Lecture: Functional Programming

This course is an introduction to the *mathematical foundations of programming languages* and the *implementation of programming languages and language-based tools.* We use OCaml for most of our programming, because it has certain key features that make it easy to implement other languages and language-based tools.

Although OCaml is a sophisticated language, we will only use a tiny sliver of OCaml in this course, which we will introduce in these notes. There are several OCaml tutorials available online[1]. In addition, the book *Real World OCaml*, which is also freely available online[2], covers the language and software engineering in OCaml in great depth.

1 Introduction to OCaml

This section briefly describes the essential features of OCaml that you need for this course. Chapters 1—6 of Real World OCaml covers the same material in more detail.

1.1 Basic Expressions

A simple OCaml program is a sequence of *declarations*. The final declaration typically has some effect, such as writing to the screen. E.g.:

```
let x = 10
let y = 20

let _ = printf "The sum is %d\n" (x + y)
```

> **Notation** The underscore (the `_` character) is an anonymous variable that cannot be used. This indicates that we don't care about the value that `printf` produces.

We can declare functions in the following way:

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

let _ = printf "square 7 = %d\n" (square 7)
```

OCaml supports *type inference* so the annotations are not necessary:

```
let square x = x * x
```

However, it is good practice to write types. Type annotations document the program and lead to better error messages when things go wrong.

To write a recursive function, you need to write `let rec` instead of `let`:

```
let rec factorial (n : int) : int =
  if n == 0 then 1 else n * factorial (n - 1)

let _ = printf "factorial 5 = %d\n" (factorial 5)
```

[1] https://ocaml.org/learn/tutorials/
[2] http://www.realworldocaml.org

1.2 Algebraic Datatypes and Pattern Matching

The following type declaration creates an *algebraic datatype* (or just *type* for short) called `tree`:

```
type tree =
  | Leaf
  | Node of tree * int * tree
```

This type has two *constructors*. The `Leaf` constructor creates an empty tree, thus it has no arguments. The `Node` constructor takes three arguments: the left sub-tree, an `int`-value at the node, and the right sub-tree.

After the type is declared, we can create new trees:

```
let tree1 = Leaf
let tree2 = Node (Leaf, 100, Node (Leaf, 200, Leaf))
let tree3 = Node (tree2, 500, Leaf)
```

We can use *pattern matching* to extract components of a tree:

```
let left_subtree (atree : tree) = match atree with
  | Leaf -> failwith "got a Leaf"
  | Node (lhs, _, _) -> lhs

let left_of_right_subtree (atree : tree) = match atree with
  | Node (_, _, Node (lrhs, _, _)) -> lrhs
  | _ -> failwith "got a Leaf or a Node (_, _, Leaf)"
```

Pattern matching is particularly useful for writing tree-processing functions:

```
(* Sums all the numbers in a tree *)
let rec tree_sum (atree : tree) : int = match atree with
  | Leaf -> 0
  | Node (lhs, n, rhs) -> tree_sum lhs + n + tree_sum rhs

(* [contains n atree] checks if [n] is contained in [atree], assuming
   the tree is a binary-search tree. *)
let rec contains (n : int) (atree : tree) : bool = match atree with
  | Leaf -> false
  | Node (lhs, m, rhs) ->
     if m = n then true
     else if n < m then contains n lhs
     else (* if n > m *) contains n rhs
```

1.3 Polymorphism

The `tree` type defined above can only store integers. We can define a *polymorphic type* that is parameterized over the type of value stored in the tree:

```
type 'a tree =
  | Leaf
  | Node of tree * 'a * tree

let rec tree_size (atree : 'a tree) : int = match atree with
  | Leaf -> 1
  | Node (lhs, n, rhs) -> tree_size lhs + 1 + tree_size rhs
```

1.4 Lists

We can define a type for (singly-linked) lists in the following way:

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

Given this type, we can define list values and list processing functions:

```
let alist = Cons (10, Cons (20, Nil))

let rec mylength (alist : 'a mylist) = match alist with
  | Nil -> 0
  | Cons (_, tail) -> 1 + mylength tail
```

```
let rec remove_odd (alist : int list) =
  match alist with
    | [] -> []
    | head :: tail ->
      if head % 2 = 0 then
        head :: remove_odd tail
      else
        remove_odd tail
```

(a) Removes odd numbers from a list

```
let rec remove_false (alist : bool list) =
  match alist with
    | [] -> []
    | head :: tail ->
      if head = true then
        head :: remove_false tail
      else
        remove_false tail
```

(b) Removes `false` from a list

Figure 1.1: These functions have a similar shape.

```
let is_even (n : int) : bool = n % 2 = 0

let rec remove_odd (alist : int list) =
  match alist with
    | [] -> []
    | head :: tail ->
      if is_even head then
        head :: remove_odd tail
      else
        remove_odd tail
```

(a) Removes odd numbers from a list

```
let is_true (b : bool) : bool = b

let rec remove_false (alist : bool list) =
  match alist with
    | [] -> []
    | head :: tail ->
      if is_true head then
        head :: remove_false tail
      else
        remove_false tail
```

(b) Removes `false` from a list

Figure 1.2: Both functions apply a predicate to the head of a list

Above, the constructor `Nil` represents the empty list and the constructor `Cons (head, tail)` represents a list that has `head` as the first element and `tail` as the rest of the list. The example list, `alist` has two elements: `10`, followed by `20`, which is followed by the empty list.

However, there is no need to define a list type ourselves. OCaml has a built-in type called `list`:

```
let ocamllist = 10 :: 20 :: []

let rec length (alist : 'a list) = match alist with
  | [] -> 0
  | _ :: tail -> 1 + length tail
```

Apart from the terse notation, there is nothing special about the builtin type. The symbols `::` and `[]` are equivalent to `Cons` and `Nil` in the definition above. They are merely a convenient shorthand for a common data structure. In fact, OCaml lets you write `[10; 20]` to define the same list, but it is just shorthand for `10 :: 20 :: []`.

1.5 Higher-Order Functions

Filter Figure 1.1 shows two functions, where the first removes odd numbers from a list and the second removes the `false` value from lists. Both functions have the same basic structure, which becomes even more obvious if we rewrite them as shown in fig. 1.2. In this figure, both functions apply a predicate to the head of the list. The only difference between them is the type of element in the list and the particular predicate that is applied. Instead of writing two functions that are almost identical, we can write a single function that takes the predicate to be applied as an argument:

```
let rec filter (pred : 'a -> bool) (alist : 'a list) : 'a list =
  match alist with
    | [] -> []
    | head :: tail ->
      if pred head then
        head :: filter pred tail
      else
        filter pred tail
```

With this definition, the two functions simply become:

```
let remove_odd lst = filter is_even lst
let remove_false lst = filter is_true lst
```

Filter is an example of a *higher-order function*, which is a function that takes other functions as arguments.

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

(a) Add all numbers in a list

```
let rec product (lst : int list) : int =
  match lst with
    | [] -> 1
    | hd :: tl -> hd * product tl
```

(b) Multiply all numbers in a list

```
let rec cat (lst : string list) : string =
  match lst with
    | [] -> ""
    | hd :: tl -> hd ^ cat tl
```

(c) Concatenate all strings in a list

```
let rec tot_len (lst : string list) : int =
  match lst with
    | [] -> 0
    | hd :: tl -> String.length hd + tot_len tl
```

(d) Sum the lengths of all strings in a list

Figure 1.3: Several functions that *fold* over a list.

Map Another common higher-order functions is `map`, which applies a function to every element in a list and produces the list of results:

```
let rec map (f : 'a -> 'b) (lst : 'a list) : 'b list = match lst with
  | [] -> []
  | hd :: tl -> f hd :: map f tl

(* Examples *)
let incr x = x + 1
let increment_list (lst : int list) : int list = map incr lst
let lengths (lst : string list) : int list = map String.length lst
```

Take The `take` function returns the prefix of a list that matches a predicate:

```
let rec take (f : 'a -> bool) (lst : 'a list) : 'a list = match lst with
  | [] -> []
  | hd :: tl -> if f hd then hd :: take f tl else []
```

Drop The `drop` function returns a tail of the list, starting with the first element that does not match a predicate:

```
let rec drop (f : 'a -> bool) (lst : 'a list) : 'a list = match lst with
  | [] -> []
  | hd :: tl -> if f hd then drop f tl else lst
```

The built-in operator `@` concatenates two lists. We can relate `take` and `drop` as follows:

```
take drop alist @ drop pred alist = alist
```

This identity holds for all predicates `pred` and all lists `alist`.

Fold The four functions in fig. 1.3 are examples of *folding functions*. A folding function "collapses" a list into a single value. A fold is characterized by a binary operation used to combine values and a base case for the empty list.

> **Notation** The binary `^` operator concatenates two strings.

- In `sum`, the binary operation is `+` and the base value is `0`:

  ```
  sum [10, 20, 30] = 10 + (20 + (30 + 0))
  ```

- In `product`, the binary operation is `*` and the base value is `1`:

  ```
  product [2, 3, 4] = 2 * (3 * (4 * 1))
  ```

- In `cat`, the binary operation is `^` and the base value is `""`:

  ```
  cat ["X", "Y", "Z"] = "X" ^ ("Y" ^ ("Z" ^ ""))
  ```

- In `tot_len`, the binary operation is the following function:

  ```
  let binop (hd : string) (tail_len : int) : int = String.length hd + tail_len
  ```

  and the base value is 0:

  ```
  tot_len ["hi", "cs631", "student"] = binop "hi" (binop "cs631" (binop "student" 0))
  ```

Now that we've seen the common pattern, it's clear that the higher-order function needs to take three arguments (1) the original list, (2) a two-argument function instead of a fixed binary operator, and (3) a value for the base-case, instead of a fixed base-case:

```
let rec fold binop base_case lst =
  match lst with
    | [] -> base_case
    | hd :: tl -> binop hd (fold bin_op base_case tl)
```

We can rewrite the functions in fig. 1.3 more succinctly as follows:

```
let sum (lst : int list) : int = fold (+) 0 lst

let product (lst : int list) : int = fold (*) 1 lst

let cat (lst : string list) : string = fold (^) "" lst

let tot_len (lst : string list) : int = fold binop 0 lst
```

Notation In OCaml, any binary operator can be turned into a two-argument function by enclosing it in parenthesis:

```
(+) 10 20 = 10 + 20
(^) "hello " "world" = "hello " ^ "world"
```

Note that in `binop`, the types of the two arguments are different. The type of the first argument is the same as the type of the elements in the list. The type of the second argument is the same as the type of the result of the function. This is because when `binop` is applied (internally within `fold`), the first argument to `binop` is an element of the list, and the second argument is an intermediate result. Therefore, the type of `fold` is:

```
val fold : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b
```

Map is a Fold The examples above fold a list into a simple, flat value. However, it is possible to fold a list into a different data structure. Any computation that transforms a list $[x; y; z]$ into $f\ x\ (f\ y\ (f\ z\ b))$, where f is a binary operator and b is the base-case value can be written as a fold.

For example, the following function increments an integer n and conses the result to a list of integers:

```
let inc_cons (n : int) (rest : int list) : int list =
  (n + 1) :: rest
```

To increment a list of integers, we can use `inc_cons` and `fold` as follows:

```
let inc_list (lst : int list) : int list =
  fold inc_cons [] lst
```

Here is an example of the function being applied:

```
    inc_list [10; 20; 30]
=   inc_cons 10 (inc_cons 20 (inc_cons 30 []))
=   11 :: (21 :: (31 :: []))
```

Similarly, the following function calculates the length of a string and conses the length to a list of integers:

```
let len_cons (str : string) (rest : int list) : int list =
  String.length n :: rest
```

We can use this function to write a function that consumes a list of strings and produces a list of their lengths:

```
let string_lengths (lst : string list) : int list =
  fold len_cons [] lst
```

```
let rec range (bounds : int * int) : int list =
  match fst bounds = snd bounds with
    | true -> []
    | false -> (fst bounds) :: range (fst bounds + 1, snd bounds)
```

(a) Produces a range of integers.

```
let rec bits (n : int) : bool list =
  match n = 0 with
    | true -> []
    | false -> (n % 2 = 1) :: bits (n / 2)
```

(b) Produces a list of bits that represent a number.

Figure 1.4: Unfolding functions

Here is an example of this function being applied:

```
    string_lengths ["Hello"; "dear"; "reader"]
=   len_cons "Hello"(len_cons "dear"(len_cons "reader"[]))
=   5 :: (4 :: (6 :: []))
```

However, it is easier to write these functions using `map`:

```
let inc (n : int) = n + 1

let inc_lst (lst : int list) : int list = map inc lst

let string_lengths (lst : string list) : int list = map String.length lst
```

In fact, `map` itself can be written using `fold`. But, we're not going to cover it here. It is a part of the homework assignment.

Unfolding A folding function collapses a list into a value. Conversely, an unfolding function, such as those in fig. 1.4, builds a list from a sequence of values and terminates the list when a condition becomes true.

For example, here is how `bits 5` evaluates:

```
=   bits 5
=   5 % 2 = 1 :: bits (5 / 2)
=   true :: bits 2
=   true :: (2 % 2 = 1 :: bits (2 / 2))
=   true :: (false :: bits 1)
=   true :: (false :: (1 % 2 = 1 :: bits 0))
=   true :: (false :: (true :: []))
```

Although `bits` and `range` have a similar shape, they are different in two ways:

1. While `bits` produces the empty list when `n = 0`, `range` produces the empty list when the condition `fst bounds = snd bounds` holds.

2. When `bits` recurs, it conses `n % 2 == 1` onto the list and recurs with `n / 2` as the argument in the recursive call. On the other hand, when `range` recurs, it conses `fst bounds` onto the list and recurs with `(fst bounds + 1, snd bounds)` as the argument in the recursive call.

The higher-order function `unfold` takes two functions as arguments to account for these two differences:

```
let rec unfold (pred : 'b -> bool) (gen : 'b -> 'a * 'b) (v : 'b) : 'a list =
  match pred v with
    | true -> []
    | false ->
      let (x, v') = gen v in
      x :: unfold pred gen v'

let range_pred (bounds : int * int) : bool = fst bounds = snd bounds
let range_gen (bounds : int * int) : int * (int * int) =
  (fst bounds, (fst bounds + 1, snd bounds))
let range (bounds : int * int) : int list = unfold range_pred range_gen bounds

let bits_pred (n : int) : bool = n = 0
let bits_gen (n : int) : bool * int = (n % 2 == 1, n / 2)
let bits (n : int) : bool list = unfold bits_pred bits_gen n
```

Perspective The functions `map`, `filter`, and `fold` are common higher-order functions and are present in most modern programming languages. You can also find them in the guise of *list comprehensions*. For example, the following Python ist comprehension maps the increment function over a list:

```
 >>> [x + 1 for x in [10, 11, 12]]
[11, 12, 13]
```

The following comprehension filters a list:

```
 >>> [x for x in [1, 6, 2, 7] if x > 5]
[6, 7]
```

2 Testing and Compiling

Printing strings is a very poor way to test code. You'll do better by writing test cases as follows:

```
let%TEST "factorial 5" = factorial 5 = 120

let%TEST "factorial 0" = factorial 0 = 1
```

The `TEST` notation is not standard OCaml, but a handy *extension* that we use extensively in this course.