Software Verification

2023-2024

Quick Tutorial on OCaml

Vincent Penelle

Topic

- OCaml

OCaml is the language used in this course. The goal of this document is rather to be a cheat sheet than anything else: we'll discuss about it together. Don't waste too much time on it. If you're not familiar with OCaml, you might find tutorial there[1], or, for more precise questions, go to the documentation[2]. We summarise here some useful notions alongside some short exercises on them.

Download the code[3] and open the file `exo1.ml`. It contains what is discussed here, alongside the prototypes of small exercises.

Before starting, you have to type the following commands in a terminal from which you'll launch vs-code (if you use that editor, otherwise the first two commands are to type where you'll compile):

- `export OPAMROOT=/opt/local/opam`
- `eval $(opam env)`
- `code`

It is advised to use the extension OCaml platform under vs-code if you use that editor.

Exercise 1: Use and Compilation: In this course, we implement a software compiled through a Makefile. However, if you want to test some functions/program separately, you have two ways of doing it. The quickest (but less durable) is to launch the *toplevel*. You can do it with the command `ocaml`, but it is not very practical. A much better toplevel exists: `utop` (it is installed in CREMI). It is also possible to open a .ml file in it.

The other option is to create a file `MyFile.ml` and compile it. The compiler, `ocaml` has a similar usage as gcc. For simple project, it is sufficient to use it, but with bigger project with multiple libraries and dependencies, it is not the easy to use it properly.

Fortunately, there are automatic builders for OCaml. The oldest (and now quite deprecated) is `ocamluild`. A more recent, that we'll use here, is named `dune`. We have hidden the process under a Makefile, you'll thus be able to compile the exercise with `make exo1.native` (native version) or `make exo1.byte` (bytecode version). You may directly

[1] `http://OCaml.org/learn`

[2] `http://caml.inria.fr/pub/docs/manual-OCaml/`

[3] `TP-SMT.tar.gz`

use dune with the command `dune build`, in which case compiled files will be placed in `_build/default`.

Dune allows as well to load libraries of a project in utop easily. For that, you simply type `#use_output "dune top";;` in a utop session, and you'll have access to the libraries of the project in which your session is. For the current project, dune is configured so it is the case of the modules treating Keen (so not for this tutorial). That might allow you to test quickly features you are developing in their context without having to write a main.

Dune is a complete build system, very customizable, and thus not so easy to learn. The goal of this tutorial is not to explain it to you. We direct you to its documentation if you wish to go further. In this course, we'll provide you with projects already configured with dune so that its use is transparent for you.

Exercise 2: Functional Language: We will use the functional paradigm of the language (technically, it is possible to use imperative, but it is less elegant in OCaml, and essentially less adapted to our project). A file is executed from top to bottom (like in C, it can contain several files: if a function from another file is called, dune will compile it – that is partly why it is complicated with the compiler).

The file is a sequence of requests which modify the memory of the program. A request is of the form `let name = expr`, where name is a string and expr is an expression of the language. For example, `let toto = 4+2` will put 6 at the address toto of the memory (not very well said, but it is broadly that). It is also possible to evaluate an expression without putting its result in the memory with

All elements manipulated by the language are functions (a constant is simply a function without argument). Functions are thus defined with the request `let` (by giving name to the arguments). For exemple, `let f x = x + 1` defines the function f which take an int x and returns x+1. It is not necessary to precise types (the compiler infers them and signals inconsistencies). The request `let ... in` defines a local function. That is generally useful in other definitions (and serves as expression terminator). For example, `let f x = let succ x = x+1 in if(x < 0) then 0 else succ x`.

There are also expression of type `unit` which are actions that are executed when defined, e.g. display with `Printf.printf`. It is not necessary to name unit expression, and thus we often simply write `let _ = Printf.printf "Toto"`. It is possible to use them in other expressions (with `let ... in`) in which case the action will be executed every time the expression is evaluated. It is alternatively possible to use `;` to chain expressions: `A ; B` is equivalent to `let _ = A in B`.

In the toplevel, it is necessary to use `;;` to evaluate a bloc of code. For compiled files, it is useless (though tolerated).

As said before, all objects manipulated by the program are functions. It is thus obviously possible to have functions as argument of other functions. For example, `let g f x = (f x) + 1` defines g which has a function f : `'a -> int` and a `'a` x as argument and returns the result of f x plus 1. Here, `'a` stands for any type: the language supports polymorphic function, and this function can thus be used on any function whose type returns an int and has a single argument. For example `g int_of_string "25"` est l'int 26.

Other example `let disjunct f g b = if(b) then f else g` is a well defined function (f and g must have the same type, and can have arguments). Finally, we can use the notation `fun` to define anonymous functions (among other uses) as follows: `let res = (disjunct (fun x -> if(x < 0) then 0 else x+1) (fun x -> x-1) true) 3` is a well defined int (3 here). It is possible to use the syntax `let f = fun x -> x+1`.

Important syntax point: in OCaml, *structural* equality is denoted `=` and not `==` as in most languages. `==` exists, but is the physical equality. Similarly, structural inequality is

`<>` and `!=` is physical inequality. As a rule of thumb, you want structural inequality. To convince yourself, observe `let a = 1.1 in a = 1.1` is true while `let a = 1.1 in a == 1.1` is false.

Other point: be wary of parentheses: there is left associativity, therefore the term `f x+1` will be interpreted as `(f x) + 1`. Iit is preferable to put parentheses to desambiguise expression. Note that passing arguments to a function is made without parentheses (what we would write `f(1,2)` in C is written `f 1 2`).

Exercice: Define a function `compo` which take two (arbitrary) functions as arguments and returns their composition (be careful here, you cannot omit the argument of the composition).

Exercise 3: Recursion and Tail Recursion: A versatile and important function in functional paradigm is recursion. When defining a recursive function, it is necessary to tell it with the compiler with the keyword rec as follows: `let rec toto x = if (x < 0) then 0 else 1 + (toto (x-1))`. Without it, the compiler will not compile it, except if toto is already defined, as redefinition is allowed (that is actually why we need this keyword).

However, the downside of recursion is that it uses the call stack, and that it is far from bottomless. For example, if you call `toto 10000000`, your program will raise a Stack Overflow exception, while with a loop there would be no issue. Fortunately, OCaml is optimised to avoid using the call stack in case of *tail recursion*, i.e., when a recursive function whose recursive call is the *last* instruction. All functions ar not writable as tail-recursive, but a lot of them are (tail recursive programs are the same as LOOP programs). For example, toto may be rewritten as follows: `let toto x = let rec aux x acc = if(x < 0) then acc else aux (x+1) (acc-1) in aux x` Note that it is aux which is now recursive, not toto. This technique of using an accumulator to convert the function to a tail-recursive one is usually a good way to do it.

Exercise: Define $x \to 2^x$ as a tail-recursive function.

Exercice (harder): Define Fibonacci function as a tail-recursive function.

Exercise 4: Types and Pattern Matching In OCaml, base types are the same as in C (int, float, bool, char, etc). There exist as well complex types usable without defining them, e.g. tuples ($(2,'c')$ is of type int*char). It is possible to use them directly (and the compiler will infer them without problem).

It is possible to name types with the request `type`. For example, `type toto = int*char`. But the most use of that request is to define ≪collection types≫, which will be defined with constructors. They can be defined as follow: `type toto = None | One of int | Two of int*char` defines a type that can contain either nothing, an int or a pair int*char. It is also possible to define recursive types (e.g. `type tree = Leaf of int | Node of tree*tree`). Note that constructors must start with an uppercase and types with a lowercase.

But these types would be of little use without one of the most powerful feature of OCaml, the pattern matching: it allows to unpack the content of a type and return different values in function of the case we are (as long as all those values have the same type of course). For example, the sum of all elements of a tree defined above can be defined as follows: `let rec sum_of_tree t = match t with Leaf a -> a | Node (t1,t2) -> (sum_of_tree t1) + (sum_of_tree t2)`.

It is possible to have more complex motives in a pattern matching, not naming certain elements if they are irrelevant (with _), and do a case ≪otherwise≫ (also with _). For example: `let rec silly t = match t with Node(Leaf _,_) -> 1 | | Node(_,t1) -> 1 +`

`silly t1 | _ -> failwith "arg"`. We'll note that here, my default case is different: it returns an exception that will terminate the program if it is not caught (that can be useful in cases that are not suppose to occur and where instead of putting a value, we simply want the program to fail). The type inference will be ok with this. It is always useful to do so: pattern matching must be exhaustive.

Finally, we can mention the keyword when that allows to restrict a pattern to certain values: `let toto a = match a with Some a when a < 0 -> 0 | Some a -> a | None -> 0`. Note that in the two above cases, there is no ambiguity: the pattern matching is read from top to bottom and stop as soon as it matches a pattern. The order is thus important. Moreover, constructors None and Some are the constructor of the type option which is a generic type allowing to a function to sometimes return no value (None). That allows to avoid exceptions in the normal flow of a program (and it is very useful).

Exercise: Define a type representing a direction (4 cardinal points) and a speed (and a value saying to stay put). Define as well a function applying an element of that type to a position, but which does nothing if the speed is negative.

Exercise 5: Lists: We will use lists a lot in our project. You can look the manual page `https://ocaml.org/api/List.html` for more information. We'll simply cover the basics here.

A list is a collection of elements linearly ordered. We have access to the head of the list in constant time. Adding an element on the head of the list is constant time, but in queue it is linear. The empty list is named `[]`. `a::l` represents the list `l` preceded with `a`. For example `1::2::[4;5] = [1;2;4;5]`. Functions `hd` and `tl` are defined as follows: `hd (a::l) = a` and `tl (a::l)` = l. It is of course possible to use pattern matching on lists.

However, the big use of the List module lies in its functions for list operations. We'll present the three more versatile and useful to us:

- `map` is a function that applies a function to all the elements of a list. Its type is `('a -> 'b) -> 'a list -> 'b list`. For example `List.map (fun x -> (int_of_string a) + 1) ["12";"0";"-2"]` is the list `[13;0;-1]`.

- `iter` is a function that recursively applies left to right to all elements of a list a function of return type unit. Its type is `('a -> unit) -> 'a list -> unit`. For example, `List.iter (Printf.printf "%d ++ ") [1;2;3]` displays "1 ++ 2 ++ 3 ++ ".

- `fold_left` is very similar to the previous except it deals with function with a return value they can take as an argument. Its type is `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a list`. `List.fold_left f a [b1;b2;b3]` is the same as `f (f (f a b1) b2) b3`. a is the initial value, and the return value might be seen as an accumulator. It is a very practical manner to iterate on a list without using a loop. For example `List.fold_left (fun x y -> x + y) 1 [2;3;4]` returns 10.

Note that the three previous function are tail-recursive (and so don't generate stack overflow). There is also `List.fold_right` that applies the function on the reverse orde, but it is not tail-recursive, so to avoid if possible.

Exercise : In the previous exercise, apply the displacement function along a direction and a speed to a list of coordinates (with map). Determine then the point further from (0,0) (with fold_left).

Exercise 6: Modules You have probably noticed in the previous paragraph that I've prefixed map, iter, etc, with List. That is because these function are declared in the

module List. Modules are in first approximation (very broad and false) the equivalent of files. The only convention of naming is that they must start with an uppercase. To call a function f of module Toto, there are two options: either use `open Toto` earlier in the file and thus it is possible to use names from Toto as if defined in the current file; or simply call them `Toto.f`. The second method is usually preferred, except if a module is used a lot (and even not always) as it allows to avoid conflicts in the name, or to have redefinition problem. It is indeed possible to give the same name to functions in different modules. If you do `open List`, you'll have access to `map`, but if later you do `let map = 3`, then map will design 3, and not the function from List (and vice-versa if you do the converse).

Modules are in reality way richer than that: nothing prevents to have modules inside modules (that is quite widespread). And it even exists ≪functors≫ that create modules parametrised with other modules (one of the most useful being probably Map[4] which is broadly a hash table and allows to represent a lot of stuff: as long as you have an order on the tree, you can index the table by whatever you want). We won't need all of this in the course (in what we give you, we use them, but it should be transparent for you). However if you want to use OCaml for other programs, the most versatile modules are in my opinion List, Map and String.

[4] `https://ocaml.org/api/Map.html`