Project 1 - Functional Pearls - Drawing Trees

June 10, 2022

Abby Audet
s212544

Johan Raunkjær Ott
s032060

Jinsong Li
s202354

Martin Mårtensson
s195469

Design of aesthetic pleasant renderings of trees

Hello

Property-Based Testing: Validation of rendering properties

In this section we describe how we test our implementation of designing aesthetically pleasant renderings of trees. We will describe the use of property based testing (PBT) for validating the four aesthetic rules described in [Functional Pearls: Drawing Trees]. Specifically, we use FsCheck.NUnit for integrating the FsCheck PBT tool into a unit testing framework. In separate subsections, we describe the four different aesthetic rules of the paper and specify how these rules can be described as boolean properties to be tested by FsCheck. Lastly, we analyze the notion of correctness as described in the paper and show how the correctness properties are tested, but first, we briefly describe how property based testing works with the simple case of the 'mean' function.

Simple case - the mean function

PBT concerns describing a property of a feature that should hold for all input and then test this for random input in order to ensure that the property holds for the implementation of the feature. The process is thus 1) Write a boolean function that describes the property 2) Use the FsCheck tool to create random input for the property 3) Run the test that includes the boolean function using the input generated by the FsCheck Tool. When using NUnit for testing purposes, we use the FsCheck.NUnit package that includes the attribute that should be added to the property based tests.

Let us consider the simple example of the mean function implemented as

```
let mean (x: float, y: float) : float =
    (x+y)/2.0
```

There are multiple properties that could be tested for this such as bounding properties (e.g. 'mean (x, y) =< max (x, y)' and 'mean (x, y) >= min (x, y)') and the symmetry property ('mean (x, y) = mean (y, x)'). For simplicity, we only implemented the symmetry property as

```
open FsCheck
let nf = NormalFloat.op_Explicit
let meanSymmetryProp (a,b) =
    mean (nf a, nf b) = mean (nf b, nf a)
```

such that the unit test is

```
open FsCheck.NUnit
[<Property>]
let symmetryOfMeanTest () =
    meanSymmetryProp
```

Notice that in the symmetry property, the floats are cast to the FsCheck type 'NormalFloat' that removes non-normal floats (e.g. 'nan' and 'infinity') from the randomly generated input since e.g. 'nan=nan' would return 'false'.

With this we have shown a simple example on how to use PBT and some of the pitfalls of using FsCheck. Next we use PBT to validate the implementation of the aesthetic rules that the tree design should obey.

Rule 1

'Two nodes at the same level should be placed at least a given distance apart.'

Rule 2

'A parent should be centred over its offspring.'

Rule 3

'Tree drawings should be symmetrical with respect to reflection—a tree and its mirror image should produce drawings that are reflections of each other. In particular, this means that symmetric trees will be rendered symmetrically. So, for example, Figure 1 shows two renderings, the first bad, the second good.'

Rule 4

'Identical subtrees should be rendered identically—their position in the larger tree should not affect their appearance. In Figure 2 the tree on the left fails the test, and the one on the right passes.'

Correctness

Visualization of trees

To visualize the tree we will need to map the tree structure into an image.

In the case of a tree of single letters we would need the following objects for each node:

- one letter
- one line from letter to its parent (except the root node which does not have a parent)

This task can easily be done using the SVG (Scalable Vector Graphics) format

SVG

SVG files are just text files following the XML (Extensible Markup Language) format.

An SVG representation example of just the root node would look like this:

```
<svg height="300" width="600">
<text x="300" y="0" fill="black">"A"</text>
</svg>
```

And if we add a child to the root node we will also need a line between the nodes

```
<svg height="300" width="600">
<text x="300" y="0" fill="black">"A"</text>
<text x="0" y="150" fill="black">"B"</text>
<line x1="300" y1="0" x2="0" y2="150" style="stroke:rgb(0,0,0);stroke-width:2"/>
</svg>
```

The SVG format depends on absolute coordinates in relation to the canvas of the image output where the x and y axis starts in the top left corner. So assuming we already have a correctly positioned tree using the code from the article, we can convert each node in the tree to a node which has a `x` and a `y` coordinate which will be used when plotting the node on the SVG.

Getting the absolute coordinates

Each node in the positioned tree has a position relative to its parent. To determine the absolute position of each node we need to first determine the absolute position of the root node. We already know the y coordinate of the root node because it is the one on the top of the canvas, so it is 0. To find the x coordinate of the root node we will need to find the outermost node in one of the sides of the tree and then accumulate the horizontal space all the way back to the root node. We can use the extends given by the `blueprint` function to find the coordinates of the horizontal poles of the Tree

```
let extremes (e: Extend): float*float =
    let (lefts, rights) = List.unzip ( e )
    -List.min(lefts), List.max(rights)
```

Then we can use the right extreme to compare with the right most element in each node while traversing to the down in the right side of the tree, when recursion is done each position will be returned and the root nodes absolute position in relation to the right side is given.

TODO: This function should be refactored

```
let firstPos (rightExtreme: float) (t : PosTree<'a>) : float =
    let rec f (PosNode(_, pos, cs)) =
        match (pos, cs) with
        | _, [] -> pos
        | pos, _ when pos < rightExtreme -> pos + (f (List.last cs))
        | _, _ -> pos
    rightExtreme - f t
```

The x coordinate is still not fully absolute in relation to the canvas, because every coordinate on the left of the root node has a negative value. To get the absolute value we just need to shift the element to the right by adding it with the inverted value of the left extreme.

The implementation resulted in a function which takes a simple tree, and a scale which is used to modify the distance between the coordinates of the nodes.

All number values used in the trees and extends are floats but in SVG we will need integers for the coordinates. The float values always follow the interval of 0.5 so we can get the same precision using integers by multiplying the values by 2 before we cast them to integers.

The inner function will recursively traverse through the tree and apply the absolute positions for the x and y coordinates to each node. At last the absolute positioned tree will be returned in a tuple together with the width and the height of the whole frame.

```
let absolutify (scale: int) (t: Tree<'a>) =
    let (tree, extends) = blueprint t
    let (left, right)   = extremes extends
    let width           = int((left + right) * 2.0)
    let start           = int(firstPos right tree)
    let rec f (depth: int) (px: float) (PosNode(x, pos, cs)) =
        let (t, d) =
            match cs with
            | []  -> [], depth
            | _   -> List.map (f (depth+1) (pos+px)) cs |> List.unzip |> fun (t, d) -> t, List.max d
        AbsPosNode( x, (int((pos+px+left)*2.0)*scale, depth*2*scale),  t ), d
    let (out, depth) = f 0 start tree
    out, (width * scale, depth * 2 * scale )
```

Mapping absolute coordinate tree to SVG image

When the absolute positions of each node is already given the mapping to SVG is simple. We define the SVG frame using the width and the height and the content of the SVG file is given by mapping the coordinates and the value of the nodes to the text and the line SVG objects.

```
let draw (scale: int) (t: Tree<'a>) =
    let tree, (width, height) = absolutify scale t
    let svg (content) = sprintf "<svg height=\"%i\" width=\"%i\">\n%s\n</svg>" height width content

    let text px py x y v =
        let text = sprintf "<text x=\"%i\" y=\"%i\" fill=\"black\">%A</text>\n" x y v
        let line = sprintf "<line x1=\"%i\" y1=\"%i\" x2=\"%i\" y2=\"%i\" \
                        style=\"stroke:rgb(0,0,0);stroke-width:2\"/>" px py x y
        if (px+py) = 0 then text else text+line

    let rec content (px: int, py: int) (AbsPosNode(v, (x, y), cs)) =
        let out = text px py x y v
        match cs with
        | [] -> out
        | _ -> out + "\n" + (List.map (content (x,y)) cs |> (String.concat "\n"))
    svg (content (0,0) tree)
```

Declaration

In alphabetic order, the following people contributed to Project 1: Abby Audet(AA), Jinsong Li(JL), Johan Raunkjær Ott(JRA), and Martin Mårtensson(MM).

Programming:

JRA and MM made the initial sketch of the tree design code. AA, JL, JRA, and MM contributed to debugging the tree design code. JL and MM worked on parallel implementations of visualization code. JRA made the initial sketch of the property based testing code including a simple test example. AA, JL, JRA, and MM contributed to different parts of the property based testing of the aesthetic rules.

Report:

AA made the initial sketch of section 1. JRA made the initial sketch of section 2. MM made the initial sketch of section 3. AA, JL, JRA, and MM all contributed to the itterative process of writing and correcting the report.

Analysis and discussion:

AA, JL, JRA, and MM all contributed in general analysis and discussions of the problems at hand.