An Introduction to Functional Programming

Intermediate Computer Science Pre-College Program
23-27 July 2018
– Module 1 –

An Introduction to Functional Programming

What is this Course about?

- Introduction to Functional Programming
- Presented in 4 modules (Modules 1-5)

What is Functional Programming?

- Programming languages can typically be categorized in terms of the set of programming abstractions they provide
  - Programming abstractions: "commands" that are "composed" to build "programs"
- Programming paradigm
  - Collection of programming abstractions
- Examples
  - object-oriented programming
  - imperative programming
  - logic programming
  - concurrent programming
  - functional programming
  - combinations of subset of the above

What is Functional Programming?

- Based on writing expressions that are then evaluated
- The crucial underlying building block is that of a function
- Examples:
  - OCaml (we'll use this one)
  - Haskell
  - ML
  - Erlang
  - Scheme
  - F#, etc.

OCaml

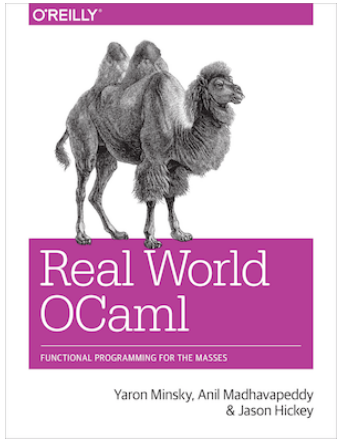
- Industrial-strength, statically-typed functional programming language
- Lightweight, approachable setting for learning about program design

Who uses OCaml?[1]

[1] Source: `www.seas.upenn.edu/~cis120`

Bibliography

- These slides
- Complementary reading:
  - Introduction to Objective Caml, a set of notes by Jason Hickey (`courses.cms.caltech.edu/cs134/cs134b/book.pdf`)
  - A great reference to continue learning (`realworldocaml.org`)



Installing OCaml (1/2)

- We'll perform two phases of installation
- For now all we need is OCaml itself and utop
  - utop is a top-level interpreter for OCaml
  - All programs will be executed inside utop
  - utop stands for "Universal Toplevel"
- Later we shall need a text editor

Installing OCaml (2/2)

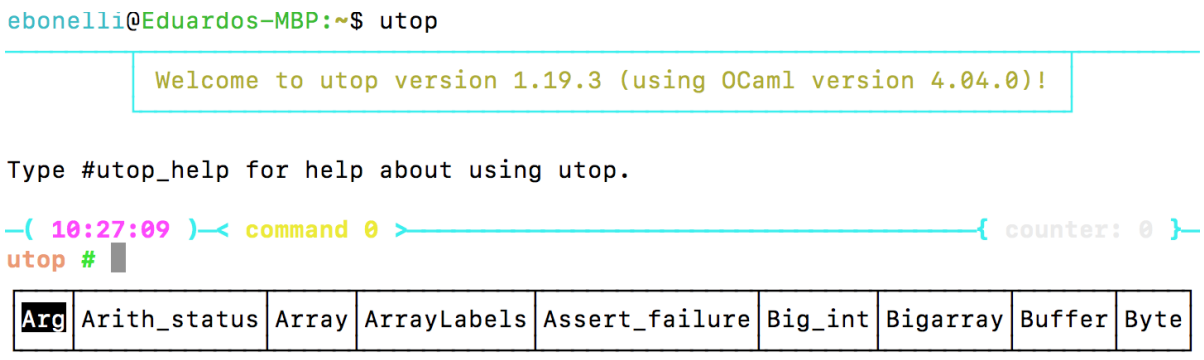
- Windows:
  `https://www.typerex.org/ocpwin.html`
- Linux/Mac
  - Install OPAM, the package manager for OCaml, following these instructions:

    `https://opam.ocaml.org/doc/Install.html`
  - After installing follow instructions:
    - `opam init`
    - `eval ‘opam config env‘`
  - In most cases, installing OPAM will also trigger the installation of OCaml
  - Install utop by typing the following in a terminal:

    `opam install utop`

Running `utop`

- Type `utop` in a terminal
- You should see something similar to this:



- utop displays:
  - the time
  - the command number
- To exit type CTRL-D or `#quit;;`

Basic Types

- Next we will begin experimenting with
  - basic types and
  - expressions of basic types
- These types include
  - `int`
  - `bool`
  - `float`
  - `string`
  - `char`
  - `unit`
- There are other types, we'll see them later

Type in every expression that follows in utop

`int` – integers

```
# 1;;
- : int = 1
# 12345 + 1;;
- : int = 12346
# 12345 - 1;;
- : int = 12344
# 3+4;;
- : int = 7
# 8/3;;
- : int = 2
# 30_000_000 / 300_000;;
- : int = 100
```

`float` – floating point numbers

```
# 3.5 +. 6.;; (* notice the dot after the + *)
- : float = 9.5
# sqrt 9.;;
- : float = 3.
# 1 + 2.0;;
      ^^^
  This expression has type float but is here used with type
    int
```

Note the last expression:

- The `+` function operates on integers, but `2.0` is not an integer

`float` – floating point numbers

- It is possible to convert from integers to floats and back
- `float_of_int` is called a function; we will study functions later

```
# float_of_int 1;;
- : float = 1.
# int_of_float 1.2;;
- : int = 1
# 1 + int_of_float 2.0;;
- : int = 3
```

`char` – characters

```
# 'a';;
- : char = 'a'
# 'x';;
- : char = 'x'
# "hello".[1];;
- : char = 'e'
```

`char` – characters

- OCaml provides a set of built-in modules
- Modules define useful operations on numerous types
- An example is the Char module which provides useful operations on chars

```
# Char.uppercase 'z';;
- : char = 'Z'
# Char.uppercase '[';;
- : char = '['
# Char.chr 97;;
- : char = 'a'
# Char.code 'a';;
- : int = 97
```

`string` – strings

```
# "Hello";;
- : string = "Hello\n"
# "Hello " ^ " world\n";;
- : string = "Hello world\n"
# "The character '\000' is not a terminator";;
- : string = "The character '\000' is not a terminator"
# "\072\105";;
- : string = "Hi"
# "Hello".[1];;
- : char = 'e'
```

`string` – strings

The `String` module provides many useful functions on strings

```
# String.length "Ab\000cd";;
- : int = 5
# String.sub "Abcd" 1 2;;
- : string = "bc"
```

`bool` - booleans

```
# 2 < 4;;
- : bool = true
# "A good job" > "All the tea in China";;
- : bool = false
# 2 + 6 = 8;;
- : bool = true
# 1.0 = 1.0;;
- : bool = true
# 2!=4;;
- : bool = true
 # true && false;;
- : bool = false
# true || false;;
- : bool = true
```

use = for equality checking

`bool` - booleans

```
# if 1 < 2
  then 3+7
  else 4;;
- : int = 10
# if 3!=4 then 1 else 2;;
- : int = 1
# if 2 then 3 else 4;;
    ^^^
Error: This expression has type int but an expression was
    expected of
type bool
```

`unit` - unit type

- Special type typically assigned to expressions that cause effects
- Example: `print_string` for printing a string

```
# ();;
- : unit = ()
# print_string "hello";;
hello- : unit = ()
# print_char 'a';;
a- : unit = ()
# print_int 3;;
3- : unit = ()
```

`unit` - unit type

- Expressions of unit type can be composed using ";"

```
# print_string "hello"; print_string "bye";;
hellobye- : unit = ()
```

Variables

- Variables are names given to values
  - These names always start with lowercase
- Variables allow these values to be reused
- Variables are declared using: `let` identifier `=` expression

```
# let x = 1;;
val x : int = 1
# let y = 2;;
val y : int = 2
# let z = x + y;;
val z : int = 3
```

Nesting Declarations

Declarations can be nested using the form

```
let variable=expression in expression
```

```
# let x = 1 in
  let y = 2 in
  x + y;;
- : int = 3
# let z =
   let x = 1 in
   let y = 2 in
   x + y;;
val z : int = 3
# let x =
    let y = 2 in y in
  x+1;;
- : int = 3
```

Basic Types vs Agregate Types

- Basic types seen so far
  - `int`
  - `bool`
  - `float`
  - `string`
  - `char`
  - `unit`
- Agregate types we shall see now
  - They are built out of simple types by composing them
  - We will see two composite type constructors (more, eg. lists, later)
    - Functions
    - Tuples

Functions

```
1 #  let succ i = i + 1;;
2 val succ : int -> int = <fun>
3 # succ 1;;
4 - : int = 2
5 # succ (succ 1);;
6 - : int = 3
7 # succ;;
8 - : int -> int = <fun>
```

Functions

An alternative definition using anonymous functions

```
1 #  let succ i = i + 1;;
2 val succ : int -> int = <fun>
3 # let succ2 = fun i -> i + 1;;
4 val succ2 : int -> int = <fun>
5 # succ2 1;;
6 - : int = 2
```

`fun`, used above, allows anonymous functions to be defined

```
1 # fun x -> x+1;;
2 - : int -> int = <fun>
```

Function Types

Lets take a closer look at the type of `succ`

```
# let succ i = i + 1;;
val succ : int -> int = <fun>
```

The type of `succ` is `int -> int`

What does this function do and what is its type?

```
# let f i = i>0;;
```

What happens if you evaluate `f 3.5`?

Exercise

- Define the function `sign` which given an integer returns 1 if it is positive, -1 if it is negative and 0 if it is zero.
- What is the type of `sign`?

Exercise

- Suppose we use a function of type `int -> bool` to denote a subset of integers, namely those for which the function returns true
- Such functions are called characteristic function of a set
- For example, `let f x = x mod 2` denotes the set of even numbers
- Define union and intersection of sets represented through their characteristic functions
- `union` and `intersection` should have type

```
(int->bool) -> (int->bool)-> int -> bool
```

Functions with Multiple Arguments

```
# let add i j = i + j;;
val add : int -> int -> int = <fun>
# add 2 3;;
- : int = 5
# add 2 3 4;;
Error: This function has type int -> int -> int
       It is applied to too many arguments; maybe you forgot
     a ';'.
```

Functions with Multiple Arguments

An alternative definition using anonymous functions

```
# let add2 = fun i j -> i + j;;
val add2 : int -> int -> int = <fun>
# add2 2 3;;
- : int = 5
```

Function Types

- Lets take a closer look at the type of `add`

```
1 # let add i j = i + j;;
2 val add : int -> int -> int = <fun>
```

- The type of `succ` is `int -> int -> int`
- How do we read this type?
  - `succ` is a function that

    given an integer `i`, returns a function that
    given an integer `j`, returns `i+j`

Partial Application

- `succ` is a function that

  given an integer `i`, returns a function that
  given an integer `j`, returns `i+j`

This means we can apply `add` to just ONE argument and get back a function

```
# add 1;;
- : int -> int = <fun>
```

A new way to define succesor!

```
# let succ3 = add 1;;
val succ3 : int -> int = <fun>
# succ3 4;;
- : int = 5
```

Exercise

- Define the function `min3` that given three integers returns the smallest one.
  - Use if-then-else and conjunction
- What is the type of `min3`?

Exercise

- Define the functions `and'`, `or'`, `not'` and `xor'` which implement the standard boolean operations.
- What is the type of each of these functions?

Tuples

Just like ordered tuples in math

```
# (2,3);;
- : int * int = (2, 3)
# (true,3);;
- : bool * int = (true, 3)
# (true,2,4);;
- : bool * int * int = (true, 2, 4)
#
# (2,(true,23));;
- : int * (bool * int) = (2, (true, 23))
```

- Tuples that have just two components are called pairs
- The type of a tuple is `t1 * t2 * ... * tn` where each `ti` is the type of the respective component

Tuples

How do we access the components of a tuple?

```
# let fst (x,y) = x;;
val fst : 'a * 'b -> 'a = <fun>
# fst (2,3);;
- : int = 2
# fst (2,3,4);;
Error: This expression has type 'a * 'b * 'c
  but an expression was expected of type 'd * 'e
```

Note that `fst` uses pattern matching:

- `(x,y)` is a pattern that can only match a pair
- binds variables `x` and `y` to the first and second component of the pair, resp.

Tuples

```
# let f2 (x,_)=x;;
val f2 : 'a * 'b -> 'a = <fun>
# f2 (2,3);;
- : int = 2
```

Exercise on Types

Provide expressions of the following types:

1. `bool`
2. `int * int`
3. `bool -> int`
4. `(int * int) -> bool`
5. `int -> (int -> int)`
6. `(bool -> bool) * int`

Records

A record is a labeled collection of values of arbitrary types.

```
# type db_entry =
{ name : string; height : float; phone : string; salary :
    float};;
type db_entry = {
  name : string;
  height : float;
  phone : string;
  salary : float;
}

# let jason =
{ name = "Jason"; height = 6.25;
phone = "626-555-1212"; salary = 50.0};;
val jason : db_entry =
{ name="Jason"; height=6.25;
phone="626-555-1212"; salary=50}
```

Records

Accessing the fields of a record

```
# jason.height;;
- : float = 6.25

# jason.phone;;
- : string = "626-555-1212"

# let { name = n; height = h } = jason;;
val n : string = "Jason"
val h : float = 6.25
```

Records

Functional update

```
# let dave = { jason with name = "Dave"; height = 5.9 };;
val dave : db_entry =
{name="Dave"; height=5.9; phone="626-555-1212"; salary=50}

# jason;;
- : db_entry =
{ name="Jason"; height=6.25;
phone="626-555-1212"; salary=50}
```

Records

Imperative update

```
# type db_entry = { name : string; height : float; phone :
    string; mutable salary : float;}

# let john = { name = "John"; height = 5.3; phone = "
    123-456-7890";  salary = 150.0};;
val john : db_entry =
  {name = "John"; height = 5.3; phone = "123-456-7890";
    salary = 150.}
# jason.salary <- 150.0;;
- : unit = ()
# jason;;
- : db_entry = {name="Jason"; height=6.25; phone="
    626-555-1212"; salary=150}
# let dave = { jason with name = "Dave" };;
val dave : db_entry =
{name="Dave"; height=6.25; phone="626-555-1212"; salary=150}
# dave.salary <- 180.0;;
: unit = ()
# dave;;
: db_entry = {name="Dave"; height=6.25; phone="626-555-1212"
    ; salary=180}
# jason;;
: db_entry = {name="Jason"; height=6.25; phone="626-555-1212
    "; salary=150}
```

Summary

- OCaml expressions
- Every expression has a unique type
- We've learned about basic types such as `int` and `bool`
- We've learned a little about agregate types such as `int -> int` and `int * int`
- We've learned to evaluate programs in `utop`
- A word on style: `https://www.seas.upenn.edu/~cis341/current/programming_style.shtml`
- Where can I practice more? `https://ocaml.org/learn/tutorials/99problems.html`