PoP g6

Carl Dybdahl, Patrick Hartvigsen, Emil Søderblom

October 18, 2016

1 Task g6.1

We have written a function `numberToDay : int -> weekday option` which converts the numbers `1` through `7` to the weekdays `Some Monday` through `Some Sunday`, returning `None` if the integer is not in the interval.

```
type weekday = Monday | Tuesday | Wednesday | Thursday | Friday
             | Saturday | Sunday

/// <summary>
///  Converts a number <code>1 .. 7</code> to a day
///  <code>Monday .. Friday</code>
/// </summary>
/// <remarks>
///  This function returns a value of type <code>weekday option</code>. This is
///  so it can return <code>None</code> if the input is not in the allowed
///  interval.
/// </remarks>
/// <example>
///   The following code:
///   <code>
///     printfn "%A" (numberToDay 1)
///   </code>
///   prints "Some Monday" to the console.
/// </example>
/// <param name="n">The index of the weekday.</param>
/// <returns>The weekday.</returns>
let numberToDay n =
   match n with
    | 1 -> Some Monday
    | 2 -> Some Tuesday
    | 3 -> Some Wednesday
    | 4 -> Some Thursday
    | 5 -> Some Friday
    | 6 -> Some Saturday
    | 7 -> Some Sunday
    | _ -> None
```

We have chosen the most direct approach possible, where we just enumerate input-output pairs in a `match` construct in the function.

2 Task g6.2

We were tasked with making a figure `g61` consisting of two copies of a different figure `o61`, which consists of a red rectangle and a blue circle.

```
type point = int * int
type colour = int * int * int
type figure =
    | Circle of point * int * colour
    | Rectangle of point * point * colour
    | Mix of figure * figure
    | Twice of figure * (int * int)

let o61 =
   let circ = Circle ((50, 50), 45, (255, 0, 0))
   let rect = Rectangle ((40, 40), (90, 110), (0, 0, 255))
   Mix (circ, rect)

let g61 = Twice (o61, (50, 70))
```

3 Task g6.3

We have extended the function `colourAt : point -> figure -> colour` to deal with the extension of `figure` with the constructor `Twice`.

```
/// <summary>Finds the colour at a position in a figure.</summary>
/// <remarks>
///  May take exponential time if the figure has many recursively nested
///  <code>Twice</code> constructors. If the point is outside the area of the
///  figure, the function returns <code>None</code>.
/// </remarks>
/// <example>
///   The following code:
///   <code>
///     printfn "%A" (colourAt (0, 0) (Circle ((0, 0), 10, (255, 0, 0))))
///   </code>
///   prints "Some (255, 0, 0)" to the console.
/// </example>
/// <param name="(x, y)">The coordinates to find the colour at.</param>
/// <param name="figure">The figure to find the colour of.</param>
/// <returns>The colour at the point on the figure.</returns>
let rec colourAt (x, y) figure =
   match figure with
   | Circle ((cx, cy), r, col) ->
      if (x-cx)*(x-cx)+(y-cy)*(y-cy) <= r*r
      then Some col else None
   | Rectangle ((x0,y0), (x1,y1), col) ->
      if x0<=x && x <= x1 && y0 <= y && y <= y1
      then Some col else None
   | Mix (f1, f2) ->
      match (colourAt (x,y) f1, colourAt (x,y) f2) with
      | (None, c) -> c // overlapper ikke
      | (c, None) -> c // ditto
      | (Some (r1,g1,b1), Some (r2,g2,b2)) ->
         Some ((r1+r2)/2, (g1+g2)/2, (b1+b2)/2)
   | Twice (f1, (dx, dy)) ->
      match (colourAt (x, y) f1, colourAt (x-dx,y-dy) f1) with
      | (c, None) -> c
      | (_, Some c2) -> Some c2
```

`Twice (f, (dx, dy))` has the denotation of consisting of two copies of the figure `f`, with the copy on top being translated by `(dx, dy)`. To find the colour at `(x, y)` of `Twice (f, (dx, dy))`, we consider the colour of the lower and the upper copy at this position. The lower copy has the same colours as `f`, so this can be examined simply with `colourAt (x, y) f`. Because the upper copy is translated, we must translate the points we query by `(-dx, -dy)` to find the colours of them.

The existence of the constructor `Twice` means that we use exponential time in the worst case to find the colour at a position in a figure, since one can nest them arbitrarily deeply and each layer of nestings doubles the amount of calls to `colourAt`.

4 Task g6.4

In order to draw `figure`s, we need to add a background color. We use gray for this.

```
let gray = (128, 128, 128)
let convert (c : colour option) : int * int * int =
   match c with
   | None -> gray
   | Some (x) -> x

makeBMP.makeBMP "g63" 150 200 (fun (x, y) -> convert <| colourAt (x, y) g61)
```

5 Task g6.5

In order to extend `checkFigure` with a case for `Twice`, we note that the position of figures does not matter for whether a figure is correct. This means that we can check the translated version that `Twice` creates simply by checking the untranslated version.

```
/// <summary>Checks that a colour is valid.</summary>
/// <param name="c">The colour to check in RGB format.</param>
/// <returns>Whether the colour is valid.</returns>
let checkColor (c: colour) : bool =
   match c with
   | (r, _, _) when r < 0 || r > 255 -> false
   | (_, g, _) when g < 0 || g > 255 -> false
   | (_, _, b) when b < 0 || b > 255 -> false
   | _ -> true

/// <summary>Check that a figure is valid.</summary>
/// <example>
///   The following code:
///   <code>
///     printfn "%A" (checkFigure (Circle ((0, 0), 10, (255, 0, 0))))
///   </code>
///   prints "true" to the console.
/// </example>
/// <param name="figure">The figure to check.</param>
/// <returns>Whether the figure is valid.</returns>
let rec checkFigure (fig : figure) : bool =
   match fig with
   | Circle (_, r, c) -> checkColor c && r >= 0
   | Rectangle ((x0, y0), (x1, y1), c) -> x1 >= x0 && y1 >= y0 && checkColor c
   | Mix (f1, f2) -> checkFigure f1 && checkFigure f2
   | Twice (f, _) -> checkFigure f


/// <summary>Computes the smallest rectangle that encloses the figure.</summary>
/// <example>
///   The following code:
///   <code>
///     printfn "%A" (boundingBox (Circle ((0, 0), 10, (255, 0, 0))))
///   </code>
///   prints "((-10, -10), (10, 10))" to the console.
/// </example>
/// <param name="figure">The figure to compute the bounding box of.</param>
/// <returns>The bounding box.</returns>
let rec boundingBox (fig : figure) : point * point =
   match fig with
   | Circle ((cx, cy), r, _) -> ((cx - r, cy - r), (cx + r, cy + r))
   | Rectangle (p0, p1, _) -> (p0, p1)
   | Mix (f1, f2) ->
      let ((b1x0, b1y0), (b1x1, b1y1)) = boundingBox f1
      let ((b2x0, b2y0), (b2x1, b2y1)) = boundingBox f2
      let minx = min b1x0 b2x0
      let miny = min b1y0 b2y0
      let maxx = max b1x1 b2x1
```

```
        let maxy = max b1y1 b2y1
        ((minx, miny), (maxx, maxy))
    | Twice (f, (dx, dy)) ->
        let ((x0, y0), (x1, y1)) = boundingBox f
        let minx = min x0 (x0 + dx)
        let miny = min y0 (y0 + dy)
        let maxx = max x1 (x1 + dx)
        let maxy = max y1 (y1 + dy)
        ((minx, miny), (maxx, maxy))
```

To compute the `boundingBox` of `Twice`, we use a method similar to `Mix`, but we don't need to recurse twice, as we can simply translate the bounding box manually. This would not work if the edges of figures could lie "between" the grid lines, as they can in Nut of the Week.