Programming Languages and Techniques (CIS120)

Lecture 18
October 15, 2018

Hidden state, Objects, GUI Library Design
Chapters 17 & 18
Announcements

• Midterm is Graded
  – Solutions & statistics will be released after makeup exams are processed

• HW05: GUI & Paint program
  – Available online
  – Due: Tuesday, October 23rd at 11:59 pm
  – Work on Tasks, not files
  – START TODAY! Goal: Complete "Task 0" by tonight
Hidden State

Encoding "objects" by encapsulating state
An “incr” function

```ocaml
type counter_state = { mutable count : int }

(* allocate a counter_state *)
let ctr = { count = 0 }

(* each call to incr will produce the next integer *)
let incr : unit -> int = fun () ->
  ctr.count <- ctr.count + 1;
  ctr.count
```

• **Drawbacks:**
  
  – *No reuse*: There is only one ctr in the world. If we want another, we need to allocate another counter_state value and define another incr function to modify it.

  – *No encapsulation*: Any other code can modify count, too.
Using Hidden State

• Make a function that allocates state and an incr function for that state each time a counter is needed.

```ocaml
(* More useful: a counter generator: *)
let mk_incr () : unit -> int =
  (* this ctr is private to the returned function *)
  let ctr = { count = 0 } in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count

(* make one counter *)
let incr1 : unit -> int = mk_incr ()

(* make another counter *)
let incr2 : unit -> int = mk_incr ()
```
Running mk_incr

Workspace

```
let mk_incr () : unit -> int =
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count

let incr1 : unit -> int =
  mk_incr ()
```

Stack

Heap
Local Functions

Workspace

Stack

Heap

fun () ->
    let ctr = {count = 0} in
fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count

D oppressive

Note how the count record is accessible only via the incr1 function. This is the sense in which the state is “local” to incr1.
Objects
One step further...

• `mk_incr` shows us how to create different instance of local state so that we can have several different counters.

• What if we want to bundle together *several* operations that share the same local state?
  – e.g. incr and decr operations that work on the same counter

Key Concept: **Object**

An object consists of:
• some encapsulated mutable state (*fields*)
• a set of operations that manipulate that state (*methods*)
A Counter Object

(* The type of counter objects *)

type counter = {
  get : unit -> int;
  incr : unit -> unit;
  decr : unit -> unit;
  reset : unit -> unit;
}

(* Create a fresh counter object with hidden state: *)

let new_counter () : counter =
  let ctr = {count = 0} in
  {
    get = (fun () -> ctr.count);
    incr = (fun () -> ctr.count <- ctr.count + 1);
    decr = (fun () -> ctr.count <- ctr.count - 1);
    reset = (fun () -> ctr.count <- 0);
  }
let c1 = new_counter ()

Stack

new_counter

Heap

fun () ->
    let ctr = {count = 0} in
    { ... }

c1

c1

get

incr

decr

reset

count

ctr

fun () -> ctr.count

ctr

fun () ->
    ctr.count <- ctr.count + 1

ctr

fun () ->
    ctr.count <- ctr.count - 1

ctr

fun () ->
    ctr.count <- 0
Using Counter Objects

(* a helper function to create a nice string for printing *)

let ctr_string (s:string) (i:int) =
  s ^ ".ctr = " ^ (string_of_int i) ^ "\n"

let c1 = new_counter ()
let c2 = new_counter ()

;; print_string (ctr_string "c1" (c1.get ()))
;; c1.incr ()
;; c1.incr ()
;; print_string (ctr_string "c1" (c1.get ()))
;; c1.decr ()
;; print_string (ctr_string "c1" (c1.get ()))
;; c2.incr ()
;; print_string (ctr_string "c2" (c2.get ()))
;; c2.decr ()
;; print_string (ctr_string "c2" (c2.get ()))
Objects and GUIs
Where we’re going...

• HW 5: Build a GUI library and client application *from scratch* in OCaml

• Goals:
  – Practice with *first-class functions* and *hidden state*
  – Bridge to object-oriented programming in Java
  – Illustrate the *event-driven* programming model
  – Give you a feel for how GUI libraries (like Java’s Swing) work
  – Apply everything we’ve seen so far to do some pretty serious programming
Have you ever used a GUI library (such as Java's Swing) to construct a user interface?
Building a GUI library & application
Step #1: Understand the Problem

- There are two separate parts of this homework: an application (Paint) and a GUI library used to build the application.

- What are the concepts involved in GUI libraries and how do they relate to each other?

- How can we separate the various concerns on the project?

- Goal: The library should be reusable. It should be useful for other applications besides Paint.
GUI Library Design

Abstractions for graphical interfaces
Goal of the GUI library: provide a consistent layer of abstraction *between* the application (Paint) and the Graphics module.
OCaml’s Graphics* library provides very basic primitives for:

– Creating a window
– Drawing various shapes: points, lines, text, rectangles, circles, etc.
– Getting the mouse position, whether the mouse button is pressed, what key is pressed, etc.

How do we go from that to a full-blown GUI library?

*Note: We actually have two Graphics libraries, one for running "natively" and one for running in the browser. We have configured the project so that you can refer to either one using the module alias Graphics.

For use within the browser, we use a tool called js_of_ocaml that translates OCaml-compiled bytecode into javascript. There are some rendering differences between the native and browser versions.
Goal of the GUI library: provide a consistent layer of abstraction between the application (Paint) and the Graphics module.
GUI terminology – Widget*

• Basic element of GUIs: examples include buttons, checkboxes, windows, textboxes, canvases, scrollbars, labels

• Every widget
  – has a position on the screen
  – knows how to display itself
  – knows how to react to events like mouse clicks

• May be composed of other sub-widgets, for laying out complex interfaces

HelloWorld

*Each GUI library uses its own naming convention for what we call “widgets.” Java Swing calls them “Components”; iOS UIKit calls them “UIViews”; WINAPI, GTK+, X11’s widgets, etc....
A “Hello World” application

(* Create some simple label (string) widgets *)

let l1 : widget = label "Hello"

let l2 : widget = label "World"

(* Compose them horizontally, adding some borders *)

let h : widget =
  border (hpair (border l1)
      (hpair (space (10,10)) (border l2)))
Module: Gctx

Contextualizes graphics drawing operations
Challenge: Widget Layout

- Widgets are “things drawn on the screen”. How to make them location independent?
- Idea: Use a graphics context to make drawing relative to a widget’s current position

The graphics context isolates the widgets from the Graphics module.
GUI terminology – Graphics Context

• Wraps OCaml Graphics library; puts drawing operations “in context”
• Translates coordinates
  – *Flips* between OCaml and “standard” coordinates so origin is top-left
  – *Translates* coordinates so all widgets can pretend that they are at the origin
• Also aggregates information about
  – foreground color
  – line width
• "Task 0" in the HW helps you understand the interaction between Gctx and OCaml's Graphics module
let top = Gctx.top_level in
(* move origin and change the color *)
let nctx = Gctx.with_color Color.red
            (Gctx.translate top (dx,dy)) in
Gctx.draw_string top (0,10) "CIS 120";
Gctx.draw_string nctx (0,10) "CIS 120"
Module Gctx

(** The main (abstract) type of graphics contexts. *)

```
type gctx
```

(** The top-level graphics context *)

```
val top_level : gctx
```

(** A widget-relative position *)

```
type position = int * int
```

(** Display text at the given (relative) position *)

```
val draw_string : gctx -> position -> string -> unit
```

(** Draw a line between the two specified positions *)

```
val draw_line : gctx -> position -> position -> unit
```

(** Produce a new gctx shifted by (dx,dy) *)

```
val translate : gctx -> int * int -> gctx
```

(** Produce a new gctx with a different pen color *)

```
val with_color : gctx -> color -> gctx
```