Project 1 - Functional Pearls - Drawing Trees

June 12, 2022

Abby Audet
s212544

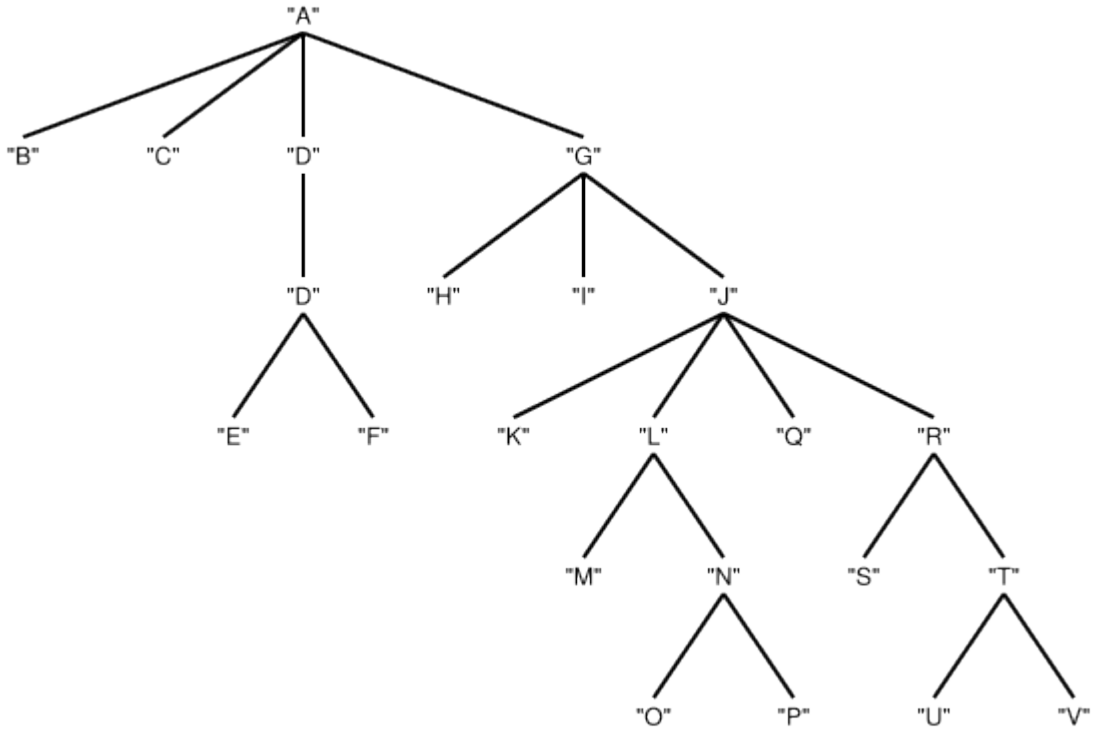
Johan Raunkjær Ott
s032060

Jinsong Li
s202354

Martin Mårtensson
s195469

Abstract

In this report, we implement a method in `F#` that is able to automatically structure trees such that they obey certain aesthical rules. It is validated that the rules are obeyed by using property based testing where FsCheck is used to generate randomly generated input using our own implementation of a random tree generator. Finally, we present a methodology for visualizing the trees by converting to SVG format.


"A"
"B"
"C"
"D"
"G"
"D"
"H"
"I"
"J"
"E"
"F"
"K"
"L"
"Q"
"R"
"M"
"N"
"S"
"T"
"O"
"P"
"U"
"V"

1 Design of Aesthetically Pleasant Renderings

The problem of designing aesthetically pleasant renderings of labelled trees using functional programming techniques is presented and described in Andrew J. Kennedy's paper, "Functional Pearls: Drawing Trees" from the Journal of Functional Programming. The paper presents four rules that an aesthetic tree must abide:

1. Two nodes at the same level should be placed at least a given distance apart.
2. A parent should be centred over its offspring.
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.
4. Identical subtrees should be rendered identically—their position in the larger tree should not affect their appearance.

1.1 Solution

The paper presents an explanation to the solution to this problem, as well as an implementation of that solution in Standard ML. In the following, we present our independent implementation of the solution in F#.

1.1.1 Representing Trees (Types)

The tree is represented in our code using polymorphism so that nodes can be of any type.

```
type Tree<'a> = Node of 'a * Tree<'a> list
```

Notice the recursiveness of the datastructure that allows to contain trees of arbitrary size in a simple type.

To represent the width of a given node, we created a datatype called Span that is a pair of floats corresponding to the left and right horizontal positions. To represent the extent of the whole Tree, we created a type Extent that is a list of Spans, i.e.

```
type Span = (float * float)
type Extent = Span list
```

Using the above datatypes, it is possible to derive the positions of the nodes as we explain in the next section. We store these positioned trees in a datatype PosTree which is similar to a Tree but each Node is a PosNode that includes a float representing the horizontal position of the Node

```
type PosTree<'a> = PosNode of 'a * float * PosTree<'a> list
```

Notice that this differs from the article, where polymorphism of the Tree type is used instead by representing the positioned tree by `Tree<('a, float)>`.

Finally, we created two datatypes that are used for the visualization of the tree. The Coordinates type is a pair of ints that represent a horizontal and vertical position given a unit measure.

```
type Coordinates = int * int
```

The AbsPosTree type is also similar to a Tree, but it includes Coordinates.

```
type AbsPosTree<'a> = AbsPosNode of 'a * Coordinates * AbsPosTree<'a> list
```

Again, here we could have used the polymorphism of the `Tree<'a>`, but instead we have chosen to define a separate type for transparency.

1.1.2 Building Trees

To construct a tree, we followed the method presented in the paper.

The basic functions used to construct the tree are:

- `moveTree` changes the horizontal position of a tree by changing the pos component of the root node.
- `moveExtent` changes the horizontal position of an extent by using a map to change the value of each span in the list.
- `mergeExtent` merges two extents that don't overlap using pattern matching to determine if either Extent list is empty, in which case we return the non-empty list, or, in the case where both lists are non-empty, we take the first float in the first span in the first list and the second float in the first span in the second list and concatenate that to a recursive call to mergeExtent using the rest of both lists.
- `mergeExtentList` performs a `mergeExtent` on a list of extents using a Fold.
- `rmax` returns the largest of two floats, used in fit to determine the minimum distance between root nodes.
- `fit` recursively determines the minimum distance between two root nodes by repeatedly taking the maximum of the distance between two nodes plus one, the minimum difference, and a recursive call to fit using the rest of the Extent.
- `fitlistl` recursively fits two trees together from the left by using pattern matching to merge each subtree.
- `fitlistr` recursively fits two trees together from the right by using pattern matching to merge each subtree, reversing the lists and the polarity of the recursive call to achieve a right-fit instead of a left-fit.
- `mean` determines the average of two floats.
- `fitlist` finds the `mean` of the left-fitted tree and the right-fitted tree to produce a tree that is fitted together and centered.

All these functions are used together in a function we call `blueprint` to build a tree

```
let rec blueprint (Node(x, xs)) =
    List.unzip (List.map blueprint xs) |> fun (ts, es) ->
    let positions           = fitlist es
    let ptrees              = List.map (fun (v,t) -> moveTree v t) (List.zip positions ts )
    let ptExtents           = List.map (fun (v,e) -> moveExtent v e) (List.zip positions es )
    let resultExtent        = (0.0, 0.0) :: mergeExtentList ptExtents
    let resulttree          = PosNode(x, 0.0, ptrees)
    (resulttree, resultExtent)
```

The blueprint takes a `Tree<'a>` as input and recursively goes through the subtrees to first identify the node postions using the extents, then move the trees and extents to the corresponding positions and subsequently append them to the resulting tree and extent of the tree and the pair of these are returned. For transparency, two helper functions are used to get the first and second arguments, i.e.

```
let designTree (t: Tree<'a>) : PosTree<'a> =
    fst (blueprint t)

let designExtents (t: Tree<'a>) : Extent =
    snd (blueprint t)
```

With the above code, it is possible to design trees that obey the four aesthetic rules stated in the beginning of this section. In the next section, we use property based testing to ensure that the rules are indeed obeyed.

2 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 Sec. 1. 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. First, we briefly describe how property based testing works with the simple case of the 'mean' function.

2.1 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 `<Property>` 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

```
let meanSymmetryProp (a,b) =
    mean (a, b) = mean (b, a)
```

such that the unit test is

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

With this we have shown a simple example on how to use PBT and some of the pitfalls of using FsCheck. Notice that in testing the symmetry property, instead of using floats we use 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'. This is done by defining a random value generator which in this case we call `NormalFloatGenerators`. Next we describe how we construct a random tree generator and use this for PBT to validate the implementation of the aesthetic rules that the tree design should obey.

2.2 Generators

For the property based test we will need to generate arbitrary trees which are not infinite, and for that we will implement a custom generator

```
let tree<'a> =
    let rec tree' s =
        match s with
```

```
        | 0 -> Gen.map (fun v -> Node(v, [])) Arb.generate<'a>
        | n when n>0 ->
            let subtrees = tree' (n/2)  |> Gen.sample 0 5 |> Gen.constant
            Gen.map2 (fun v ts -> Node(v, ts)) Arb.generate<'a> subtrees
        | _ -> invalidArg "s" "Only positive args are allowed"
    Gen.sized tree'
```

The function `tree'` will recursively generate an arbitrary tree of arbitrary type where each node has between 0 and 5 children nodes. The reason that only 5 children is allowed is that the test will be extremely slow with a higher amount.

When testing the data we need to specify a type, because we can't test generic types. So when we implement a type which implements the Arbitrary interface we will specify the type as char.

```
type TreeGenerator =
    static member Tree() =
        {new Arbitrary<Tree<char>>() with
            override x.Generator = tree<char>
            override x.Shrinker t = Seq.empty }
```

The reason we implement the TreeGenerator as an Arbitrary interface is to register it so that it will be recognized as a valid type when annotating the property based tests.

```
Arb.register<TreeGenerator>() |> ignore
```

2.3 Testing the aesthetic rules

We are now ready for testing the aesthetic rules which is done in the subsequent four subsections.

2.3.1 Rule 1

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

In order to test this rule, ew defined a function for flattening a positioned tree by recursively going through the nodes, i.e.

```
let flatten (t: Tree<'a>) =
    let rec f d px (PosNode(_, pos, cs)) = (pos+px, d) :: List.collect (f (d+1) (pos+px)) cs
    f 0 0 (designTree t) |> List.groupBy (fun (_,d) -> d)
                         |> List.map ( fun (_, xs) -> xs |> List.map (fun x -> fst x) )
```

This converts a positioned tree into a list where the index corresponds to the depth in the tree and the element is a list containing the positions of Nodes at that depth. At each depth, we then check that all nodes are at least a unit apart using the function

```
let minimum_distance_check(t: Tree<'a>) =
    let isInOrder xs = xs
                       |> Seq.pairwise
                       |> Seq.forall (fun (a, b) -> a <= b-1.0)
    flatten t |> Seq.forall isInOrder
```

This boolean function is used to describe the property that two nodes on the same level should be at least a given distance apart.

We note that we do not take the size of the node value into account. As a further requirement, one could include that the values of two nodes at the same level are not allowed to overlap. This could easily be

included by finding the maximal size of the values of all the nodes in the tree and scale the tree with this unit, e.g. by altering the unit in the `fit` function. We have left this as an exercise for the reader.

2.3.2 Rule 2

'A parent should be centred over its offspring.'

In order to test this rule, for each node, we compare the minimal and maximal positions of the subnodes and assert that the parent is indeed centered above these. Since the positions are relative to the parent this reduce to checking that the maximal position equals minus the minimal position. If this holds, we recursively go through the subtrees to check that it holds throughout the tree, i.e.

```
let rec centeringProperty (PosNode (_, _, subtrees) as tree ) =
    match subtrees with
    | [] -> true
    | sts ->
        let subtreePositions = subtrees |> List.map getSubtreePositions
        if List.min subtreePositions = - List.max subtreePositions then
            sts |> List.forall centeringProperty
        else
            false
```

This boolean function is used to describe the property that a parent should be centered above its subtrees.

2.3.3 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.'

We test this rule by noting that this corresponds to the symmetry property that changing the sign of the positions of a positioned tree should be the same as finding the positioned tree of a reversed tree. That is, by writing a function that reflect a tree, e.g.

```
let rec reflect (Node(v, subtrees)) =
    Node(v, List.map reflect (List.rev subtrees))
```

and a function that reflects a positioned tree, e.g.

```
let rec reflectpos (PosNode(v, x, subtrees)) =
    PosNode(v, -x, List.map reflectpos  (List.rev subtrees))
```

we have the symmetry property that

```
let symmetryProperty tree =
    designTree tree = reflectpos (designTree (reflect (tree)))
```

This is used to check that the drawings indeed are symmetrical with respect to reflection.

2.3.4 Rule 4

'Identical subtrees should be rendered identically—their position in the larger tree should not affect their appearance.'

We test this rule by noticing that the `blueprint` function, that creates the positioned tree, recursively goes through the subtrees to first identify the node postions using the extents, then move the trees and

extents to the corresponding positions. Due to the recursive nature, we only need to show that the extents and the trees are not altered by moving them.

3 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, i.e. trees of type `Tree<char>`, 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. As already meantioned in Sec. 2.3.1, the size of the value could easily be included, but we focus on the simple trees of type `Tree<char>` for simplicity.

3.1 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.

3.2 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 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
        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 )
```

3.3 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 conte

    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)
```

3.4 Fine tuning SVG by adding margin and centering letters

TODO

4 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).

4.1 Implementation:

Task	JRA	MM	JL	AA
Initial sketch of tree design	x	x		
Debugging tree design	x	x	x	x
Implementation of absolute positioned tree		x		
Implementation of SVG visualization		x		
Initial sketch of property based testing	x			
Simple property based test	x			
Example trees				x
Tree generator		x		
Aesthetic rule 1 test		x	x	
Aesthetic rule 2 test	x			
Aesthetic rule 3 test	x			
Aesthetic rule 4 test	x			

4.2 Report:

Task	JRA	MM	JL	AA
Design of aesthetic pleasant renderings	x	x		x
Property based testing	x	x		
Visualization	x	x		

4.3 Analysis and discussion:

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