Image Manipulation ⋆

Topics

- Algebraic data types
- Pattern matching
- Higher order functions

Exercise 1 – Abstract image description

In this exercise, we will define color pictures as OCaml types with the following definitions:

```
type image =
  | Background of color
  | Crop of image * path (** resulting image is transparent outside of path *)
  | Compose of image * compose_operator * image
and path =
  | Polygon of coordinate list   (** vertex list *)
  | Circle of coordinate * float (** center, radius *)
and compose_operator = Over | In | Out | Atop | Xor
and coordinate =
  { x: float;
    y: float; }
and color =
  { ra: float;
    ga: float;
    ba: float;
    a: float; }
```

An image is either: a uniformaly colored plane (`Background color`), the restriction of another image to the interior of a closed path (`Crop (img, path)`), or the composition of two images by using one of SVG's compositing operators[1] (`Compose (src, op, dest)`). A path is either: a polygon (defined by the list of its vertices), or a circle (defined by its center and its radius).

Question 1.1 – Write the functions `square side color` and `disk radius color` that construct the corresponding image.

Question 1.2 – Write a function `compose_right o l` taking a compositing operator `o` and a list of images `l` and returning the composition with `o` of all the images of `l`, starting with the last two images. Also write the symetrical function `compose_left`.

[1] http://www.w3.org/TR/2009/WD-SVGCompositing-20090430/#containerElementCompositingOperators

Exercise 2 – Displaying images

Question 2.1 – Write a function that tests if a given point is inside a closed path.

Question 2.2 – Write a function evaluate of type `image -> coordinate -> color`, that computes the color of an image at a given coordinate.

Question 2.3 – Amend the `render` function, from the previous exercises session, to the new types `coordinate` and `color`, and display some simple image.

Exercise 3 – Image manipulation

Question 3.1 – Write a function `move v img`, of type `coordinate -> image -> image` that translates the image by a vector of coordinates $(0, 0) \rightarrow (\mathrm{v.x}, \mathrm{v.y})$.

Question 3.2 – Write a function `rotate a img`, of type `float -> image -> image` that rotates the image of an angle a around $(0, 0)$.

Question 3.3 – Write a function `scale z img`, of type `float -> image -> image` that zooms the image of a factor z around $(0, 0)$.

Exercise 4 – Compatibilty layer

We wish to reuse the images we defined in the previous tutorial (at least for images that do not depends on polar transformations). For that purpose, we want to define functions similar to the image combinators of the previous tutorial session, but that eventually build a value of type `image` instead of a functional image.

```
type fun_image = coordinate -> color

val plane : color -> fun_image
val disk : color -> float -> fun_image
val square : color -> float -> fun_image

val at : float -> float -> fun_image -> fun_image
val repeat : float -> float -> fun_image -> fun_image

val rotate : float -> fun_image -> fun_image
val scale : float -> fun_image -> fun_image

val compose_src_over : fun_image -> fun_image -> fun_image
val compose_src_in : fun_image -> fun_image -> fun_image
```

Image Manipulation

SOLUTIONS – SOLUTIONS – SOLUTIONS – SOLUTIONS

Solution to question 1.1

```
let square side color =
  let r = side /. 2. in
  Crop (Background color,
        Polygon [{ x = -.r;  y = -.r };
                 { x = -.r;  y =   r };
                 { x =   r;  y =   r };
                 { x =   r;  y = -.r }])
let disk radius color =
  Crop (Background color, Circle ({ x = 0.; y = 0.}, radius))
```

Solution to question 1.2

```
let rec combine_right binop fs =
  match fs with
  | [] -> Background Color.transparent
  | [f] -> f
  | f :: fs -> Compose (f, binop, combine_right binop fs)
let combine_left binop fs = combine_right binop (List.rev fs)
```

Solution to question 2.1

```
let intersect { x; y; } (a, b) =
  ((b.y > y && a.y < y) || (b.y < y && a.y > y))
  && (x <= b.x +. (b.y -. y) *. (a.x -. b.x) /. (b.y -. a.y))
let segments = function
  | [] -> invalid_arg "segments␣(empty␣list)"
  | [_] -> invalid_arg "segments␣(singleton)"
  | x :: _ as xs ->
    let rec loop = function
      | [] -> assert false
      | [y] -> [(y, x)]
      | y1 :: (y2 :: _ as xs) -> (y1, y2) :: loop xs in
    loop xs
let rec count_intersect p = function
  | [] -> 0
  | x :: xs ->
      (if intersect p x then 1 else 0) + count_intersect p xs
let is_in_path p = function
  | Polygon points ->
      count_intersect p (segments points) mod 2 = 1
  | Circle (p', r) ->
      let dx = p.x -. p'.x
      and dy = p.y -. p'.y in
      dx *. dx +. dy *. dy <= r *. r
```

Solution to question 2.2

```
let eval_op = function
  | Over -> Funpics.compose_src_over
  | In -> Funpics.compose_src_in
  (* ... *)
let rec evaluate = function
  | Background c -> Funpics.plane c
  | Crop (src, path) ->
      let src = evaluate src in
      fun { x; y; } ->
        if is_in_path { x; y; } path then
          src { x; y; }
        else
          Color.transparent
  | Compose (src, op, dest) ->
      eval_op op (evaluate src) (evaluate dest)
```

Solution to question 3.1

```
let move_point (dx, dy) { x; y; } = { x = x +. dx; y = y +. dy }
let move_path v = function
  | Polygon points -> Polygon (List.map (move_point v) points)
  | Circle (p, r) -> Circle (move_point v p, r)
let rec move v = function
  | Background _ as src -> src
  | Crop (src, path) ->
    Crop (move v src, move_path v path)
  | Compose (src, op, dest) ->
    Compose (move v src, op, move v dest)
```

Solution to question 3.2

```
let rotate_point da { x; y; } =
  let a = atan2 y x +. da in
  let p = sqrt (x *. x +. y *. y) in
  { x = p *. cos a; y = p *. sin a }
let rotate_path da = function
  | Polygon points -> Polygon (List.map (rotate_point da) points)
  | Circle (p, r) -> Circle (rotate_point da p, r)
let rec map f = function
  | Background _ as src -> src
  | Crop (src, path) -> Crop (map f src, f path)
  | ACompose (src, op, dest) -> ACompose (map f src, op, map f dest)
let rotate da = map (rotate_path da)
```

Solution to question 3.3

```
let scale_point s { x; y; } = { x = x *. s; y = y *. s }
let scale_path s = function
  | Polygon points -> Polygon (List.map (scale_point s) points)
  | Circle (p, r) -> Circle (scale_point s p, r *. s)
let scale s = map (scale_path s)
```

Solution to exercise 4

```
type rel_image =

  (* Basic shapes *)
  | Plane of Color.t
  | Disk of Color.t * float (* radius *)
  | Square of Color.t * float (* half-width *)

  (* Positioning *)
  | At of coordinate * image
  | Repeat of float (* w *) * float (* h *) * image

  (* Composition *)
  | Compose of image (* src *) * compose_operator * image (* dest *)

  (* Basic transformation *)
  | Scale of float * image
  | Rotate of float * image
  | PolarFun of (float * float -> float * float) * image

  let plane c = Plane c
  let disk c r = Disk (c, r)
  let square c r = Square (c, r)

  let at x y src = At ({ x; y }, src)
  let scale z src = Scale (z, src)
  let rotate da src = Rotate (da, src)
  let repeat w h src = Repeat (w, h, src)
  let waves wl src =
    let f (p, a) = (p -. sin (p /. wl) *. 0.5 *. wl, a) in
    PolarFun (f, src)
  let warp ph n amp src =
    let f (p, a) = (p +. sin (ph +. a *. float n) *. amp, a) in
    PolarFun (f, src)

  let compose_src_over src dest = Compose (src, Over, dest)
  let compose_src_in src dest = Compose (src, In, dest)

let rec (--) i j =
  if i > j then [] else i :: succ i -- j

let rec product xs ys =
  match xs with
  | [] -> []
  | x :: xs ->
      List.map (fun y -> (x, y)) ys @ product xs ys

let rec image_of_rel_image (b_l, t_r) : image -> absolute_image =
  function
  | Plane c -> Background c
  | Disk (c, r) ->
      Crop (Background c, Circle ({ x = 0.; y = 0.}, r))
  | Square (c, r) ->
    Crop (Background c, Polygon [{ x = -.r;  y = -.r };
```

```
                                { x =   r;  y = -.r };
                                { x =   r;  y =   r };
                                { x = -.r;  y =   r }])
  | At (p, src) ->
      let v = (p.x, p.y) in
      let mv = (-.p.x, -.p.y) in
      move v
        (image_of_rel_image (move_point mv b_l, move_point mv t_r) src)
  | Compose (src, op, dest) ->
    ACompose (image_of_rel_image (b_l, t_r) src, op,
              image_of_rel_image (b_l, t_r) dest)

  | Rotate (da, src) ->
    let b_l = rotate_point (-.da) b_l
    and b_r = rotate_point (-.da) { x = t_r.x; y = b_l.y }
    and t_r = rotate_point (-.da) t_r
    and t_l = rotate_point (-.da) { x = b_l.x; y = t_r.y } in
    let b_l = { x = min (min b_l.x b_r.x) (min t_l.x t_r.x);
                y = min (min b_l.y b_r.y) (min t_l.y t_r.y); }
    and t_r = { x = max (max b_l.x b_r.x) (max t_l.x t_r.x);
                y = max (max b_l.y b_r.y) (max t_l.y t_r.y); } in
    rotate da (image_of_rel_image (b_l, t_r) src)
  | Scale (z, src) ->
    let b_l = scale_point (1./.z) b_l in
    let t_r = scale_point (1./.z) t_r in
    scale z (image_of_rel_image (b_l, t_r) src)

  | Repeat (inner_w, inner_h, src) ->
    let half_inner_w = inner_w /. 2. in
    let half_inner_h = inner_h /. 2. in
    let inner_b_l = { x = -. half_inner_w; y = -. half_inner_h } in
    let inner_b_r = { x =    half_inner_w; y = -. half_inner_h } in
    let inner_t_r = { x =    half_inner_w; y =    half_inner_h } in
    let inner_t_l = { x = -. half_inner_w; y =    half_inner_h } in
    let src =
      Crop (image_of_rel_image (inner_b_l, inner_t_r) src,
            Polygon [inner_b_l; inner_b_r; inner_t_r; inner_t_l]) in
    let min_w = int_of_float @@ floor @@ (b_l.x +. inner_w /. 2.)  /. inner_w in
    let min_h = int_of_float @@ floor @@ (b_l.y +. inner_h /. 2.) /. inner_h in
    let max_w = int_of_float @@ ceil @@ (t_r.x -. inner_w /. 2.) /. inner_w in
    let max_h = int_of_float @@ ceil @@ (t_r.y -. inner_h /. 2.) /. inner_h in
    let positions = product (min_w -- max_w) (min_h -- max_h) in
    let images =
      List.map
        (fun (i, j) -> move (inner_w *. float i, inner_h *. float j) src)
        positions in
    List.fold_right
      (fun src dest -> Compose (src, Over, dest))
      images (Background Color.transparent)
  | PolarFun _ -> invalid_arg "not␣implemented␣(Polar)"
```