Practical OCaml

Joshua B. Smith

Apress®

Practical OCaml

ISBN-13: 978-1-59059-620-3

ISBN-10: 1-59059-620-X

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Matt Wade
Technical Reviewer: Richard Jones
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Sofia Marchant
Copy Edit Manager: Nicole Flores
Copy Editor: Nancy Sixsmith
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Linda Weidemann, Wolf Creek Press
Proofreader: April Eddy
Indexer: Brenda Miller
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code/Download section.

CHAPTER 6

Primitive and Composite Types

Chapter 3 introduced you to the OCaml concept of type; here you will learn more about types—their importance and things you can do with and about them. This chapter discusses the *primitive* types in OCaml (they are sometimes referred to as *basic* types; both terms are used interchangeably here).

OCaml is strongly and statically typed, which means that all type information in a given program must be known at compile time. Further, the type information cannot change within the program while it is running. The compiler enforces this type information, and there are restrictions on which operations can operate on a given type.

The benefit of this feature is that certain types of errors are not possible in a strongly typed language. For example, the following code in C compiles and runs, but outputs the wrong answer:

```
#include <stdio.h>

int main() {
        int b = 0;
        float c = 10.;

        printf("%i\n",b+c);
        return 0;
}
```

This program prints `0` (on some systems, it might print a random number or other garbage) to the screen and returns normally. This kind of error is pretty easy to make and can be very difficult to find unless you are specifically looking for it. C++ handles the situation and prints out the correct answer, but it does so because of implicit conversion. Dynamically typed languages such as Perl and Python handle the situation and print out a correct answer—again due to implicit conversion.

"Wait a minute," you might say. "This example is faulty because it has numbers, and OCaml is one of the few languages that make a real distinction between ints and floats." Although it is true that OCaml differentiates, this program is still fundamentally wrong. You can make a more detailed case by looking at strings.

If you switch variable b to be a string, the C and C++ code no longer compile. The C++ code could be made to compile, though, via operator overloading. In the case of operator overloading, the programmer might never even know it happened.

These languages take a different view of type safety than OCaml does. No language can be ignorant of types because types are a very important part of computation and computer science. (Even machine language deals with types, although a given machine language might have only one type.) However, no single rule covers the way any given language deals with types.

In fact, there is no "One True Way" when talking about types. In the computer language world, there is often disagreement about what the term *strongly typed* even means. After you come to terms with the definition of the strength of the typing, you come to the difference between latently typed and dynamically typed behavior.

OCaml takes a particular stance about the importance of knowing type information: type information should be constant. The compiler should enforce the rules, and the runtime environment should, too. This is part of the OCaml design goal of safety.

Constant Type, Dynamic Data

The OCaml compiler knows all the type information in a program at the time it is compiled. This static typing does not mean that the data your program operates on must be static as well.

Much of the type checking and verification is done in the name of safety. Being safe means you can do away with a lot of the code that you would have to write to verify without these checks. Compile-time checks are more rigorous and much faster than runtime checks, (which is why OCaml code is so fast).

Integers (Ints)

In OCaml, *integers (ints)* are 31 bits because OCaml uses the 32nd bit for its own purposes.

Ints in OCaml are very fast; they just aren't as big as they are in some other languages. Precision is used to describe numeric types, but precision has nothing to do with accuracy when calculations are concerned.

The OCaml normal int might not be enough for your task, so OCaml supports three other integer types in the core library. The first is int32, which is a 32-bit signed integer. There are utilities to operate on int32 values in the Int32 module. You can define an int32 by simply adding an l (lowercase L) to the end of your number.

The int64 integer type is much like int32, except that it supports 64-bit ints (if your platform does). You append an L to the end of your int to create one. The last type of integers OCaml supports are nativeints, which are the native integers on the platform you are using. They are very similar to int32 and int64 types and can be defined by adding an n to your integers.

```
# 10l
  ;;
- : int32 = 10l
# 10L;;
- : int64 = 10L
# 10n;;
- : nativeint = 10n
# 10;;
- : int = 10
# let (+) x y = Int32.add x y;;
val ( + ) : int32 -> int32 -> int32 = <fun>
# 10l + 10l;;
- : int32 = 20l
# 10 + 10;;
Characters 0-2:
  10 + 10;;
  ^^
This expression has type int but is here used with type int32
#
```

Defining high-precision ints is very easy. You can also convert freely to and from the different ints using the specific module, though some information may be lost.

Operator overloading is not possible (if you have code that uses these other integers, you cannot use the + operator and have it work automatically), but operator overriding works very well. You can override the + operator to work on Int32 or Int64 by just redefining it in your code.

■ **Caution** You cannot mix code modules with this kind of redefinition in place.

Floating-Point Numbers (Floats)

The mathematical operators for *floating-point numbers (floats)* are different from the ones for ints, which can sometimes be frustrating for both new and seasoned programmers (especially when you are prototyping and are not yet sure what type you want to use for a given number).

I strongly caution against joining ints and floats in real code. You can easily define your own numeric type and make it behave however you want; the problem is that ints and floats really are different.

People who are used to coding in languages that unify ints and floats might disagree with this recommendation. This separation is also a source of flame warfare on Usenet (and will probably continue for many years).

Think of this: you do not have to round ints. The rounding problem present in floats has been a situation for a long time. It is such a big problem that languages such as Java have a standard library for handling arbitrary precision math and numbers. Equality is also an issue:

when is a float really equal to a float? When you add rounding into the mix, you can easily get a situation in which things are close to equal—but not quite. An example can be found in `modf`, which in OCaml is the same as in C (basically). This function takes a float (a double in C) and returns the integer part and the fractional part.

A good example of the rounding and equality problems inherent in floats is with `modf`, as follows:

```
# Printf.printf "%.16f\n" (fst (modf 1.2));;
0.2000000000000000
- : unit = ()
# fst (modf 1.2);;
- : float = 0.19999999999999996
# 0.2 = fst (modf 1.2);;
- : bool = false
# 1 = 1;;
- : bool = true
#
```

This example shows one of the other insidious issues with floating-point math: it might display correctly. If you debug by `printf`, you might never even see the subtle rounding problems that are causing trouble. Although rounding might never cause problems, this attitude is pretty far from the OCaml philosophy. This code is correct and valid code, showing once again that "correct" and "doing what I want" are not always the same thing.

Strings and Chars

Chars and strings are supported natively by OCaml. *Chars* are ASCII chars and can be defined using `'\<NUMBER>'`. The number can be any number from 000 to 255 and it must always have three digits. To display ASCII char 1, you use `'\001'`, and so on. Pattern matching has full support for chars and even supports a range operator (which enables inclusive matching of a given character range). Although strings are supported natively, there is a string-manipulation module (called String) in the standard library. The String module has many of the string operations you would expect, except for regular expressions.

```
# char_of_int 1;;
- : char = '\001'
# '\001';;
- : char = '\001'
# int_of_char '\001';;
- : int = 1
#
```

You can use chars in pattern matching. The following example also demonstrates the range operator (`..`). The range operator, which is used instead of enumerating each value in the range, works on chars and ints. In this example, the first pattern match is true on `'a'`, `'b'`, and `'c'`, but nothing else:

```
# let charfunc x = match x with
  'a' .. 'c' -> Printf.printf "You got a passing grade\n"
  | 'd' -> Printf.printf "You've got some academic trouble\n"
  | 'f' -> Printf.printf "Would you like fries with that?\n"
  | _ -> Printf.printf "Better a quitter than a failure, eh?.\n";;
val charfunc : char -> unit = <fun>
# charfunc 'a';;
You got a passing grade
- : unit = ()
# charfunc 'b';;
You got a passing grade
- : unit = ()
#
```

Regular expression support does not exist in the String module. In fact, many string operations that users of languages such as Python would expect are entirely absent from this module. However, these functions exist in the Str module. There is also a module that provides Perl-compatible regular expressions, although it is not part of the standard distribution.

Strings can be concatenated, searched, and otherwise manipulated. Strings also can be indexed like arrays via the normal OCaml indexing syntax:

```
# let b = "aalflfld";;
val b : string = "aalflfld"
# b.[1];;
- : char = 'a'
# b.[1]<-'d';;
- : unit = ()
# b;;
- : string = "adlflfld"
#
```

Strings are mutable, which is very important because many OCaml programmers use strings as buffers to pass mutable data. This practice can be very useful when working with data that might need to be mutable, but you don't want to use input/output (I/O) to handle.

String elements are modified via the OCaml assignment operator `<-`, which enables you to modify one element of a string. There is no built-in way to get a char list from a string (or vice versa).

OCaml does not know anything about Unicode; it uses ISO-8859-X to encode characters and strings. There are several third-party libraries that provide support for Unicode. You will need to use these (or implement your own) if you want to work with anything but ISO-8859-X.

Using the Pervasives Module

The *Pervasives module* is the module that is open by default in the OCaml toplevel. The Pervasives module includes functions that are, well, pervasive. For example, the `open_in` function is actually a function in the Pervasives module. You do not have to prefix it with the module name because the Pervasives module is open by default.

This section did not cover modules in depth, but don't worry. There is more coverage later on, and right now this information isn't critical to your use of OCaml.

Lists and Arrays

Lists are very powerful tools in OCaml. Lists can contain elements of only a single type (for example, a list of all integers or a list of all strings). However, you can make a type that suits your needs and then make a list of it. Most of the operations on lists are found in the List module instead of in the Pervasives module.

Lists in OCaml are implemented underneath by using a singly linked list, making the traversal of OCaml lists very efficient.

Arrays are much like mutable lists. Unlike lists, arrays allow for efficient random access. Arrays should be used when you want random access to elements in the container. Lists provide the capability to access random elements, but that access is not efficient.

Both arrays and lists are *polymorphic*, which means that they can be used with any OCaml type (even other polymorphic types).

Which tool should you use? This subject is not something I can easily give advice about. I usually use lists until I find the need to do a lot of random access. In my work, this does not happen often, so I normally just use lists. However, I always use arrays in the area of matrix manipulation because the Array module has matrix support.

Exceptions

Exceptions are their own type. This type, `exn`, is the type of all exception values.

The fact that exceptions are their own type has several ramifications for the OCaml programmer. You can, for example, write a function that takes an exception as an argument:

```
# let raise_if_unequal x y e =
    if not (x = y) then raise e;;
  val raise_if_unequal : 'a -> 'a -> exn -> unit = <fun>
# raise_if_unequal 10 30 Not_found;;
Exception: Not_found.
# raise_if_unequal 10 30 (Invalid_argument "hello?");;
Exception: Invalid_argument "hello?".
# raise_if_unequal 30 30 (Invalid_argument "hello?");;
- : unit = ()
#
```

This somewhat contrived example shows that exceptions are just like any other OCaml type. (Exceptions are covered in depth later in this book.) Just remember that the OCaml exception type is just like any other type in OCaml and is subject to the same limitations.

Other Types

OCaml has other types—some simple and some more specialized. For example, OCaml has a Boolean type called `bool` that can have only two possible values: `true` and `false`. You can convert a string to a `bool` using the function `bool_of_string`. If you are familiar with `bool` types in other languages, the `bool` types in OCaml will not be strangers to you.

The `lazy` type is another type. Although OCaml normally uses `eager` evaluation to evaluate all function arguments, there are times when it is easier to use `lazy` evaluation (especially in situations in which you want to defer computation until later). A simplistic example that highlights the difference between `eager` and `lazy` evaluation follows:

```
# let somef x = 100;;
val somef : 'a -> int = <fun>
# somef (lazy (1 / 0));;
- : int = 100
# somef (1 / 0);;
Exception: Division_by_zero.
#
```

When the function is called by using `lazy` evaluation, it yields to the correct answer. However, when `eager` evaluation is used, an exception is raised because the argument is evaluated first. Evaluation of the `lazy` value is performed by the `Lazy.force` function. After a `lazy` expression is evaluated, it does not get evaluated again, even if you force it. This is convenient because multiple forces do not result in multiple calculations. Also, after a `lazy` value is forced, it evaluates to that value from then on.

```
# let b = lazy (10 + 30);;
val b : int lazy_t = <lazy>
# Lazy.force b;;
- : int = 40
# b;;
- : int lazy_t = lazy 40
# Lazy.force b;;
- : int = 40
```

Polymorphic Types

OCaml supports polymorphic types natively. You can define your own polymorphic types as you define other types. For example, if you want to define a polymorphic type, you create the following:

```
# type 'a polytype = Dataitem of 'a;;
type 'a polytype = Dataitem of 'a
# Dataitem 10;;
- : int polytype = Dataitem 10
# Dataitem "hello";;
- : string polytype = Dataitem "hello"
```

These types are polymorphic until they are used with a concrete type (such as an int). The type is then concrete, so it can no longer operate on more than one type. However, you can use a polymorphic type, too. OCaml provides the `option` type, which is a polymorphic type, to handle many of the more common situations programmers face:

```
# Some 10;;
- : int option = Some 10
# None;;
- : 'a option = None
#
```

You can define a polymorphic type with more than one polymorphic element. You do this by adding more polymorphic notes:

```
# type ('a,'b) morestuff = MNone | MSome of 'a | MSomeMore of 'b;;
type ('a, 'b) morestuff = MNone | MSome of 'a | MSomeMore of 'b
# MSomeMore 1;;
- : ('a, int) morestuff = MSomeMore 1
#
```

Polymorphic functions can operate on polymorphic types. They are more difficult to define in practice, however, because so many of the OCaml operators are bound to a specific type that it can be difficult to write a function that does something valuable and have it be polymorphic. (You will learn more in later chapters.)

Composite Types

Previous chapters discussed records and variants. *Composite types* can be polymorphic and are defined in the same way as other polymorphic types.

You can define a type that is just a grouping of other primitive types. These kinds of types are represented much like tuples.

The elements of these types are inaccessible except via pattern matching, which has an important impact on how you write your code. This access is also one of the reasons why pattern matching is so important in OCaml.

Unlike other types you might define, naming composite doesn't bind them to this name. Although you can assign a name to a given composite type, the compiler doesn't report everything that matches that pattern as that type. However, you can use the named type to provide restrictions on function parameters, which can be very helpful to prevent confusion in polymorphic types.

```
# type 'a polytype = int * float * 'a;;
type 'a polytype = int * float * 'a
# let b x = match x with
  m,n,o -> m+n+o;;
val b : int * int * int -> int = <fun>
#
```

```
# let b (x:'a polytype) = match x with
  m,n,o -> (m,o);;
val b : 'a polytype -> int * 'a = <fun>
# b (10,10.,"hello");;
- : int * string = (10, "hello")
# b (10,10,"hello");;
Characters 2-17:
  b (10,10,"hello");;
    ^^^^^^^^^^^^^^^
This expression has type int * int * string but is here used with type
  'a polytype = int * float * 'a
#
```

This is an error that would have been caught at compile time instead of runtime. In this case, the error is not particularly important because the function drops that element. This restriction could have been done without using a named type and by substituting the type information directly into the function definition. However, named types can be helpful for documentation purposes.

The function is defined using pattern matching explicitly in the previous example. You also can define the function a different way and get the same signature (and thus the same functionality):

```
# let b (m,n,o) = m + n + o;;
val b : int * int * int -> int = <fun>
```

How you do this is up to you, although it is more convenient for many functions (especially functions using pattern matching for data structures) to do the latter (it is also more idiomatic for OCaml).

Polymorphic Variant Types

Some polymorphic types, referred to as *variant* types, can be created without using the `type` keyword. These types do not belong to a specific type the way that named types do. Instead, they are tagged with a value, and the compiler will ensure that the tag is valid and correct.

```
# [`Heart;`Club;`Diamond;`Spade];;
- : [> `Club | `Diamond | `Heart | `Spade ] list =
[`Heart; `Club; `Diamond; `Spade]
```

These types also can be named by using the `type` keyword; this type name can then be used for pattern matching:

```
# type suit = [`Heart | `Club | `Diamond | `Spade];;
type suit = [ `Club | `Diamond | `Heart | `Spade ]
# let winner m = match m with
    `Heart -> true
  | #suit -> false;;
    val winner : [< suit ] -> bool = <fun>
```

The variant tag does not belong to a particular type, although the type system ensures that the tag used is valid and correct. A variant type is inferred for every use of the type.

Why not use these types? Although they are somewhat efficient, it is harder to make optimizations without static typing information. Another problem with polymorphic variants is that they weaken the type discipline in your code. They are still typesafe, but they do more than simply ensure type safety. These other operations make them more heavyweight, so some kinds of errors are more difficult to detect. This is especially true because standard type definitions require more explicit type definitions that cannot be modified. For example, the following function definition is probably not correct:

```
# let winner m = match m with
    `Unknown -> true
  | `Heart -> true
  | #suit -> false;;
      val winner : [< `Club | `Diamond | `Heart | `Spade | `Unknown ] -> bool =
  <fun>
# winner `Unknown;;
- : bool = true
```

It does compile and it even works. You can make the compiler generate a warning if you specify the type (but it is probably not what you want):

```
# winner `Unknown;;
- : bool = true
# let winner (m:suit) = match m with
    `Unknown -> true
  | `Heart -> true
  | #suit -> false;;
      Characters 39-46:
Warning U: this match case is unused.
      `Unknown -> true
      ^^^^^^^
val winner : suit -> bool = <fun>
# winner `Unknown;;
Characters 7-14:
  winner `Unknown;;
         ^^^^^^^
This expression has type [> `Unknown ] but is here used with type suit
#
```

It compiles, but then fails when it is used. This is one of the biggest reasons to use polymorphic variants with care.

Conclusion

Take a look at a short example that displays an actual program and uses some of the concepts discussed in this chapter (some random number generation was added). Note that `self_init` is very important. Without this initialization, the OCaml pseudo-random-number generator is more pseudo than random.

```
open Random;;
Random.self_init ();;

let situations = [| "Ship about to explode";
                                "Ship Hailing Us";
                                "Klingons off the Starboard bow"|];;
let responses = [| "Hail Ship";
                                "Send Friendship Message";
                                "Shoot To Kill";
                                "Abandon Ship"|];;

let display_current_situation () =
  Printf.printf "Captain! %s\nWhat do we Do?\n"
    (Array.get situations (Random.int (Array.length situations)));;

let show_menu lst =
  Array.iteri (fun x y -> Printf.printf "%i     %s\n" x y) lst;
  Printf.printf "\nResponse? ";;

let respond x = match x with
  "Hail Ship" -> "Hailing, Sir."
  | "Send Friendship Message" -> "They like me,  they really like me!"
  | "Shoot To Kill" -> "But, we come in Peace!?!"
  | "Abandon Ship" -> "Iceberg, right ahead!"
  | _ -> "Captain, I just don't understand you!";;

let _ =
  display_current_situation ();
  show_menu responses;
  Printf.printf "%s\n"
    (respond (Array.get responses (int_of_string (read_line ()))));;
```

As you move forward, you'll see that many of the techniques used in this light-hearted example are useful in many areas. This is especially true now that you have a good grounding in the OCaml built-in types and how to create your own. A good understanding of the OCaml type system takes you a long way toward fully understanding OCaml.