CSC 330 / Spring 2023

OCaml
Various idioms

Ibrahim Numanagić /nu-ma-nagg-ich/

based on Dan Grossman's lecture materials

Common idioms

- We know the rules for lexical scope and function closures
  - Now: what is it good for?
- Partial but wide-ranging list:
  - [x] Pass functions with private data to iterators
  - [ ] Currying (multi-argument functions and partial application)
  - [ ] Combine functions (e.g., composition)
  - [ ] Callbacks (e.g., reactive programming)
  - [ ] Implementing an ADT with a record of functions (optional)

Currying

Currying

- Recall every OCaml function takes exactly one argument
- Previously encoded n arguments via one n-tuple
- Another way: Take one argument and return a function that takes another argument and…
  - Called **currying** after famous logician Haskell Curry

Example

```
let sorted3 = fun x -> fun y -> fun z ->
  z >= y && y >= x
let t = ((sorted3 7) 9) 11
```

- Calling `(sorted3 7)` returns a closure with:
  - Code: `fun y -> fun z -> z >= y && y >= x`
  - Environment maps `x` to 7
- Calling *that* closure with 9 returns a closure with:
  - Code `fun z -> z >= y && y >= x`
  - Environment maps `x` to 7, `y` to 9
- Calling *that* closure with `11` returns `true`

Syntactic sugar

```
let sorted3 = fun x -> fun y -> fun z ->
  z >= y && y >= x
let t = ((sorted3 7) 9) 11
let t = sorted3 7 9 11
```

- In general, `e1 e2 e3 e4 ...` means `( ... ((e1 e2) e3) e4)`
  - So instead of `((sorted3 7) 9) 11` we can write `sorted3 7 9 11`
- Callers can just think *multi-argument function with spaces instead of a tuple expression*
  - Different than tupling; caller and callee must use same technique

Syntactic sugar

```
let sorted3 = fun x -> fun y -> fun z ->
  z >= y && y >= x
let sorted3 x y z = z >= y && y >= x
let t = ((sorted3 7) 9) 11
let t = sorted3 7 9 11
```

- In general:

  `let [rec] f p1 p2 p3 ... = e` means `let [rec] f p1 = fn p2 -> fn p3 -> … -> e`

  - So we can just write:

    ```
    let sorted3 x y z = z >= y && y >= x
    ```

- Callees can just think *multi-argument function with spaces instead of a tuple pattern*

  - Different than tupling; caller and callee must use same technique

Final version

```
let sorted3 x y z = z >= y && y >= x
let t = sorted3 7 9 11
```

- As elegant syntactic sugar (even fewer characters than tupling) for:

```
let sorted3 = fun x -> fun y -> fun z ->
  z >= y && y >= x
let t = ((sorted3 7) 9) 11
```

Curried fold

- A more useful example and a call to it (will improve call next)
  - Exactly how ML standard library defines `fold_left`

```
let rec fold_left f acc xs =
  match xs with
  | [] -> acc
  | x :: xs' -> fold_left f (f acc x) xs'
let sum_bad_style xs =
  fold_left (fun acc x -> acc + x) 0 xs
```

More unnecessary function wrapping

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

let sum_bad_style xs =
  fold_left (fun acc x -> acc + x) 0 xs
```

- As we already know, `fold_left (fun acc x -> acc + x) 0` evaluates to a closure that, given `xs`, evaluates the match-expression with `f` bound to `(fun acc x -> acc + x)` and `acc` bound to 0

More unnecessary function wrapping

```
let sum_bad_style xs =
  fold_left (fun acc x -> acc + x) 0 xs

let sum_good_style =
   fold_left (fun acc x -> acc + x) 0
```

- Previously learned not to write `let f x = g x`
- This is the same thing with `fold_left (fun acc x -> acc + x) 0` in place of `g`

“Too few” arguments

- Previously used currying to simulate multiple arguments
- But if caller provides “too few” arguments, we get back a closure *waiting for the remaining arguments*
  - Called *partial application*
  - Convenient and useful (and occasionaly *really* annoying!)
  - Can be done with *any* curried function
- No new semantics here: just a (occasionally) pleasant, general idiom

Iterators redux

- Partial application is particularly nice for iterators
  - Provided iterator implementations *put the function argument first*
  - So that's what libraries do

```
let remove_negs_meh xs = List.filter (fun x -> x >= 0) xs
let remove_negs = List.filter (fun x -> x >= 0)
let remove_all n = List.filter (fun x -> x <> n)
let remove_zeros = remove_all 0
```

In fact…

- General style in OCaml is currying for multiple arguments 🤯
  - Not tupling!
  - Expect to see curried arguments in libraries, even for first-order functions
- It's *weird* that we used tupling for 3 weeks, but wanted to show currying after we could understand its semantics

```
let rec append xs ys =
  match xs with
  | [] -> ys
  | x :: xs' -> x :: append xs' ys
(* 'a list -> 'a list -> 'a list *)
```

Efficiency & convenience

- So which is faster/better: tupling or currying multiple-arguments?
- Both are constant-time operations, so it doesn't matter in most of your code: quite fast
  - Don't program against an *implementation* until it matters!
- For the small (zero?) part where efficiency matters:
  - It turns out OCaml compiles currying more efficiently (it optimizes full application)
  - OCaml could have made the other implementation choice
- More interesting trade-off is convenience:
  - Partial application vs. "compute a tuple to pass"

Value restriction 😐

- If you use partial application to *create a polymorphic function*, it may not work due to the **value restriction**
  - May give it the monomorphic type you first use it with or a strange type error
  - This should surprise you: you did nothing wrong! 😁
    - but you still must change your code! 😞
  - See the accompanying code for workaround: not-so-unnecessary function wrapping
  - Can discuss a bit more when discussing type inference

Code demonstration

More idioms

- We know the rule for lexical scope and function closures
  - Now: what is it good for?
- Partial but wide-ranging list:
  - [x] Pass functions with private data to iterators
  - [x] Currying (multi-argument functions and partial application)
  - [ ] Combine functions (e.g., composition)
  - [ ] Callbacks (e.g., in reactive programming)
  - [ ] Implementing an ADT with a record of functions (optional)

Composition

Combine functions

- Canonical example is function composition:

```
let compose f g = fun x -> f (g x)
```

- Creates a closure that *remembers* what `f` and `g` are bound to
- Type `('b -> 'c) -> ('a -> 'b) -> ('a -> 'c)`
  - But the REPL prints something *equivalent*
- Can do infix (just a style thing, not closure-specific):

```
let (%) = compose
```

Example

- Third version is the “best” with our infix function:

```
let sqrt_of_abs i = sqrt (float_of_int (abs i))
let sqrt_of_abs i = (sqrt % float_of_int % abs) i
let sqrt_of_abs = sqrt % float_of_int % abs
```

Left-to-right or right-to-left

```
let sqrt_of_abs = sqrt % float_of_int % abs
```

- As in math, function composition is "right to left"
  - take absolute value, convert to real, and take square root
  - square root of the conversion to real of absolute value
- **Pipelines** of functions are common in functional programming and many programmers prefer left-to-right
  - Predefined exactly like this in OCaml

```
let (|>) x f = f x
let sqrt_of_abs i = i |> abs |> float_of_int |> sqrt
```

Another example

```
let pipeline_option f g =
  fun x ->
    match f x with
    | None -> None
    | Some y -> g y
```

- As is often the case with higher-order functions, the types hint at what the function does:
  `('a -> 'b option) -> ('b -> 'c option) -> 'a -> 'c option`

More higher-order functions

- What if you want to curry a tupled function or vice-versa?
- What if a function's arguments are in the wrong order for the partial application you want?
- Naturally, it is easy to write higher-order wrapper functions
  - and their types are neat logical formulas that tell us a lot!

```
let curried_of_paired f x y = f (x, y)
let paired_of_curried f (x, y) = f x y
let swap_tupled f (x, y) = f (y, x)
let swap_curried f x y = f y x
```

Code demonstration

More idioms

- We know the rule for lexical scope and function closures
  - Now: what is it good for?
- Partial but wide-ranging list:
  - [x] Pass functions with private data to iterators
  - [x] Currying (multi-argument functions and partial application)
  - [x] Combine functions (e.g., composition)
  - [ ] Callbacks (e.g., in reactive programming)
  - [ ] Implementing an ADT with a record of functions (optional)

Callbacks and references

I lied 🤔! OCaml actually *has* mutation

- Mutable data structures are OK in some situations
  - When *update to state of world* is appropriate model
  - But want most language constructs truly immutable
- OCaml does this with a *separate* construct: **references**
- Introducing now because will use them for next closure idiom
- Do not use references on your assignment
  - You will get zero points
  - You need practice with mutation-free programming
  - They will lead to less elegant solutions

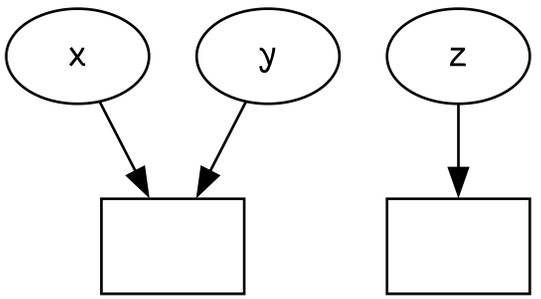
References

- New types: `t ref` where `t` is a type
  - References are first-class
- New expressions:
  - `ref e`

    to create a reference with initial contents `e`
    - Just a record with one *mutable* field `contents`
  - `e1 := e2` to update contents
  - `!e` to retrieve contents (not negation)
- Aliasing matters for references because of mutability

References example

```
let x = ref 42 (* : int ref *)
let y = ref 42
let z = x
let _ = x := 43
let w = (!y) + (!z) (* 85 *)
(* x + 1 does not type-check *)
```



- A variable bound to a reference (e.g., `x`) is still immutable: it will always refer to the *same reference*
- But the *contents* of the reference may change via `:=`
- And there may be aliases to the reference, which matter a lot
- References are first-class values

Callbacks

- A common idiom: library takes functions to apply later, when an *event* occurs; e.g.:
  - when a key is pressed, mouse moves, data arrives
  - when the program enters some state (e.g., turns in a game)
- A library may accept multiple callbacks
  - Different callbacks may need different private data with different types
  - Fortunately, a function's type does not include the types of bindings in its environment
  - In OOP, objects and private fields are used similarly, e.g., Java "listeners"

Mutable state

- While it's not absolutely necessary, mutable state is reasonably appropriate here…
- We really do want the *collection of callbacks registered* to **change** when a function to register a callback is called

Example call-back library

- Library maintains mutable state for *what callbacks are there* and provides a function for accepting new ones
  - A real library would also support removing them, and so on
  - In our example, callbacks have type `int -> unit`
    - `unit` is equivalent to C or Java's `void`
- So the entire public library interface would be the function for registering new callbacks:

```
val onKeyEvent : (int -> unit) -> unit
```

- Because callbacks are executed for side-effect, they may also need mutable state

Library implementation

```
let callbacks: (int -> unit) list ref = ref []
let on_key_event f =
  callbacks := f :: !callbacks
let do_key_event i =
  List.iter (fun f -> f i) !callbacks
```

Clients

- Can only register an `int -> unit`, so if any other data is needed, must be in closure's environment
  - And if they need to *remember* something, they can use mutable state
- Examples:

```
let times_pressed = ref 0
let _ = on_key_event (fun _ -> times_pressed := !times_pressed + 1)
let print_if_pressed i =
  on_key_event (fun j ->
    if i = j
    then print_endline ("pressed " ^ string_of_int i)
    else ())
```

Code demonstration

More idioms

- We know the rule for lexical scope and function closures
  - Now: what is it good for?
- Partial but wide-ranging list:
  - [x] Pass functions with private data to iterators
  - [x] Currying (multi-argument functions and partial application)
  - [x] Combine functions (e.g., composition)
  - [x] Callbacks (e.g., in reactive programming)
  - [ ] Implementing an ADT with a record of functions (optional)

Abstract data types

Abstract data types

- As our last idiom, closures can implement **abstract data types** (ADTs)
  - Can put multiple functions in a record
  - The functions can share the same private data
  - Private data can be mutable or immutable
  - Feels a lot like objects, emphasizing that OOP and functional programming have some deep similarities
- See code for an implementation of immutable integer sets with operations `insert`, `member` and `size`
- The actual code is advanced/clever/tricky, but has no new features
  - Combines lexical scope, variant types, records, closures, etc.
  - Client use is less tricky

Abstract data types: library

```
type set =
  S of { insert : int -> set; member : int -> bool; size : unit -> int; }

let empty_set =
  let rec make_set xs = (* xs is a "private field" *)
    let contains i = List.mem i xs in (* contains is a "private method" *)
    S { insert = (fun i -> if contains i then make_set xs else make_set (i :: xs))
      ; member = contains
      ; size = (fun () -> List.length xs) }
  in
  make_set []
```

Abstract data types: client

```
let sz =
  let S s1 = empty_set in
  let S s2 = s1.insert 34 in
  let S s3 = s2.insert 34 in
  let S s4 = s3.insert 19 in
  s4.size ()
```