CIS 1200 Midterm 2 November 14, 2022

Benjamin C. Pierce and Swapneel Sheth, instructors

SOLUTIONS

1. **Deques and the OCaml ASM** (9 points)

   Recall the definitions of `deque` and `dqnode` from homework 4:

```
type 'a dqnode = {
  v: 'a;
  mutable next: 'a dqnode option;
  mutable prev: 'a dqnode option;
}

type 'a deque = {
  mutable head: 'a dqnode option;
  mutable tail: 'a dqnode option;
}
```

   In Appendix A you will find a collection of ASM drawings showing various possible configurations of the heap. In Appendix B you will find the invariant for deques, with each clause annotated with a letter between (a) and (f).

   For each of the following code sequences, please (1) choose which picture from Appendix A corresponds to the ASM's stack and heap after executing this code, and (2) check either the "satisfies invariant" box (if the heap at the end satisfies the deque invariant) or else one or more of the boxes labeled (a) through (f) to indicate which clauses of the invariant are *not* satisfied.

(a)
```
let d =
  let node = { v = 1; prev = None; next = None } in
  { head = Some node; tail = Some node }
```

Matches drawing

☐ A ☒ B ☐ C ☐ D ☐ E ☐ F

☒ Satisfies invariant *or*

☐ Clause (a) fails ("tail reachable from head via next")
☐ Clause (b) fails ("nothing next after tail")
☐ Clause (c) fails ("head reachable from tail via prev")
☐ Clause (d) fails ("nothing previous before head")
☐ Clause (e) fails ("next then prev")
☐ Clause (f) fails ("prev then next")

(b)
```
let d =
  { head = Some { v = 1; prev = None; next = None };
    tail = Some { v = 1; prev = None; next = None }; }
```

Matches drawing

☒ A ☐ B ☐ C ☐ D ☐ E ☐ F

☐ Satisfies invariant *or*

☒ Clause (a) fails ("tail reachable from head via next")
☐ Clause (b) fails ("nothing next after tail")
☒ Clause (c) fails ("head reachable from tail via prev")
☐ Clause (d) fails ("nothing previous before head")
☐ Clause (e) fails ("next then prev")
☐ Clause (f) fails ("prev then next")

(c)
```
let d =
  let node1 = { v = 1; prev = None; next = None } in
  let node2 = { v = 1; prev = Some node1; next = None } in
  { head = Some node2; tail = Some node1 }
```

Matches drawing

☐ A ☐ B ☐ C ☒ D ☐ E ☐ F

☐ Satisfies invariant *or*

☒ Clause (a) fails ("tail reachable from head via next")
☐ Clause (b) fails ("nothing next after tail")
☒ Clause (c) fails ("head reachable from tail via prev")
☒ Clause (d) fails ("nothing previous before head")
☐ Clause (e) fails ("next then prev")
☒ Clause (f) fails ("prev then next")

2. **Programming with Deques** (12 points)

   Now suppose we want to define a function `swapWithNext` that, given a `deque d` and a pointer `n` to a `dqnode` somewhere inside it, swaps `n` with the node immediately following it in the queue.

   Complete the code below for `swapWithNext`. Make sure that it correctly handles the case when `n` is the first or last node in `d`.

```
let swapWithNext (d : 'a deque) (n1 : 'a dqnode) : unit =
  begin match n1.next with
  | None -> ()
  | Some n2 ->
     begin match n1.prev with
     | None -> d.head <- Some n2
     | Some n0 -> n0.next <- Some n2
     end;
     begin match n2.next with
     | None -> d.tail <- Some n1
     | Some n3 -> n3.prev <- Some n1
     end;
     n2.prev <- n1.prev;
     n1.prev <- Some n2;
     n1.next <- n2.next;
     n2.next <- Some n1
end
```

3. **OCaml Objects and GUI Concepts** (5 points)

(a) A *closure* is simply the text of a function that has been copied into the heap.

True ☐ False ☒

Closures in the heap also include saved copies of the portion of the stack that must be restored when they are called.

(b) The only way to change the encapsulated state of any of the widgets defined in the `widget` module is by calling the `handle` method of that widget.

True ☐ False ☒

Some widgets come with controllers *that can also access and modify their encapsulated state.*

(c) According to the design principles of our GUI library, calling the `repaint` or `size` method of a widget should not change its state—only `handle` should do that.

True ☒ False ☐

(d) When writing a `repaint` method in the style of our GUI library, every call to a low-level drawing primitive from the `Graphics` module should use the provided `Gctx.gctx` to transform from the widget's local coordinates to screen coordinates.

True ☒ False ☐

(e) In our GUI library, a `notifier` is a first-class function stored in the hidden state of an `event_listener` widget. When an event occurs on the widget, the `event_listener` invokes all of the stored `notifier`s.

True ☐ False ☒

Other way 'round.

4. **GUI Programming** (12 points)

   Consider the GUI library from HW05, part of which is also shown in Appendix C and D.

   Several widgets (`label`, `border`, etc.) draw on the screen in whatever pen color is found in the `Gctx.gctx` they are passed. This means that an outer widget can use `Gctx.with_color` to change how they look.

   Suppose we want to extend the widget library in the `Widget` module with a version of `with_color` that works on *widgets* instead of on graphics contexts—that is, it has type

   ```
   Widget.widget -> Gctx.color -> Widget.widget
   ```

   instead of:

   ```
   Gctx.gctx -> Gctx.color -> Gctx.gctx
   ```

   Internally, it will call `Gctx.with_color` as needed; additionally, it will take care of passing `repaint`, `handle`, and `size` calls down to its inner widget.

   For example, writing this

   ```
   let wgray : widget =  with_color (border (label "Gray")) Gctx.gray
   let wblack : widget = border (label "Black")
   let top : widget = hpair wgray wblack
   ;; Eventloop.run top
   ```

   should display this:

   Complete the implementation of `Widget.with_color` on the next page.

```
let with_color (w: widget) (c: Gctx.color) : widget =
  {
    repaint = (fun (g:Gctx.gctx) -> w.repaint (Gctx.with_color g c));
    handle = (fun (g:Gctx.gctx) (e: Gctx.event) -> w.handle g e);
    size = (fun () -> w.size ())
  }
```

5. **Java Objects and Equality** (6 points)

Consider the following Java interface and class definitions:

```
interface Incrementable {
  int incr ();
}

class Counter implements Incrementable {
  private int x = 0;
  public int incr () { x = x+1; return x; }
}

class Box implements Incrementable {
  public Incrementable i;
  public Box (Incrementable init) { i = init; }
  public int incr () { return i.incr(); }
  public Incrementable contents () { return i; }
}
```

Fill in the blanks in the following JUnit test cases so that all the tests pass.

```
@Test
public void test1 () {
  Counter c1 = new Counter();
  Counter c2 = new Counter();
  Box b1 = new Box(c1);

  assertEquals(b1.incr(), 1);

  assertEquals(c2.incr(), 1);

  assertEquals(c1.incr(), 2);

}

@Test
public void testB() {
  Counter c = new Counter();
  Box x1 = new Box(c);
  Box x2 = new Box(x1);

  assertEquals(x1 == x2, false);
  assertEquals(x1.contents() == x2.contents(), false);
  assertEquals(x1 == x2.contents(), true);
}
```

6. **Java Array Programming** (16 points)

Write a function `find` that takes two **`int`**`[]` arrays, `original` and `pattern`, as parameters and returns **`true`** if the `original` array contains the elements of `pattern` *in a single contiguous block* and otherwise returns **`false`**.

E.g., the following calls to `find` should return **`true`**...

```
original          pattern
{1, 2, 3, 3, 4}   {1, 2, 3}
{1, 2, 3, 3, 4}   {2, 3}
{1, 2, 3, 3, 4}   {3, 4}
{1, 2, 3, 3, 4}   {3, 3, 4}
{1, 2, 3, 3, 4}   {}
```

... whereas these calls should return **`false`**:

```
original          pattern
{1, 2, 3, 3, 4}   {7}
{1, 2, 3, 3, 4}   {2, 2}
{1, 2, 3, 3, 4}   {2, 4}
{1, 2, 3, 3, 4}   {1, 2, 3, 3, 4, 5}
```

You may assume that neither input array is **`null`**. Do not use any helper functions.

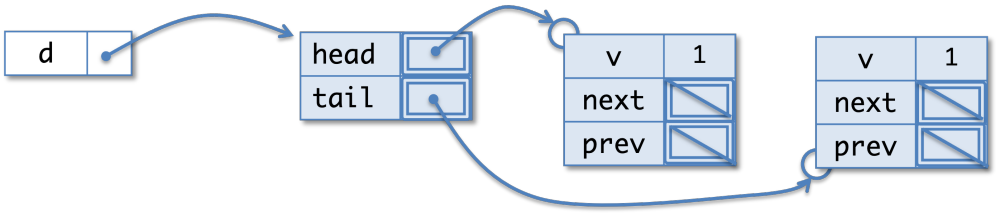
```
public static boolean find (int[] original, int[] pattern) {
    for (int n = 0; n <= original.length - pattern.length; n++) {
        boolean wrong = false;
        for (int p = 0; p < pattern.length; p++) {
            if (original[n+p] != pattern[p]) {
                wrong = true;
            }
        }
        if (!wrong) return true;
    }
    return false;
}
```

Scratch Space

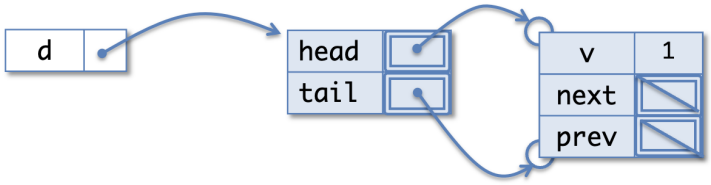
*Use this page for work that you do not want us to grade. If you run out of space elsewhere in the exam and you **do** want to put something here that we should grade, make sure to put a clear note in the normal answer space for the problem in question.*

A ASM Drawings

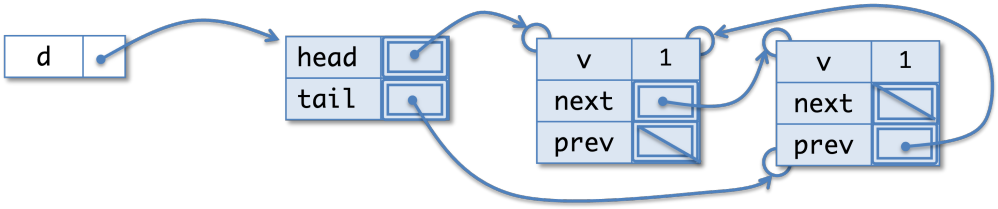
A:



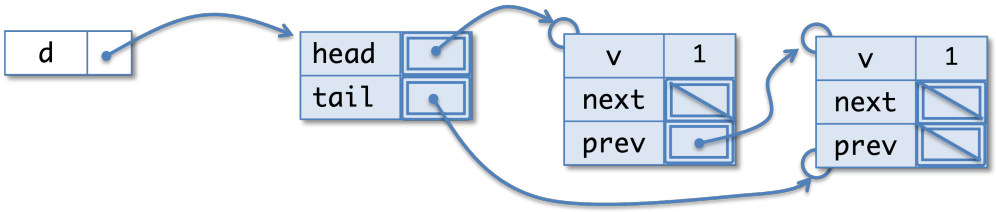
B:



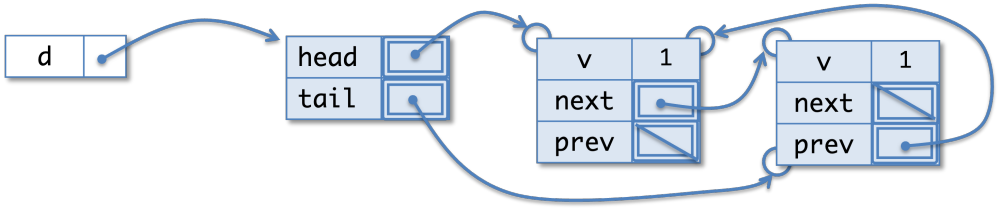
C:



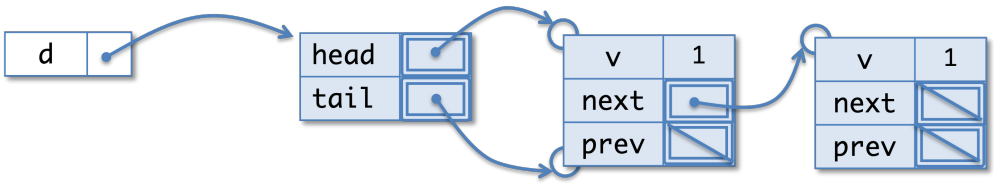
D:



E:



F:



B Deque Invariant

Either (1) the deque is empty and the head and tail are both `None`, or (2) the deque is non-empty and

- `head = Some n1` and `tail = Some n2`, where
  (a) `n2` is reachable from `n1` by following `next` pointers ("tail reachable from head via next")
  (b) `n2.next = None` ("nothing next after tail")
  (c) `n1` is reachable from `n2` by following `prev` pointers ("head reachable from tail via prev")
  (d) `n1.prev = None` ("nothing previous before head"); and
- for every node `n` in the deque,
  (e) if `n.next = Some m` then `m.prev = Some n` ("next then prev")
  (f) if `n.prev = Some m` then `m.next = Some n` ("prev then next").

C GUI library: `Widget.mli` excerpt

```
type widget = {
  repaint : Gctx.gctx -> unit;
  handle  : Gctx.gctx -> Gctx.event -> unit;
  size    : unit -> Gctx.dimension;
}

val hpair : widget -> widget -> widget

type label_controller = {
  get_label : unit -> string;
  set_label : string -> unit;
}
val label : string -> widget * label_controller

(* New: *)
val with_color : widget -> Gctx.color -> widget
```

D GUI library: `Gctx.mli` excerpt

```
(****************)
(** {1 Colors } *)
(****************)

type color = {r:int; g:int; b:int}

(* New: *)
val gray    : color

(*****************************)
(** {1 Basic Gctx operations } *)
(*****************************)

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