Values, types, variables and functions

Ziyan Maraikar

July 10, 2014

Table of Contents

Programming Languages

- ★ Countless languages created, dozens in widespread use.
- ★ Impossible (and useless) to cover them all!
- ★ How to decide what languages are “important”?

Programming Paradigms

Definition

A fundamental style of computer programming, a way of building the structure and elements of computer programs.

Imperative C/C++, Java/C#, Python/PHP, MATLAB

Functional ML family (SML, Ocaml, F#, Scala), LISP family, Haskell

Logic & constraint Prolog, constraint solvers

Very useful to learn more than one paradigm!

Ocaml

- ★ Versatile language supporting multiple paradigms.
- ★ Descended from ML (metalanguage) by R. Milner in the late ‘70s.
- ★ Research use: theorem provers, to compilers and OSs.
- ★ Industry use: Facebook(hack), Xen toolstack (XTS), financial trading systems.

Ocaml Toplevel

- ★ We will use the interactive Ocaml *toplevel* which gives immediate feedback to learn the language.
- ★ As you develop larger programs you will save them in files and compile them using the ocaml compiler for faster execution.

Table of Contents

Simple programs

A functional program is an *expression*. On execution, Ocaml *evaluates* the program and calculates its *value*. [1]
Example: arithmetic and boolean expressions are simple programs.

```
# 1 + 2 * 3 ;;
- : int = 7
# 1 < 2 ;;
- : bool = true
# true || false ;;
- : bool = true
```

Note how the value of an expression as well as its *type* is evaluated.

[1] Note: The ;; tells the toplevel to *evaluate* an expression, but this is *not* required Ocaml program files.

Types

Ocaml has a powerful, *strong* type system (unlike C.) that includes the following *primitives*.

Type	Common operators
`int`	`+ - * / mod`
`float`	`+. -. *. /.`
`bool`	`&& || not`
`char`	
`string`	`^` (concatenation)
All types	`= <> > <`

Mixing values of different types (implicitly) is not allowed!

Integers vs. Floating point

Floating point numbers use distinct arithmetic operators — the usual operator suffixed with a “.”

```
# 1.1 * 2.2 ;;
Error: This expression has type float but an expression was
     expected of type int
# 1.1 *. 2.2 ;;
- : float = 2.42
```

You need to explicitly convert ints to floats (and vice versa.)

```
# float 1 +. 2.2 ;;
- : float = 3.2
```

Dealing with typing Errors

- ★ Ocaml's type system can catch a lot of errors at compile-time, so less chance of runtime errors.
- ★ To fix a type error, locate the part of the expression that is underlined. Check whether the underlined term is of the type expected by the operator or function used.

Exercise

Are the following expressions correctly typed? If so, what is its type?

- `1 + 2 *. 1.5`
- `true && -1 >0`
- `true || 0`
- `"Your score is " ^ 80`

Table of Contents

Naming values

A value can be give a *name* using the `let` keyword.

```
let area = 2.0 *. 3.0;;
val area : float = 6.
```

- ★ This is *not* the same as assignment in C[2]. Variables (names) in Ocaml are used in the fashion of mathematics.
- ★ This form of `let` introduces a name into *global scope* — it is visible throughout the program.
- ★ Note that we did not specify the type of the variable **`area`**. Ocaml automatically *infers* the type by itself!

[2] more like constants in other languages

Local scope

The keyword *in* limits a variable's scope to the expression that follows.

```
# let pi = 3.14 in
    2.0 * pi * 5.0

# pi ;;
Error: Unbound value pi
```

We can define multiple local variables by nesting definitions:

```
# let pi = 3.14 in
  let r = 5. in
    2. *. pi *. r ;;
- : float = 31.4
```

Exercise

What is the value and type of each of the following expressions?

```
let x=3 mod 2 in
let y=3/2 in
    x*x + x*y + y*y

let a=(not true)||false in
let y=10.0 in
    y > 0. && a

let x=1 in
let x=2 in
    x * x
```

The definition of `x` in the closest surrounding scope "wins". Redefining the same variable like this is bad practice!

The type of a let expression

Question: In the general case, what is the type of the expression

```
let x = v in
    e_x
```

Answer: The type of an expression is the type of its result. Therefore the type of the let expression is the same as the type of e.

Table of Contents

Function definitions

Functions are defined using same *let* keyword that is used to define variables.

```
# let square  (x:int) :int =
    x * x ;;
```

- ★ **square** is name of the function.
- ★ **x:int** defines a parameter and its type.
- ★ **:int** at the end defines the type of the function result.
- ★ The expression following the = defines how function result's value is computed.

The type of a function

Note how Ocaml prints the type of the function.

```
val square : int -> int = <fun>
```

We read this as

square *is a function* ***from ints to ints***.

What is the type of a function with multiple parameters like this?

```
let foo (x:int) (y:bool) :string
```

```
int->bool->string
```

Type inference

Just as for variables Ocaml is able to infer a function's type, without us having to manually provide type information.

```
# let square  x =
    x * x ;;
val square : int -> int = <fun>
```

Adding explicit types helps resolve type errors, so follow this practice during the first half of the course.

Function Application

We *apply* (call) a function by writing the arguments after the function name.

```
# square 10
- : int = 100
```

In the simple case parentheses are not required. Note that the standard operators are just functions written in *infix* notation.

Operator precedence

When an argument is an expression, put it in parentheses so that it is evaluated before applying the function.

```
square (1 + 2);;
- : int = 9

# square 1 + 2 ;;
- : int = 3
```

The *precedence* of function application is higher than other operators so `square 1 + 2` means `(square 1) + 2`.

Composing function application

In functional programming we often build new functions by *composing* existing functions (*bottom-up design.*)

```
let quad x = square (square x)
```

Alternatively, we build programs by *decomposing* them into functions (*top-down design.*)

Exercise

- ★ Write a function **`sum_of_squares`** that takes two parameters x and y, computes $x^2 + y^2$. Use the **`square`** function.
- ★ Write a function **`circle_area`**. Use this to write the function **`cylinder-volume`**.

Summary of Concepts

- ★ Values and expressions (programs)
- ★ Primitive types
- ★ Names (variables)
- ★ Global and local scope
- ★ Function definitions
- ★ Type of a function
- ★ Function application
- ★ Precedence of function application
- ★ Function composition
- ★ Type inference

Reading: Ch1 and Ch2 of OFTVB.