CSC 330 / Spring 2023

OCaml Scoping

Ibrahim Numanagić /nu-ma-nagg-ich/

based on Dan Grossman's lecture materials

Lexical scope

- **Key concept**: must get **this** for homework and competency
- We already know function bodies can use anything in scope…
  - But now function are getting passed around and/or returned! In "scope"… where?
- ⚠️ Function bodies evaluate in the environment where they were defined! (**lexical scoping** 😇)
  - and **NOT** in the environment where the function is called (**dynamic scoping** 😈)
- *Today*: examples and why lexical scope is "the right thing"
  - Later: how to implement lexical scope

Example

```
1 let x = 1
2 let f y = x + y
3 let x = 2
4 let y = 3
5 let z = f (x + y)
```

- Line 2 defines function `f` which, when called, evaluates `x + y` in an environment where $x \mapsto 1$ and $y \mapsto \text{arg}$
- Call on Line 5:
  - Looks up `f` to get definition from line 2
  - Evaluates argument expression `(x + y)` in *current* environment
  - Calls function with argument `5`, result is `6`

Closures

- How can functions be evaluated in old environments?
  - OCaml stores old environments as needed!
- Refined, *official semantics* of functions:
  - A function value has two parts:
    - The **code**
    - The **environment** from when the function was defined
  - This is a *pair*, but hidden from the programmer (no `fst` or `snd`)
  - All we can do is *call with an argument*
  - This pair is called **a closure**
  - Calls are evaluated in the closure's environment extended by parameter $\mapsto$ argument mapping

Example

```
1 let x = 1
2 let f y = x + y
3 let x = 2
4 let y = 3
5 let z = f (x + y)
```

- Line 2 creates a closure and binds `f` to it
  - **Code**: take `y` and have body `x + y`
  - **Environment**: `x ↦ 1` (plus anything else in scope)
- Line 5 calls the closure with argument `5`, thus assigning `6` to `z`:
  - So body evaluated in environment with `x ↦ 1`, `y ↦ 5`

Lexical scope via closures

- We know the rule: **call evaluates function body in environment where function defined**
  - and the model for implementing **lexical scoping** via *closures*
- So what's left? We already know everything!
  - See some (silly) examples to demonstrate **lexical scope** with higher-order functions
  - Discuss why **dynamic scope** is usually The Bad Idea™
  - See many powerful *idioms* with higher-order functions
    - Example: passing functions to iterators like `filter`

The rule stays the same

- With higher-order functions, our semantics stays the same
  - Function body evaluated in environment where it was defined, extended with `param ⟼ arg`
- Nothing changes when we take or return functions
  - But *the environment* may not be the current **toplevel** environment
- May be unintuitive at first, but makes first-class functions much more powerful!

Note: C++ programmers might relate this to:

```
std::vector<int> a = ...;
auto f = [&](const Foo &foo) { return foo.bar(a); };
```

However, in OCaml you don't need to worry about `[]`, `[&]` or `[=]`!

Returning a function

```
1 let x = 1
2 let f y =
3   let x = y + 1 in
4   fun q -> x + y + q
5 let x = 3
6 let g = f 4
7 let y = 5
8 let z = g 6
```

- Trust the rule: line 6 binds `g` to a closure:
  - **Code**: take `q` and have body `x + y + q`
  - **Environment**: $y \mapsto 4$, $x \mapsto 1$, … (anything else in scope)
    - so this closure will always add 9 to its argument
- Thus line 8 binds `z` to `15`

Passing a function

```
1 let f g =
2   let x = 3 in
3   g 2
4 let x = 4
5 let h y = x + y
6 let z = f h
```

- Trust the rule: line 5 binds `h` to a closure:
  - **Code**: take `y` and have body `x + y`
  - **Environment**: `x ↦ 4`, `f ↦ <closure>`, … (anything else in scope)
    - so this closure will always add 4 to its argument
- Thus line 6 binds `z` to `6`
  - Note: line 2 can't affect anything! You can (and should) safely delete it

Code demonstration

Why lexical scope?

- **Lexical scope**: use environment where function was defined
- **Dynamic scope**: use environment where function is called
- In the early days of PL, folks pondered: *which rule is better?*
  - Experience has shown that lexical scope is *almost always* the right default
- Let's consider three precise, technical reasons why
  - This is not a matter of opinion 🤔

Why lexical scope?

1. Functions meaning does not depend on variable names, only “shape”

```
let f y =
  let x = y + 1 in
  fun q -> x + y + q
```

Example: can change `f` to use `w` instead of `x`

- **Lexical scope**: cannot change behavior of function
- **Dynamic scope**: depends on the environment of the caller

```
let f g =
  let x = 3 in
  g 2
```

Example: can remove unused variables

- **Lexical scope**: variable unused, cannot change behavior
- **Dynamic scope**: some `g` may use `x` and depend on it being `3`

  – Weird 😈

Why lexical scope?

2. Functions can be type checked and reasoned about where they are defined
   - **Example**: **dynamic scope** tries to add string and has unbound variable `y`

OCaml:

```
let x = 1
let f y =
  let x = y + 1 in
  fun q -> x + y + q
let x = "hi"
let g = f 4
let z = g 6
```

Python:

```
x = 1
def f(y):
  global x
  return lambda q: x + y + q
x = "hi"
g = f(4)
g(6)  # TypeError yay!
```

(Yes, I am cheating here a bit with `global`…)

Why lexical scope?

3. Closures can easily store the data they need

```
let rec filter f =
  fun xs ->
    match xs with
    | [] -> []
    | x :: xs' ->
      if f x
      then x :: (filter f) xs'
      else (filter f) xs'
```

```
let greater_than x =
  fun y -> x < y
let is_positive =
  greater_than 0
let only_positives =
  filter is_positive
let all_greater (xs, n) =
  filter (greater_than n) xs
```

Does dynamic scope even exist?

- **Lexical scope** is definitely the right default, seen in most languages
- **Dynamic scope** is occasionally convenient in some situations
  - Allows code to *just bind variables used in another function* to change behavior without passing parameters
    - Can be convenient, but is often a nightmare 😱
  - Some languages (including Racket!) have special "opt-in support", but most do not
  - Python is an offender here
  - It's also easier to implement, therefore…
- If you squint, exception handling is similar to dynamic scope
  - `raise e` jumps to *most recent* (dynamically registered) handler
  - Does not need to be syntactically inside the handler!
  - Fun fact: some places (e.g., Google) outright ban exceptions!

When things evaluate

- We know:
  - Function body not evaluated until call
  - Function body evaluated every time function is called
    - in environment from when when function was defined
  - Variable bindings evaluate right-hand-side once at binding time
    - **NOT** every time variable is used
- With closures, can avoid repeating computations that don't depend on arguments
  - Can make code run faster, but also interesting semantics

Recomputation

```
let all_shorter (xs, s) =
  filter (fun x -> String.length x < String.length s) xs

let all_shorter' xs s =
  let n = String.length s in
  filter (fun x -> String.length x < n) xs
```

- These both work and are equivalent
  - First version computes `String.length s` repeatedly (once per `x` in `xs`)
  - Second version computes `String.length s` once before filtering
    - No new features! Just new use of closures!

Code demonstration

Fold

```
let rec fold_left (f, acc, xs) =
  match xs with
  | [] -> acc
  | x :: xs' -> fold_left (f, f (acc, x), xs')
```

- Another hall-of-fame higher-order function; a.k.a. `fold` or `reduce` (map-reduce idiom, anybody?)
  - Accumulate answer by repeatedly applying `f`: `fold_left (f, acc, [v1; v2; v3])` is basically `f (f (f (acc, v1), v2), v3)`
  - Type: `fold_left : ('a * 'b -> 'a) * 'a * 'b list -> 'a`
- Contrast with `fold_right`: goes through list the other way: `f (v1, f (v2, f (v3, acc)))`

```
let rec fold_right (f, acc, xs) =
  match xs with
  | [] -> acc
  | x :: xs' -> f (x, fold_right (f, acc, xs'))
```

  - Is this tail-recursive?

Iterators

- `map`, `filter` and `fold` are similar to iteration patterns we see in imperative PLs
  - But not built into OCaml: we can easily define them ourselves!
  - Just an idiom: power stems from combining elegant features
- Separate *traversal* from *computation*
  - Allows reusing traversal code
  - Allows reusing computation code
  - Allows common way to talk about traversing data structures
    - Most collections provide `map`, `filter`, `fold` functions that *work the same way*

Code demonstration