section-02

October 12, 2021

1 Section 2

1.1 Agenda

- Review:
  - Algebraic data types
  - Let-expressions
  - Currying
  - Higher-order functions
  - Recursion on trees
- Assignment 1 Q&A

[1]:
```
let hole () = failwith "todo"
;;
```

1.2 Algebraic Data Types

Consider the data types defined in Q2:

[2]:
```
(* Problem 2 *)
type binop = Add | Sub | Mul | Div

type expr = Const of int
          | Binary of binop * expr * expr
;;
```

- Both `binop` and `expr` are what's called *algebraic data types*, because they are constructed from sum types (`|`) and produce types (`*`), like how polynomials are constructed in mathematics (e.g. `f(x,y) = x*x + x*y + y*y*y`)
- Each case in a sum type is called a *variant*. The labels (e.g. `Const`, `Binary`, `Add`, etc.) are called *constructors*.
- Constructor are like functions!
  - `Const` is like a one-argument function with type `int -> expr`
  - `Binary` is like a three-argument function with type `binop * expr * expr -> expr`
  - `Add`, `Sub`, `Mul`, `Div` are like zero-argument functions of type `binop`, which are just constants!

1.3 Let

There are two kinds of `let` in OCaml:

1.3.1 I. Top-level let-bindings

[3]:
```
(* Problem 1 *)

let insert (k: 'k) (v: 'v) (al: ('k * 'v) list) : ('k * 'v) list =
  (k,v) :: al

let rec lookup_opt (k: 'k) (al: ('k * 'v) list) : 'v option =
  match al with
  | _ -> failwith "Not yet implemented" (* your code here *)


(* Uncomment to test your solution *)
let al = insert "x" 3 (insert "y" 2 (insert "x" 1 []))
(* al is now [("x", 3), ("y", 2), ("x", 1)] *)

(* let _ = assert (lookup_opt "z" al = None)
let _ = assert (lookup_opt "y" al = Some 2)
let _ = assert (lookup_opt "x" al = Some 3) *)
;;
```

In general, an OCaml program is a sequence of top-level `let` bindings (and `type` declarations)

```
let a = ...
let rec f x = ...
and g x y = ...
let b = ...
let _ = ...
let _ = ...
...
```

1.3.2 II. Let-expressions, for creating local variables

Common scenario: 1. Compute 2. Store temporary result 3. Use the temporary result in subsequent computations

In OCaml,

```
let tmp = computation () in more_computation tmp
```

More generally,

```
let <name> = <e> in <body>
```

To evaluate a let-expression: 1. Evaluate `e` 2. Bind the value of `e` to `name` 3. Evaluate `body`, where `name` is available 4. **The value of `body` becomes the value of the overall let-expression**

Exercise: What is the value of the following expression?

[4]:

```
(let x = 1 in x + 2) + 3
;;
```

You can nest let-expressions!

Exercise: What is the value of the following expression?

[5]:

```
let x = 1 in
let y = x * 2 in
x + y
;;
```

The left-hand side of `let` can actually be a pattern!

Q3 starter code:

[6]:

```
(* Problem 3 *)
type binop = Add | Sub | Mul | Div
type expr = Const of int
          | Binary of binop * expr * expr
          | Id of string                (* new *)
          | Let of string * expr        (* new *)
          | Seq of expr list            (* new *)


type environment = (string * int) list

let rec interpret (e: expr) : int =
  let empty = [] in
  let _, n = (interpret' e empty) in n

and interpret' (e: expr) (env: environment) : (environment * int) =
  match e with
  | _ -> failwith "Not yet implemented" (* your code here *)
;;
```

Pay attention to the line

```
let _, n = interpret' e empty in n
```

Here,

- `interpret'` returns an `(environment * int)` pair
- But `interpret` doesn't need the returned environment, so it matches the pair with the pattern `_, n`
  - `_` matches the returned environment but throws it away

- The return value of `interpret` is `n`

Exercise: Replace the second line (`let _, n = interpret' e empty in n`) of `interpret` with a pattern match.

[7]:
```
let rec interpret (e: expr) : int =
  let empty = [] in
  (*   Solution   *)
  match (interpret' e empty) with
  | (_, n) -> n
;;
```

1.4 Currying

Let us define the exponentiation function.

Question: Without running the snippet below, tell us the type of `exp`.

[8]:
```
let rec exp base power =
  if power = 0 then
    1
  else
    base * exp base (power-1)
;;
```

Now consider an alternative definition, `exp'`.

Question: What is the type of `exp'`?

[9]:
```
let rec exp' pair =
  let base, power = pair in
  if power = 0 then
    1
  else
    base * exp' (base, (power-1))
;;
```

Note that `->` associates to the right, so the type of the curried version is actually `'a -> ('b -> 'c)`.

For any "two-argument" function (from arguments `'a` and `'b` to result `'c`), it really has two incarnations in OCaml:

1. Uncurried: `('a * 'b) -> 'c`
2. Curried: `'a -> 'b -> 'c`

Thus, both the curried and the uncurried versions are technically one-argument functions!

1. `('a * 'b) -> 'c` takes some input, and returns `'c`. The input has type `'a * 'b`
2. `'a -> ('b -> 'c)` takes some input, and returns `'b -> 'c`. The input has type `'a`. The output is not a simple value, but a *function* from `'b` to `'c`.

Exercise: Implement:

1. a (higher-order) function, called `uncurry`, that converts curried functions to their uncurried form, and
2. a function, called `curry`, that converts uncurried functions to their curried form.

What's the type of `uncurry`? What's the type of `curry`?

[10]:
```
(*   Solution   *)

let uncurry f p = let a,b = p in f a b

let uncurry' f (a,b) = f a b

let uncurry'' f = fun (a,b) -> f a b
;;
```

[11]:
```
(*   Solution   *)

let curry f a b = f (a,b)

let curry' f = fun a -> fun b -> f (a,b)

let curry'' f = fun a b -> f (a,b)
;;
```

Consider the types of `uncurry` and `curry`:

If you replace `*` with logical conjunction (`/\`, aka logical AND), and `->` with logical implication (`==>`, aka logical IMPLY), you will get a boolean proposition!

What are the truth values of the resulting propositions?

1.4.1 Interlude: Anonymous Functions

Let's say we want to define our own version of the addition function:

[12]:
```
let plus1 (x: int) (y: int) : int = x + y
;;
```

Without type annotation, this is really just

[13]:
```
let plus2 x y = x + y
;;
```

But OCaml also lets us rewrite above as

[14]:
```
let plus3 = fun x -> fun y -> x + y
;;
```

which has a even nicer syntactic sugar:

[15]:
```
let plus4 = fun x y -> x + y
;;
```

Now the definition of the function is completely separated from its name!

[16]:
```
(fun x y -> x + y) 5 6
;;
```

This is extremely useful when you program with higher-order functions (e.g. `map`, `filter`, `fold`, etc), many of which take a function as input.

But you often just want to construct the input function *ad hoc*, because that particular function won't ever be used again. Anonymous functions thus save you the labor of giving names, when those functions don't deserve a name.

A related note: curried functions are often better than their uncurried couterparts, because the curried version lets you do partial application, which is very useful in combination with higher-order functions.

1.5 Higher-Order Functions

To practice higher-order functions, a good exercise is actually to sit down, and try to write down their types and recursive definitions.

1.5.1 Map

What does `map` do, intuitively?

What is the type of `map`?

Can you write down the recursive definition of `map`?

[17]:
```
List.map
;;
```

Exercise: Fill in the hole to raise 2 to a list of powers:

[18]:
```
let two_to_the ps =
      (*   Solution   *)
      List.map (fun p -> exp 2 p) ps
;;
```

[19]:
```
assert (two_to_the [0; 1; 2; 3; 4] = [1; 2; 4; 8; 16])
;;
```

We can simplify the definition of `two_to_the` even further!

[20]:
```
let two_to_the' ps = List.map (exp 2) ps
let two_to_the'' = List.map (exp 2)
;;

assert (two_to_the' [0; 1; 2; 3; 4] = [1; 2; 4; 8; 16])
;;
assert (two_to_the'' [0; 1; 2; 3; 4] = [1; 2; 4; 8; 16])
;;
```

[20]:
```
val two_to_the' : int list -> int list = <fun>
```

[20]:
```
val two_to_the'' : int list -> int list = <fun>
```

[20]:
```
- : unit = ()
```

[20]:
```
- : unit = ()
```

This is an alternative way to think of `map`.

Given any function `f: 'a -> 'b`, `map` promotes `f` into a more powerful function!

The new function `map f` now has the type `['a] -> ['b]`; it operates on the list version of its argument type and return type.

1.5.2 Filter

What does `filter` do, intuitively?

What is the type of `filter`?

Can you write down the recursive definition of `filter`?

[21]:
```
let filter = List.filter
;;
```

[21]:
```
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Exercise: Using `filter`, implement `positives`, which keeps only the positive integers in a list.

[22]:
```
(* val positives: int list -> int list *)
let positives xs =
    (*   Solution   *)
    filter (fun x -> x > 0) xs
;;
```

[22]: val positives : int list -> int list = <fun>

[23]:
```
assert (positives [2; -3; 5; -7; 11] = [2; 5; 11])
;;
```

[23]: - : unit = ()

Exercise: Using `filter`, implement `partition`, which takes an input predicate and partitions a list into two lists, where elements of the first list satisifies the predicate while those of the second list does not.

[24]:
```
(* val partition : ('a -> bool) -> 'a list -> 'a list * 'a list *)
let partition pred xs =
    (*   Solution   *)
    (filter (fun x -> pred x) xs, filter (fun x -> not (pred x)) xs)
;;
```

[24]: val partition : ('a -> bool) -> 'a list -> 'a list * 'a list = <fun>

[25]:
```
assert (partition ((<) 0) [2; -3; 5; -7; 11] = ([2; 5; 11], [-3; -7]))
;;
```

[25]: - : unit = ()

1.5.3 Fold

What does `fold` do, intuitively?

What is the type of `fold`?

Can you write down the recursive definition of `fold`?

[26]:
```
let fold = List.fold_left
;;
```

[26]: val fold : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

Exercise: Using `fold`, implement `sum` which sums an integer list.

[27]:
```
(* val sum : int list -> int *)
let sum ns =
    (*   Solution   *)
    fold (+) 0 ns
;;
```

[27]: val sum : int list -> int = <fun>

[28]:
```
assert (sum [1; -1; 2; -2; 3; -3; 4] = 4)
;;
```

[28]: - : unit = ()

Exercise: Using `fold`, implement `rev` which reverses a list.

[29]:
```
(* val rev : 'a list -> 'a list *)
let rev xs =
    (*   Solution   *)
    fold (fun acc x -> x :: acc) [] xs
;;
```

[29]: val rev : 'a list -> 'a list = <fun>

[30]:
```
assert (rev [1;2;3;4;5] = [5;4;3;2;1])
;;
```

[30]: - : unit = ()

Exercise: Using `fold`, implement `is_sorted` which tests if an integer list is sorted.

[31]:
```
let is_sorted (xs: int list) : bool =
    (*   Solution   *)
    let init = (true, Int.min_int) in
    let f (sorted_acc, m) n = (sorted_acc && m <= n, n) in
    let (sorted, _) = fold f init xs in
    sorted
;;
```

[31]: val is_sorted : int list -> bool = <fun>

[32]:
```
assert (is_sorted [1;2;3;4;5])
;;
```

[32]: - : unit = ()

Note that the `fold` we are using is called `fold_left`. So naturally, there is also a `fold_right`!

- Can you guess what `fold_right` does?

- What do you think is the type of `fold_right`?
- Can you write down a recursive definition of `fold_right`?
- One of `fold_right` and `fold_left` is tail-recursive, while the other one is not. Which one is tail-recursive?

1.5.4 Recursion on trees

Exercise:: Implement `nodes`, which counts the number of nodes (i.e. constructors) in an expression.

[33]:
```
let rec nodes (e: expr) : int =
  (*   Solution   *)
  match e with
  | Const m -> 1
  | Id y -> 1
  | Binary (op, e1, e2) -> 1 + nodes e1 + nodes e2
  | Let (y, e) -> 1 + nodes e
  | Seq es -> sum (List.map nodes es)
;;
```

[33]:
```
val nodes : expr -> int = <fun>
```

[34]:
```
assert (nodes (Const 0) = 1)
;;
assert (nodes (Binary (Add, Const 0, Id "x")) = 3)
;;
```

[34]:
```
- : unit = ()
```

[34]:
```
- : unit = ()
```

Exercise:: Implement `subst`, which takes a string `x` and an expression `ex`, substitutes every occurrence of `Id x` with `ex` in an input expression.

[35]:
```
let rec subst (x: string) (ex: expr) (e: expr) : expr =
  (*   Solution   *)
  let recur = subst x ex in
  match e with
  | Const m -> e
  | Id y -> if x = y then ex else e
  | Binary (op, e1, e2) -> Binary (op, recur e1, recur e2)
  | Let (y, e) -> Let (y, recur e)
  | Seq es -> Seq (List.map recur es)
;;
```

[35]:
```
val subst : string -> expr -> expr -> expr = <fun>
```

[36]:
```
assert (subst "x" (Const 1) (Binary (Add, Id "x", Id "x")) = Binary (Add, Const␣
↪1, Const 1))
;;
assert (subst "x" (Id "y") (Seq [Id "z"; Id "y"; Id "x"]) = Seq [Id "z"; Id "y";
↪ Id "y"])
;;
```

[36]:
```
- : unit = ()
```

[36]:
```
- : unit = ()
```

Notice how `nodes` and `subst` look awfully the same: 1. In the base cases, they map the base case to a value of the return type. - For `nodes`, it maps `Const` and `Id` to the integer 1. - For `subst`, it maps `Const` to itself, and `Id` to the substituted expression if there is a match)

2. In the recursive cases, they recurse on sub-expressions, after which they build a value of the return type with the results of recursing on the sub-expressions.
   - For `nodes`, it simply sums the result of recursion on sub-expressions and adds 1.
   - For `subst`, it reconstructs an `expr` using the same constructor and the result of recursion on sub-expressions.

So there's a lot of code repetition here! In particular, both `nodes` and `subst` follows a pattern called *post-order traversal*, which you may have seen in your previous data structure course.

Whenever we see code repetition in OCaml, we should think, "Can we define a higher-order function (like `map` and `fold`) that distills the pattern / template / common-behavior?"

A Very Challenging Exercise: Write a higher-order function that distills the pattern examplified by `nodes` and `subst`. Then define new versions of `nodes` and `subst` in terms of the higher-order function you just wrote.