imperative)
- **Type system**: static, strong, inferred
- OCaml is very popular with researchers all over the world as a basis for experimental languages.
- **HelloWorld in OCaml**:
  ```ocaml```
  ```
  print_string "HelloWorld!\n";;
  ```
Outline

Fundamentals

Data types

Higher-order functions

OOP v.s. FP
Simple expressions

Expressions might be

- variables
- arithmetic expressions
- values
- conditions
- boolean expressions
- function calls
- ...

See our_first_program.ml.
Simple functions

- A function is a value! (No evaluation yet)
- Types of functions are called *arrow types*. $t_1 \to t_2 \to t_3 \to t_r$
- All the types are “magically” inferred out.

See `functions.ml`. 
Shadowing

Expressions in variable bindings are evaluated eagerly
- Before the variable binding finishes
- Afterwards, the expression producing the value is irrelevant

There is no way to assign to a variable in ML. Can only shadow it in a later environment.

See shadowing.ml
A function value has two parts
  ▶ The code (obviously)
  ▶ The environment when the function was defined

This pair is called a function closure – a very important concept in FP.

See closure.ml.
OCaml has mutations.

- `ref e` to create a reference with initial contents `e`
- `e1 := e2` to update contents
- `!e` to retrieve contents `s`
- New types: `t ref` where `t` is a type.

See `closure.ml`.
let and let...in.. expressions

- **let binding in e**, the scope of variables in binding is e
- **let binding**, the scope of variables in binding is the blocks afterwards

- let a = 1343*2344*5 + (f 1343*2344)
- let a = let b = 1343*2344 in b*5 + (f b)

Good style and more efficient

See let_efficiency.ml.
Outline

Fundamentals

Data types

Higher-order functions

OOP v.s. FP
Basic types

- **int.** e.g. 0, 5, 42, -17, 0x00FF, 0o77, 0b1101
  - +,-,*,/,% mod, abs
  - 31-bits, no unary +

- **float.** e.g. 0., -5.3, 1.7e14, 1.7e+14, 1e-10
  - +,-,*,/,,**,sqrt,ceil,floor,sin,cos, ...
  - Can't start with a decimal point.
  - coercions:
    float_of_int, float, string_of_int, int_of_string, ...

- **bool** contains two values: true, false
  - !,&&,||, with short-circuit

- **string.** e.g. "", "one\ntwo"
  - <,=,...,^,String.concat,String.length,...,
  - String is mutable! s.[i], s.[i]<-c

- **char.** e.g. 'a', '\n'

- unit only has one value (), like void in C.
Tuple and lists

- **Tuples**: fixed “number of pieces” that may have different types
  - Syntax: \( e_1, e_2, \ldots, e_n \), or \((e_1, e_2, \ldots, e_n)\)
  - Type: \( t_1 \times t_2 \times \ldots \times t_n \)
  - Built-in functions: \( \text{fst}, \text{snd} \)
  - Usage: multiple bindings, multiple return values

- **Lists**: any “number of pieces” that all have the same type
  - Syntax: \( [e_1; e_2; \ldots; e_n], [] \)
  - Type: \( t \text{ list} \)
  - Built-in functions:
    - \( ::, @, \text{List.length}, \text{List.hd}, \text{List.tl}, \text{List.nth}, \ldots \)
  - List is very important data type in functional PLs.
Functions over lists

- List is a recursive type.
- Functions over lists are always defined recursively.
- Pattern matching is heavily used in functional programs. It makes programs easy to write and read.

See lists_functions.ml.
Types in any language

- “Each of type”: A t value contains values of each of t1,...,tn
  Example: int * bool

- “Self reference”: A t value can refer to other t values
  Example: int list

- “One of type”: A t value contains values of one of t1,...,tn
  Example: ?

In let efficiency.ml, we return 0 for empty list []. We need some type to represent none or int.
Build your own “one of type”

type mytype = None | Int of int

- Adds a new type mytype to the environment
- Adds constructors to the environment: None, Int
- *Construct* the data of new types: tag + value
- *Access* the data of new types: pattern matching

See `type_bindings.ml`. 
Recursive types

code

```ocaml
type myintlist = Empty | Cons of int * myintlist
```

- `myintlist` is the same as `int list`
- Can define recursive functions on it, e.g. `length`

See `mylist.ml`.
Polymorphic types

length function has type myintlist \rightarrow \text{int}. How to apply it to the list of any types?

\text{type} \ 'a \text{mylist} = \text{Empty} \mid \text{Cons of} \ 'a \ast 'a \text{mylist}

- Polymorphic types: : put one or more type variables before type name
- mylist is not a type, but a type constructor.
- Must say int mylist, string mylist, or 'b mylist
Outline

Fundamentals

Data types

Higher-order functions

OOP v.s. FP
What is Functional programming?

“Functional programming” can mean a few different things:

- Avoiding \textit{mutation} in most/all cases
- Using functions as values
- Style encouraging recursion and recursive data structures

The most important concept in FP is \textit{first-class function}. 
First-class functions: Can use them wherever we use values

- arguments,
- results,
- parts of tuples,
- bound to variables,
- ...

Most common use is as an argument/result of another function. This “another function” is called higher-order function.

See higher_order_functions.ml.
Map and filter are, without doubt, in the “higher-order function hall-of-fame”.

- The name is standard
- You use them all the time once you know them: saves a little space, but more importantly, communicates what you are doing
- Predefined: List.map, List.filter

See map_and_filter.ml.
One more famous higher-order function fold

- fold also known as reduce, inject, etc.
- It accumulates an answer by repeatedly applying f to acc so far: fold_left f acc [x1;...;xn]) = f ..(f acc x1) ...xn

See fold.ml.
Outline

Fundamentals

Data types

Higher-order functions

OOP v.s. FP
Assume we want to implement a small language called *Expression*:

- Different *variants* of expressions: ints, additions, negations,
- Different *operations* to perform: eval, toString, hasZero,

<table>
<thead>
<tr>
<th></th>
<th>eval</th>
<th>toString</th>
<th>hasZero</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implement in OCaml:

- Define a type, with one constructor for each variant
- “Fill out the grid” via one function per column
- Each function has one branch for each column entry
Implement with Java/C++:

- Define a **class**, with one abstract method for each operation
- Define a **subclass** for each variant
- “fill out the grid” via one class per **row** with one method implementation for each grid position
FP and OOP often doing the same thing in exact opposite way.
Which is "most natural" may depend on what you are doing or personal taste.
Code layout is important, but there is no perfect way since software has many dimensions of structure.
Conclusion

- OCaml is a functional programming language with static, strong and inferred type system.
- We will illustrate several concepts in this course by using programs of OCaml.
- Download homework2.pdf on the course website.