PS1: Introduction and OCaml

Lior Zur-Lotan, Tal Barami

October 17, 2018

Contents

1 Introduction .................................................. 1
  1.1 Course communication policy .......................... 1

2 OCaml .......................................................... 1
  2.1 Programming in OCaml ................................. 1
  2.2 The Type System and Type Inference .......... 2
    2.2.1 Basic OCaml syntax .............................. 2
  2.3 Union Types ............................................. 3
    2.3.1 Union Types examples ......................... 4
    2.3.2 Polymorphic types .............................. 5
  2.4 Pattern matching ..................................... 6
  2.5 Function Currying .................................. 8
  2.6 Modules and Module signature .................... 9

1 Introduction

- Course homepage: http://www.cs.bgu.ac.il/~comp191/
- Useful Resources: http://www.cs.bgu.ac.il/~comp191/Useful_Resources

1.1 Course communication policy

All course communication will be done only via the course email (cs.comp191@gmail.com).
Emails sent to the personal mailboxes of course staff will be deleted and you will not get any response.

2 OCaml

This year the compiler will be written (mostly) in OCaml, an object oriented, functional, Strongly and Statically typed programming language. It can be compiled to
just about anything: native code, byte code for various VMs, and also comes with
an interactive REPL.

There are lots of good sources for learning Ocaml. In particular, we suggest
reading through Learn Ocaml in Y minutes and the official (and complete) online
Ocaml manual.

In the rest of this PS, we’ll cover a few key aspects of Ocaml. This is not,
however, a comprehensive introduction to Ocaml. Learning Ocaml in depth in this
course will mostly be a self-learning process, and it is your responsibility to learn how
to use it in order to be able to complete course assignments.

2.1 Programming in Ocaml

An Ocaml REPL is called a toplevel. The default toplevel shipped with Ocaml is
called ocaml.

An alternative toplevel you can use on a personal computer is utop which you
can install using opam (Ocaml PApple Manager) which in turn can be installed
with apt-get on Debian and its derivations (e.g. Debian, Ubuntu, Mint etc). utop’s
features include tab-completions, suggestions and curser-control.

Writing Ocaml code can be done in any IDE or text editor. A couple of note-wor-
thy mentions are Visual Studio Code (not to be confused with the full IDE Visual
Studio) and emacs. On a personal computer, you can also use ocaml-top (also in-
stalled with opam) and add better support for ocaml in emacs using Tuareg mode
(i.e. Tuareg plugin).

2.2 The Type System and Type Inference

Ocaml types are a major component in the language. Not only do they define the
way and shape data is stored, but also drive some of the logic and control flow.

A strongly typed language is one where the programmer isn’t allowed to implicitly
convert between types. No language is strictly strongly or weakly typed, but Ocaml
is considered quite strong and stronger than c or Java, for instance.

A statically typed language is one where the types are completely known at
compile time. Unlike languages like Scheme or Python, where only values have a
type (at runtime) and variables are untyped (i.e. dynamic), in Ocaml and other
statically typed languages (such as c and Java) the type of each variable or function
is known to the compiler. In Ocaml you can omit the type declaration of variables,
and the compiler will infer it from usage (except in rare cases where it is actually
impossible).

2.2.1 Basic Ocaml syntax

\# (* this is an Ocaml comment *)
\# let x = 1;; (* the sign ;; terminates expressions. *)
val x : int = 1 (* output includes both type (int) and value (1) *)

# let y : char = '2';; (* Declaring the type is optional *)
val y : char = '2'

# let z = [x + 2];; (* An Ocaml list *)
val z : int list = [3; 2]

# let foo = fun x -> x + 2;;
val foo : int -> int = <fun> (* 'fun' is Ocaml's syntax for 'lambda' *)

# let goo = fun x y -> x + y;; (* goo is a function from int * int to int *)
val goo : int -> int -> int = <fun>

# let u = ();; (* the 'unit' value, denoted with (),

unit is most commonly used to define a function
that takes no arguments *)
val u : unit = ()

# let foo2 x = y (* note that foo2 is a function. In Ocaml, you can
define a function without using the 'fun' keyword
by declaring a "variable" with an argument *)
val foo2 : 'a -> char = <fun>

(* 'a is a polymorphic type, meaning it can stand in for any other type.
  More on this soon *)

# let foo3 () = 3;; (* foo3 is a function since it takes an argument - 'unit'.
  Without the argument foo3 would have been a variable *)
val foo3 : unit -> int = <fun>

# let rec fact n = (* Recursive fucntions are defined with "let rec n" *)
  if n = 0 then 1 else fact (n - 1) * n;;
val fact : int -> int = <fun>

# fact 4;;
- : int = 24

# let foo4 () = (* "let" in Ocaml has two meanings. The first is like
  Scheme's "define" *)
  let x = 1 (* The second is like Scheme's "let" *)
  and y = 6 in
  x + y;;
val foo4 : unit -> int = <fun>
# foo4 ();;
- : int = 7

2.3 Union Types

The correct way to program in Ocaml is to accurately model the program’s data using custom types and let the powerful Ocaml type system verify the code that uses that data (leaving very few possible runtime bugs). To facilitate this, Ocaml lets programmers build custom types called union types. Union types can be combined into more complex union types so that any type, simple or complex, can be defined.

Union types are defined using the syntax:

type < type name > =< C'tor > of < underlying type > | < C'tor > of < underlying type > | ...

2.3.1 Union Types examples

# type simple = Ctor of int;; (* This is a union of a single
   type constructor named "Ctor" *)

type simple = Ctor of int

# Ctor 5;;
- : simple = Ctor 5

(* Note, constructor names must begin with upper-case letter *)

# type mixed_type = S1 of string | S2 of string | I of int;;

type mixed_type = S1 of string | S2 of string | I of int

# let x1 = S1 "a string";;
val x1 : mixed_type = S1 "a string" (* note that the type constructor is
   a part of the variable's value *)

# let x2 = S2 "a string"
val x2 : mixed_type = S2 "a string"
#
- : bool = false (* x1 is not equal to x2 since they where built
   using different type constructors *)

# let x2 = I 1;;
val x2 : mixed_type = I 1

# [I 1; S1 "a" ; S2 "b"];; (* Create a list with 3 elements

4
of the same type, mixed_type *)

- : mixed_type list = [I 1; S1 "a"; S2 "b"]

As a slightly more interesting example, we'll define a list type, similar to scheme
lists (nil terminated nested pairs). We'll call it slist (scheme-list):

# type slist = Nil | Cons of int * slist;; (* int * slist means a tuple of
an int and an slist *)
type slist = Nil | Cons of int * slist

This means that the type slist is the union of the two types defined by two type
constructors: Nil and Cons. Nil takes no parameters and so it has no underlying
type. Cons, however, is a pair of int * slist (note that we can define union types
recursively).

To create an slist, we use one of the two type constructors:

# let l1 = Nil;;
val l1 : slist = Nil

# let l2 = Cons(1,Cons(2,Nil));; (* The parentheses and commas
here denote a tuple, not
a list of arguments *)
val l2 : slist = Cons (1, Cons (2, Nil))

    type mixed_type = S1 of string | S2 of string | I of int
    mixed_type = S1 of string
            | S2 of string | I of int
The types of both l1 and l2 are both slist. Nil and Cons are
just two ways to construct an slist.

2.3.2 Polymorphic types

Our slist isn't very useful. It only allows for lists of integers. Using Polymorphic
Types We can do better.

(* our Scheme-like Polymorphic List type *)

# type 'a spolist = Nil | Cons of 'a * 'a spolist;;
type 'a spolist = Nil | Cons of 'a * 'a spolist

Polymorphic types might remind you of templates in C++ or generics in Java,
but they are not the same (Ocaml's generics mechanisms are used with Ocaml OO
features, not union types). 'a is a place-holder for a concrete type. It serves the
purpose of defining the structure of spolist without giving a concrete underlying
type. Since we don't care what 'a is, we don't really need it to be concretely defined.
Once the type system infers the concrete type used instead of 'a for some
variable/expression the 'a spolist type will be constricted and the placeholder 'a will
be replaced with a concrete type.
Remark. Note that instead of 'a we could name the place hold however we like. For instance 'some_type_place_holder. It's just a name prefixed with a quote ('). The reason 'a, 'b and 'c are commonly used is to follow the the convention in the literature which denots types with $\alpha \beta$ and $\gamma$ (which in turn are commonly denoted with 'a 'b and 'c)

Using this new spolist type we can create scheme-like lists of any type:

```
# let l1 = Nil;;
val l1 : 'a spolist = Nil (* The underlying type is not constricted
                                yet since Nil is no underlying type *)

# let l2 = Cons(1,Nil);;
val l2 : int spolist = Cons (1, Nil) (* This is now an int spolist
                                instead of an 'a spolist *)

# let l3 = Cons('c',Cons(1,Nil));;
Characters 23-24:
let l3 = Cons('c',Cons(1,Nil));;W
```

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

On line 1, we see that the underlying type of l1 is still abstract ('a). That’s because the compiler still isn’t forced to constrict it (the underlying type is irrelevant to the Nil constructor).

Contrarily on line 5, l2’s type is concretely defined as int spolist, since it’s constricted by the Cons constructor that received an int.

Finally line 9 isn’t a valid definition because we created a conflict in the type system. This shows the difference between a spolist and an actual Scheme list. Scheme lists can contain elements of different types in a single list, while spolists can only hold a single data type in a single list. This is enforced by the type system. The first call to the Cons constructor created the constraint that the first member of the tuple is a char, so 'a is replaced with char. However, this constraints the second tuple member to be a char spolist, so the second call to Cons expected a char (to construct a char spolist). When it received an int, it raised a compilation error.

2.4 Pattern matching

Using union types, we can create complex types from simple ones (like spolist above). Functions that take these complex types need to be able to handle the different constructed values as well as get access to those underlying values (those sent to the constructors).

For instance, we would like to create a car and cdr functions for our spolist type. To
do this we would need to handle Nil values and Cons(hd, tl) values differently. We want something that’s like function overloading coupled with a switch-case. This, as well as value unpacking (for underlying types) is done with Pattern Matching in OCaml.

Pattern matching allows the programmer to define handler (i.e. body of code) based on matching a value with a pattern. Take car for example:

```ocaml
(* Creating an exception type constructor that takes a string *)
exception Invalid_list of string

# let car l =
  match l with
  | Nil -> raise(Invalid_list "Can't apply \"car\" to Nil")
  | Cons(hd, tl) -> hd
  | Cons(hd, Nil) -> hd;;

val car : 'a spolist -> 'a = <fun> (* Our polymorphic car *)
```

(* When pattern matching entire functions you can use the
"function" syntactic sugar and omit the "match" keyword.
Note that we don't define a parameter for cdr
since "function" takes just a single argument to match . *)

```ocaml
# let cdr = function
  | Nil -> raise(Invalid_list "Can't apply \"cdr\" to Nil")
  | Cons(hd, tl) -> tl;;
val cdr : 'a spolist -> 'a spolist = <fun>
```

```ocaml
# let l4 = Cons(1, Cons(2,Nil));;
val l4 : int spolist = Cons (1, Cons (2, Nil))
```

```ocaml
# car l4;;
- : int = 1
```

```ocaml
# cdr l4;;
- : int spolist = Cons (2, Nil)
```

The match directive takes a variable, and tries to match it with a list of patterns, and executes the matching body. Pattern matching is not only useful for value unpacking. We can also create more complex patterns to implement part of the logic as pattern matching. For instance if we want to create list contains:

```ocaml
      Note that if a case can’t be used since it’s fully covered by previous cases the compiler will issue a warning (see line 10). This is called Shadowing. When a pattern is shadowed it can never be used. This is almost always a mistake.
```
# let l5 = Cons(1, Cons(2, Cons(3, Nil)))::;
val l5 : int spolist = Cons (1, Cons (2, Cons (3, Nil)))

# let l6 = Cons(1, Cons(2, Cons(0, Nil)))::;
val l6 : int spolist = Cons (1, Cons (2, Cons (0, Nil)))

# let rec list_contains l x = (* "rec" is required to declare
   a recursive function *)
match l, x with (* Match against 2 variables *)
| Nil, _ -> false
| Cons(hd, tl), x when x = hd -> true
| Cons(hd, tl), x -> list_contains tl x;;
val list_contains : 'a spolist -> 'a -> bool = <fun>

# list_contains l5 0;;
- : bool = false

# list_contains l6 0;;
- : bool = true

A few things to note:

- On line 10, note that the pattern matches x against _. This is a catchall
  pattern, which means "I don’t care about the type or value of x". _ is used
  as a convention. We could have used any variable name we wanted.

- On line 11 we use a guard, an extra condition that guards the pattern. The
  body of the pattern will only execute if the pattern is matched and the guard
  is true.

- Reordering the 2nd and 3rd patterns will produce different results due to shadowing.

2.5 Function Currying

Notice the signature of list_contains above:

val list_contains : 'a spolist -> 'a -> bool = <fun>

This means that list_contains is "a function which expects a 'a spolist and returns a
function which expects a polymorphic type 'a and returns a bool", but this doesn’t
seem like what we intended exactly, and is not exactly how we used it. In a way,
all functions in Ocaml take a single argument (remember that a function that takes
no arguments actually takes the Unit argument). If the signature of a function
includes several arguments, what is actually defined is a high-order function (i.e. a
function that takes or returns another function) that takes a single argument and
returns a new function that takes the next argument and so on. For instance, we
can call our list_contains function with a single argument list like so:

```ocaml
# let l6_contains = list_contains l6;;
val l6_contains : int -> bool = <fun>
```

and what we get is a new function (notice the signature), that takes an int and returns
whether that int can be found in l6. Currying is order sensitive. The following call
is a type error:

```ocaml
# let list_has_zero = list_contains 0;;
Characters 34-35:
  let list_has_zero = list_contains 0;;
```

Error: This expression has type int but an expression was expected of type
'a spolist

This should be clear from the signature list_contains : 'a spolist -> 'a -> bool, or at
least if we remember that the -> annotation is right-associative. So we might write
this signature as ('a spolist -> ('a -> bool)).

Note that we can also use Ocaml auto-currying the other way. Call a high-order
function with multiple arguments.

```ocaml
# let foo x =
  fun y -> x + y;;
val foo : int -> int -> int = <fun>
```

```ocaml
# foo 2 4;; (* equivalent to (foo 2) 4;; *)
- : int = 6
```

2.6 Modules and Module signature

Now let's take our 'a spolist type along with its 3 functions and pack it all up to-
gather. This "package" is called a module. Every piece of code in Ocaml is placed in
a module. An Ocaml file named spolist.ml implicitly defines a module named Spolist
(modules always start with a capital letter) which holds all the contents of that file.
Libraries in Ocaml contain a set of modules. Ocaml modules contain any Ocaml con-
struct, including sub-modules, which are just modules inside other modules defined
with the "module" directive.

Remark. Note that modules are not classes in Ocaml despite the use of the dot
(\.) operator to access module members which can be confusing, but Ocaml has
an extensive object oriented system which includes classes and many other features,
which are not covered in this PS.
By default, if the programmer does nothing, the entire module is exposed to the program that uses it. The default signature of our Spolist module is:

```plaintext
module Spolist :
  sig
  val x : int
  val y : char
  val z : int
  val foo : int -> int
  val u : unit
  val foo2 : unit -> int
  type slist = Nil | Cons of int * slist
  type 'a spolist = Nil | Cons of 'a * 'a spolist
  val l1 : 'a spolist
  val l2 : int spolist
  exception Invalid_list of string
  val car : 'a spolist -> 'a
  val cdr : 'a spolist -> 'a spolist
  val l4 : int spolist
  val l5 : int spolist
  val l6 : int spolist
  val list_contains : 'a spolist -> 'a -> bool
  val l6_contains : int -> bool
end
```

The only things we would like to expose are the 'a spolist type, the Invalid_list exception and the 3 functions; car, cdr and list_contains.

To "hide" the internals of a module, we can create a signature using the sig and module keywords:

```plaintext
module Spolist : sig
  type 'a spolist = Nil | Cons of 'a * 'a spolist
  exception Invalid_list of string
  val car : 'a spolist -> 'a
  val cdr : 'a spolist -> 'a spolist
  val list_contains : 'a spolist -> 'a -> bool
end = struct
  type 'a spolist = ...
  let car = ...
  let cdr = ...
  let list_contains = ...

  ...
  let hidden_var = 1 (* Internal implementation details *)
```
... end;;

Now, when we evaluate the Spolist module we get:

module Spolist :
  sig
    type 'a spolist = 'a Spolist.spolist = Nil | Cons of 'a * 'a spolist
    exception Invalid_list of string
    val car : 'a spolist -> 'a
    val cdr : 'a spolist -> 'a spolist
    val list_contains : 'a spolist -> 'a -> bool
  end

Remark. To create a signature for an entire file, you need to create an interface file with the suffix ".mli" with the same syntax as the module signature above.