CS 546

Lectures 2-3

Introduction to OCaml

Polyvios Pratikakis

Based on slides by Jeff Foster

History

- ML: Meta Language
  - 1973, University of Edinburg
  - Used to program search tactics in LCF theorem prover
- SML: Standard ML
  - 1990, Princeton University
- OCaml
  - 1996, INRIA

Reading Material

- Main page
  - http://ocaml.org
- Online repository of packages
  - https://opam.ocaml.org/
- O'Reilly book (Translation from French, online)
  - `http://caml.inria.fr/pub/docs/oreilly-book/`

Language Features

- Functional, with imperative and OO elements
- Garbage collection (no `free()`)
- Strongly typed, type-safe
  - No segfaults or pointer bugs
- Type inference
  - The programmer doesn't need to write types, but can
  - Polymorphic types (similar to Java Generics)
- Data types and pattern matching
  - Easy e.g., to write syntax trees

Functional Programming

- Programs are expressions (not instructions)

```
let double x = (x + x);;
double 2;;
let quad x = double (double x);;
```

- Avoid explicit memory management and state
  - Avoid mutable memory (pointers)
- Closer to mathematical functions
- Meaning of an expression does not depend on state
  - Two calls to the same function with the same arguments always return the same result
  - Programs are more predictable
  - Easier to read and understand parts of the program

The OCaml runtime

- Part of most Linux Distributions
- Windows (Visual Studio or CygWin)
- MacOS X (standalone and in fink)
- Source and binaries at `http://www.ocaml.org`
- Run interpreter: `ocaml`

```
# print_string “Hello world!\n”;;
Hello_world!
- : unit = ()
# let x = 10;;
val x : int = 10
#
```

The OCaml runtime

- Part of most Linux Distributions
- Windows (Visual Studio or CygWin)
- MacOS X (standalone and in fink)
- Source and binaries at `http://www.ocaml.org`
- Run interpreter: `ocaml`

```
# print_string “Hello world!\n”;;
Hello_world!
- : unit = ()
# let x = 10;;
val x : int = 10
#
```

Types

Values

Example program

```
print_string “Hello world!\n”;;
                    (* use ;; to end top-level expressions *)
let x = 40;;         (* this is a comment
                       (* and this is nested *) *)
let answer: int =    (* you don't have to use types,
                       but you can *)
  x + 2              (* whitespace, returns are ignored *)
                     (* and empty lines, too *)
;;
print_int answer;;
print_string “\n”;;
```

The OCaml compiler

- Can be compiled, too
  - It is not necessary in source files to end top-level expressions with `;;`
- To compile programs use `ocamlc`
  - Compiles to object (.cmo) and interface (.cmi) files using -c
  - Links to a.out by default
- Compiling the previous example

```
$ ocamlc example1.ml
$ ./a.out
Hello world!
42
$
```

Scoping and `let`

- Used to create local variables
  - `let x = 40 + 2 in exp` means `x` has the value 42 during the evaluation of `exp`
- Scoped:

```
let x = 40 + 2 in exp;;
x;;  (* error, x is out of scope *)
```

- Similarly in C:

```
{ int x = 40 + 2;
  exp;
}
x;  // error, x is out of scope
```

- Omit `in` at the top level to create global variable

```
let x = 42;;  (* in scope from now on *)
x;;  (* this is fine, x is in the scope *)
```

More scoping

- `let` can be nested:

```
let x = 42 in
let y = successor x in
print_int (x + y);;
```

- The innermost binding hides outer scopes:

```
let x = 42 in
let x = successor x in
print_int x;;  (* prints 43, not 42 *)
let x = 1;;
```

- No side effects, immutable memory:

```
let addtox y = x + y;;  (* refers to the x in scope *)
let x = 42;;  (* new variable, hides previous declaration *)
addtox 3;;    (* will compute 4, not 45 *)
```

Other Syntax

- `if e1 then e2 else e3`
  - Evaluate `e1`, if `true` evaluate and return `e2`, otherwise evaluate and return `e3`

- `e1; e2`
  - Evaluate `e1` and ignore the result. Then evaluate and return `e2`
  - Used to *separate* expressions, not to terminate an expression

Basic Types

- Strongly typed language, no casts
- Basic types:
  - `unit`: Type with one value, no data
  - `int`: Integers
  - `float`: Floats
  - `char`: Characters
  - `string`: Strings
  - `bool`: Booleans

Type Inference and Annotations

- To state that `e` has type `t`, write `(e : t)`

```
let (x : int) = 42;;
let (y : int) = “some string” (* type error *);;
let f (x : int) : float = (float_of_int x) *. 3.14;;
let x = (y: int) + (z: int);;
let f x : int = 3 (* it means f returns int *);;
```

- The compiler will infer all types, and verify the annotations (or produce an error message)
  - Very useful for debugging, incremental changes

Function Types

- Function types use the `->` type constructor
  - `float -> int`: Type of a function that takes a float and returns an integer
- Examples:
  - `let successor x = x + 1 (* int -> int *)`
  - `let iszero x =                (* int -> bool *)`

    `if x = 0 then true else false`

Functions

- Use `fun` to define functions and `let` to name them

```
let successor = fun x -> x + 1;;
```

  - Functions are also values, can be given names using `let` expressions
- A simpler notation

```
let successor x = x + 1;;
```

- Functions can take multiple arguments

```
let multiply x y = x * y;;
```

  - This is really a function that returns a function!

```
let multiply = fun x -> fun y -> x * y;;
```

  - This is called Currying (after the logician Haskell Curry)
- Functions can take functions

```
let apply_twice2 f = f (f 2);; (* apply the function f twice on 2
*)
```

- Here `apply_twice2` has type `(int -> int) -> int`, its argument is a function
- What is `apply_twice2 (multiply 3)`?

Currying

- Currying is very common in OCaml programs
- Currying conventions:
  - `->` is right associative

    `int -> int -> int` is the same as

    `int -> (int -> int)`
  - Function application is left associative

    `multiply 2 3` is the same as

    `(multiply 2) 3`
- Use currying up to an arbitrary number of arguments

  ```
  let f x1 x2 x3 x4 x5 ... = ...
  ```
- Enables partial application
- Not slow - compiler smart enough to avoid creating a new function for each argument

Recursive Functions

- The scope of `x` in `let x = e1 in e2` does not include `e1`

```
let fib n =
  if n = 0 then 0
  else if n = 1 then 1
  else fib (n-1) + fib (n-2)
```

  - Error, `fib` is not in scope at the last line!
- Recursive functions have special definition

Recursive Functions (cont'd)

- The scope of `x` in `let rec x = e1 in e2` includes `e1`

```
let rec fib n =
  if n = 0 then 0
  else if n = 1 then 1
  else fib (n-1) + fib (n-2)
```

  - Binds `fib` recursively during its definition
- Can only be used for functions, not values
- Mutually recursive functions connected with `and`

```
let rec f x = if x = 0 then 1 else g (x – 1)
and g x = (f x) * 2;;
```

Tuples

- Tuples group together other data:

  `(1, false)` is a tuple of two values (or a pair)

  `(e1, e2, e3)` constructs a tuple holding the results of `e1`, `e2` and `e3`
- To deconstruct, use `let`

  ```
  let (x, y) = x;; (* where x is a pair *)
  ```
- Tuple types are products of element types:

  ```
  (2, true) : (int * bool)
  ```
- Tuples can be used to group function arguments:

  ```
  let f (x,y) = x + y;;  (* f has type (int * int) -> int *)
  ```
  - Bad practice, results in unnecessary memory allocation at each function call
- Can name tuple types using `type`

  ```
  type triple_int_bool = (int * int * int * bool)
  ```
  - Works for naming types in general

Lists

- The list is the basic data structure in OCaml
- Lists are written `[e1; e2; e3;...]`

```
# [1; 2; 3]
- : int list = [1; 2; 3]
```

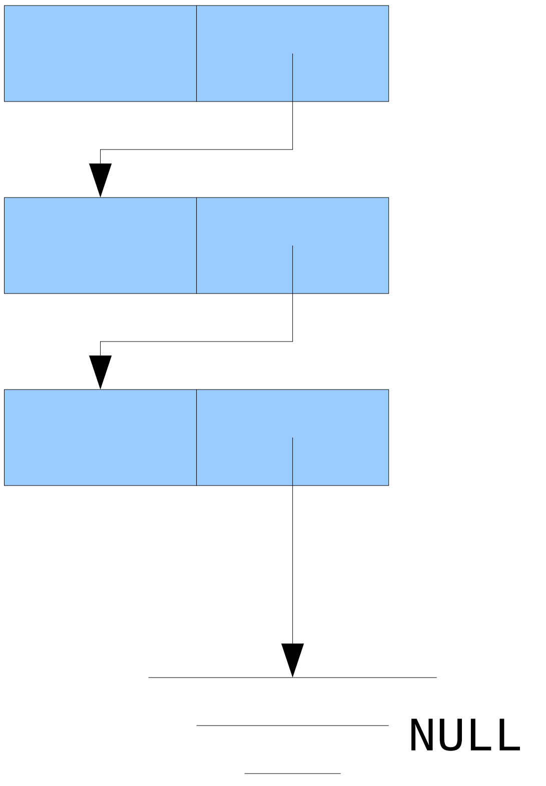
  - What happens when using comma instead of semicolon?
- All the elements in the list must have the same type
  - `list` is a type constructor, meaning `int list` is a list of integers
- `[]` is the empty list

```
# []
- : 'a list = []
```

  - `'a` means any type, `'a list` is a list of any one type

Lists in C

```
struct list {
  int data;
  struct list *next;
};
...
struct list *l;
...
while (l != NULL) {
  l = l->next;
}
```



Lists in OCaml

- The mathematical definition of a list is recursive: a list is either
  - empty: `[]`
  - a pair of an element (the head) and the rest of the list, which is also a `list` of the same type, recursively
- `a::b` is the list that starts with element `a`, and continues with list `b`
  - `::` is a constructor, because it creates a list
  - `a :: b` allocates a list cell, sets its data to `a` and the next pointer to `b`
- Examples:

  `1::[]` is the same as `[1]`

  `1::(2::(3::[]))` is the same as `[1; 2; 3]`

Using Lists with Pattern Matching

- To read the contents of a list `e`, use the match construct

  ```
  match e with p1 -> e1 | p2 -> e2 | ... | pn -> en
  ```

- Patterns `p1`,...,`pn` use `[]`, `::` and *pattern variables* to describe a list structure
- `match` tests each pattern that matches the shape of `e` and evaluates the corresponding expression
  - The pattern variables are bound to the corresponding parts of the structure for the evaluation of the case expression
  - Example:

    ```
    match [1;2] with
      [] -> print_string “impossible\n”
    | (head::tail) -> ... (* head is 1, tail is [2] *)
    ```

- The underscore pattern `_` matches anything and does not bind it
- Compiler warns if the patterns do not cover all cases

Pattern Matching

- `match` can deconstruct tuples too

  `match (1, 3.14, true) with (x, y, z) -> ...`

- Conversely, `let` can pattern match lists

  `let hd::_ = [1;2;3] in ...`

  - Compiler will warn about uncovered cases

- Example pattern matching
  - ```
    let [x; y; z;] = e in ...  (* produces warning about
                                  lists with !=3 elements *)
    ```
  - ```
    match e with
      [] -> ...
    | (x,y)::(_,z)::_::_ -> ...
    | _ -> ...
    ```
  - ```
    let list_function = fun (hd::tl) -> …
        (* produces a warning about [] case *)
    ```
    ```

Polymorphic Types

- Some of the above functions require specific list types
  - `# let add_first_two (x::y::_) = x + y;;`
    `val add_first_two : int list -> int = <fun>`
- Others work on any kind of list
  - `let hd (h::_) = h`
  - `hd [1; 2; 3]         (* returns 1 *)`
  - `hd ['a'; 'b'; 'c']  (* returns 'a' *)`
- Polymorphic types describe such functions
  - hd : 'a list -> 'a
  - `'a` is a type variable
  - means that hd takes a list of any type `'a`, and returns a value of that type

Example Polymorphic Functions

- ```
  # let swap (x, y) = (y, x);;
  val swap : 'a * 'b -> 'b * 'a = <fun>
  ```

- ```
  # let tl (_::t) = t;;
  val tl : 'a list -> 'a list = <fun>
  ```

- ```
  # let fst (x, y) = x;;
  val fst : 'a * 'b -> 'a = <fun>
  ```

- ```
  # let inc_fst (x, y) = (x + 1, y);;
  val inc_fst : int * 'a -> int * 'a = <fun>
  ```

Looping with Recursion

- The only way to iterate
  - for this class at least
- Example: print sequence

```
let rec print_seq start finish =
  print_int start; print_string “\n”;
  if start < finish
  then print_seq (start + 1) finish
```

  - `else` clause can be omitted when type is `unit`

Recursive List Traversal

- List are recursively defined
  - Functions on list are also usually recursive

```
let rec count l = match l with
  [] -> 0
| (_::t) -> 1 + (count t)
```

- Resembles induction in mathematics
  - Base case: the empty list
  - Inductive case: construct the solution for the whole list by reducing to the solution of the tail
  - Called *inductive definition*
- What is the type of `count`?

Recursive Examples

- `let rec sum l = match l with`
```
  [] -> 0
| (hd::tl) -> hd + (sum tl)
```

- `let rec project_first = function`
```
  [] -> []
| (a,_)::tl -> a::(project_first
tl)
```

Recursive Examples (cont'd)

- ```
  let rec list_append l1 l2 = match l1 with
     [] -> l2
   | (hd::tl) -> hd::(list_append tl l2)
  ```

- ```
  let rec list_reverse = function
     [] -> []
   | (hd::tl) -> list_append (list_reverse tl) [hd]
  ```

- `list_reverse` takes O(n^2)!

Recursive Examples (cont'd)

- ```
  let list_reverse l =
    let rec rev r = function
      [] -> r
    | (hd::tl) -> rev (hd::r) tl
    in
    rev [] l
  ```
- Example execution:

  `list_reverse [1; 2; 3]` calls

  `rev [] [1; 2; 3]` which calls

  `rev [1] [2; 3]` which calls

  `rev [2; 1] [3]` which calls

  `rev [3; 2; 1] []` which returns `[3; 2; 1]`
  ```

Recursive Examples (cont'd)

- ```
  let rec list_gt n = function
    [] -> []
  | (hd::tl) ->
    if hd > n then hd::(list_gt n tl) else list_gt n tl
  ```
- Can you think of a better way, like with list_reverse?
- ```
  let list_gt n =
    let helper r = function
      [] -> r
    | (hd::tl) ->
      if hd > n then helper (hd::r) tl
      else helper r tl
    in helper []
  ```

Tail recursion

- Every recursive call is the last thing that happens
- The compiler can optimize
  - Reuse local variables
  - Avoid new stack frame
  - Amounts to a `for` loop

Higher order functions

- So far all recursive functions walk through the list
  - do something to every element, or
  - compute something of every element
- Remember: in OCaml, functions are values
  - We can pass functions as arguments
- Let's try to separate the recursion from the action on each element
- Write a function that takes another function, and a list, applies it to every element, and returns a list of all the results
- What is its type?

The `map` function

- ```
  let rec map f = function
     [] -> []
   | (hd::tl) -> (f hd)::(map f tl)
  ```
- Can it be tail recursive?
- How about `map_rev` that returns a reversed list of the results?
- Examples:

```
let double x = x + x
let is_zero x = (x = 0)
map double [1; 2; 3; 4]
map is_zero [0; 2; 1; 0]
map (fun (x,_) -> x)  (* what is the type of this? *)
```
```

The `fold` function

- Compute an aggregate on every element of a list
  - Need to keep track of the results so far

```
let rec fold f a = function
    [] -> a
  | h::t -> fold f (f a h) t
```

- `a` is the “accumulator”
  - used to hold the intermediate result
- For a list `[e1; ...; en]`, `fold` computes

```
f (...(f a e1) ...) en)
```

- What is its type?

`fold` examples

- `fold (fun a x -> a + x)`
- `fold (fun a _ -> a + 1)`
- `fold (fun a x -> x::a)`

Data Types

- Like C unions, only safe
  - Use tag, or *label*, to identify the “case” of the union

```
type number =
  Zero
| Integer of int
| Real of float
| Complex of float * float
```

- Labels like Real or Integer above, are *type constructors*
  - Functions that take a type and return a type
  - Not first class in OCaml

Data Types (cont'd)

- Use constructors to make values

```
let pi: number = (Real 3.14159)
let one = (Integer 1)
let i = Complex (0.0, 1.0)
```

- What is the type of `[Zero; Real 1.0; pi]` ?
- Deconstruct data types using match, cases differentiate on constructor

```
match n: number with
  Zero -> print_string “nada\n”
| Integer i -> print_int i; print_string “\n”
| Complex (real, 0.0) -> print_string “not too complex\n”
| _ -> print_string “uninteresting\n”
```

- Constructors must start with a capital letter

Data Types (cont'd)

- Examples of data types

```
type optional_int =
  None
| Some of int

let set_or_add n = function
  None -> Some n
| Some n' -> Some (n + n')
```

- Arity: how many arguments a constructor takes
  - `None` : nullary constructor (arity is zero)
  - `Some` : unary constructor (arity is one)

Polymorphic Data Types

- A data type that can be parameterized by another type
  - ```
    type 'a option =
      None
    | Some of 'a
    ```
  - ```
    let x : string option =
      if ... then Some “result” else None
    ```
  - ```
    let rec find_in_list (f: 'a -> bool) = function
      [] -> None
    | h::tl ->
        if (f h) then Some h else find_in_list f tl
    ```
- What is the type of `find_in_list`?
- Option type is built-in in OCaml (like lists)
    ```

Recursive Data Types

- A constructor can refer to the type name
  - `type t = ...` is like `let rec`, only for types
- Lists, using data types:

```
type 'a list =
  Nil
| Cons of 'a * 'a list

let rec length = function
  Nil -> 0
| Cons (_, tl) -> 1 + (length tl)
```

- Works similarly with other kinds of types

```
type 'a pair = 'a * 'a
```

Recursive Data Types (cont'd)

- Examples

```
type ('a, 'b) alt_list =
    Nil
  | Cons of 'a * ('b, 'a) alt_list

type 'a bintree =
    Empty
  | Node of 'a * 'a bintree * 'a bintree
```

- `map : ('a -> 'b) -> 'a bintree -> 'b bintree`
- Data types are handy in writing language ASTs!

Exceptions

```
exception No_such_element of int
exception Found_it of int

let rec contains n l = match l with
  [] -> raise (No_such_element n)
| h::tl -> if h=n then raise (Found_it n)
  else contains n tl

let lookup n l =
  try
    contains n l
  with No_such_element i -> None
    |  Found_it n -> Some n
```

Exceptions (cont'd)

- Declared using `exception`
- Work like normal data type constructors
- Can take arguments (or not)
- Raise exceptions with `raise`
- Catch exceptions using `try ... with ...`
- Pattern matching works normally under `with`
- When not caught by any pattern
  - Propagate upwards in the stack
  - Until the first `with` that matches
  - Uncaught exceptions at the top-level end the program

Functional Programming

- So far, no way to change memory
  - Can only create new data that never change
  - Each function returns its result
    - e.g., a new list with the changes, the old list is the same
- Easier to program
  - Aliasing does not matter
  - Easy to reuse data in memory without actually making copies
  - Functions are predictable, do not depend on outside state

Imperative Programming

- OCaml has pointers, mutable state and side-effects
- Create a pointer (alloc)

```
ref : 'a -> 'a ref
```

- Read the data from a pointer (dereference)

```
! : 'a ref -> 'a
```

- Write a new value to a pointer (update)

```
:= : 'a ref -> 'a -> unit
```

- Example

```
let x = ref 1
let y = !x
let z = x
x := 42  (* y is 1, !z is 42 *)
```

Functions with State

- Create unique numbers

```
let next =
  let counter = ref 1 in
  fun () ->
    let i = !counter in
    counter := i + 1;
    i
let x = next ();; (* x = 1, !counter = 2 *)
let y = next ();; (* y = 2, !counter = 3 *)
```

No NULL

- Memory allocation requires an initial value
  - `ref 1` to allocate an int, etc
- No null-pointer errors
- Sometimes NULL is useful
  - Use option types when `None` is a possible result
    - Benefit: the compiler will force you to check for `NULL` (`None`)
  - Alternatively, use a dummy value as a placeholder
- Example

```
let x = ref “dummy”
(* later in the program, after all initialization *)
x := “real value”
```

Modules

- Good software engineering practice: break code into relevant parts, isolated from each other
- OCaml modules
  - Similar to Java packages, class files
  - Have interface (can be used for information hiding)
  - Types checked by compiler (and linker)
- For examples, look into OCaml standard library

  `http://caml.inria.fr/pub/docs/manual-ocaml/libref/index.html`

Modules (cont'd)

```
module Numbers = struct
    type number =
        Zero
      | Integer of int
      | Real of float
      | Complex of float * float
    let pi = Real 3.14159
    let add_one = function
        Zero -> Integer 1
      | Integer n -> Integer (n+1)
      | Real r -> Real (r .+ 1.0)
      | Complex (r,i) -> Complex (r .+ 1.0, i)
  end;;
Zero;; (* not defined *)
Numbers.Zero (* defined with type number *)
open Numbers;; (* import module names into current scope *)
Zero;; (* defined *)
```

Module Signatures

- The interface of a module
  - Can hide implementation details
- Example

```
module type NUM =
  sig
    type number
    val add_one : number -> number
  end;;

module Number : NUM = struct
  type number = ...
  let add_one = ...
  let helper_function = ...
end;;
```

Modules and Files

- Each file is a module:
  - foo.ml is module Foo (without the `struct`...`end`)
  - foo.mli is the signature (without the `sig`...`end`)
  - The files must have the same name
- Compilation order matters
  - Compiler must compile modules in order
  - Cannot refer to a module later in the compilation order
    - Except recursive modules, which have to be in the same file

Functors

- A function that takes a module and returns a module
  - E.g. Set in the standard library

    ```
    module Strset = Set.Make(String)
    ```
- Signature of the formal argument must match signature of the actual argument
  - Like what happens with function argument types

Lazy Evaluation

- Defer evaluating an expression until it is necessary
- Create a lazy value

```
let x = Lazy.lazy (e)
```

- Read the result (might evaluate the expression)

```
let y = Lazy.force x
```

- Subsequent `force x` return the value computed at the first `force`

Lazy Fibonacci

```
type 'a inf_list = Cons of 'a * 'a inf_list lazy_t

let fiblist : int inf_list =
  let rec build prevprev prev =
    Cons(prevprev,
             Lazy.lazy (build prev (prevprev+prev))
  in build 0 1

let rec nth (l: 'a inf_list) (n: int) : 'a =
  match (l, n) with
    (_, n) when n < 0 -> invalid_arg “negative index”
  | (Cons(x, _), 0) -> x        (* if n = 0 we are at the nth *)
  | (Cons(_, t), n) -> nth (Lazy.force t) (n-1)
```