02257 - Applied Functional Programming

DTU - Technical University of Denmark

Date of submission: June 9, 2021

Drawing Trees

Minki Chun (s202530)

Daniel F. Hauge (s201186)

Max Thrane Nielsen (s202785)

Abstract

This project presents the drawing of trees and it is based on the journal article *"FUNCTIONAL PEARLS - Drawing Trees", Andrew J. Kennedy*, which describes the application of functional programming techniques.

We explore the advantages of a functional programming language, specifically F#. The resulting product is a program that can render a tree in a aesthetically pleasing manner. Furthermore, the program is tested for its correctness with property-based testing.

1 Design of Aesthetic Pleasing Renderings

1.1 Implementation of the Design in F#

The paper [1] describes a solution to drawing trees automatically with the functional programming language SML. The article motivates the application of functional programming techniques with the claim that the design reflects the abstract solution clearer than compared to solutions with imperative languages.
The design utilizes the discriminated union type *Tree<'a>* to represent the tree data structure in a recursive definition that makes it straight forward for a functional programming language to handle recursively. The solution is build up incrementally with pure functions that takes advantage of techniques from the functional paradigm such as pattern matching and functions being first class citizen. All of the blocks are combined into the DESIGN function that recursively designs an aesthetically pleasing tree.

We made a decision to subdivide the project into modules with each module corresponding to one of the parts in the project disposition. The code for this module can be found under *Treerendering.fs*.
The main obstacle towards implementing the design, was the initial understanding of the reasoning behind the abstract solution and how functions such as FITLIST and DESIGN achieves their described results. For the actual translation there were only minor syntactically differences that we had to adjust to when we implemented the functions in F#, i.g using library functions with lists or with the syntax for doing pattern matching.
To aid ourselves in implementing the functions we decided to enforce the function signature of many of the functions by restricting the type of the input and output as a step before we began to implement the logic of the given function. This could of course be avoided as it adds more clutter to the implementation, but we thought it was a good help for understanding what a function does.
To enhance the readability of the code, we have increased the volume of some of the functions such as FITLISTL by adding local bindings to intermediate results, making it easier to see how the function works. The catch is that the functions are less concise.
After testing the program we found an issue stemming from the implementation of the FITLISTR function, which violated property 1. When we ran the test on a sample size 100 we would only pass around 70% of the tests. We could not find the bug, although we tried to reason about it. We chose to implement the alternative functions to fit from the right, and this led to all test passing.
We noticed that the FITLISTL function was not tail recursive. We have optimized it by adding an accumulator argument to make it tail recursive, by passing an empty list to begin with and accumulating the resulting values for re-positioning the extents.

2 Property Based Testing

In this section, we implemented property-based testing in order to validate that renderings satisfy four aesthetic rules and correctness that should not be broken. We can notice whether a property holds on randomly generated values and get random tree every time by using FsCheck in F#.

1. For showing two nodes at the same level should be placed at least a given distance 1, we set root position as 0.0 then calculate absolute positions from relative positions recursively by accumulating the absolute position of the root in the sub-trees. We utilize a map data

structure for storing the absolute positions of all nodes at a given depth like below:

```
// Each key represents the depth
// Value is the absolute position of the children in that depth.
type TreeMap = Map<int,float list>
```

After we compute the absolute position of a node, we appended the value to a list stored in the map with the given depth as the key. Finally checking that all positions (floats) in any depths are at least 1 part.

2. For displaying whether a parent is centered over its offspring, we checked whether FITLIST does center the parent by averaging FITLISTL and FITLISTR. (See PropertytestingOne.fs)

   We extract extents from subtrees list, then check both minimum value of FITLISTL ≥ 0 and maximum value of FITLISTR is ≤ 0 since FITLISTL range between 0 and x, and FITLISTR range between -x and 0. Additionaly, we can make sure that the mean between FITLISTL and FITLISTR is 0. (See PropertytestingTwo.fs)

3. To observe that tree drawings will be rendered symmetrically with respect to reflection, we checked two properties. One of the property is an original tree and reflected tree are still invariant by using the function in rule 1 which checks the property is fit for invariant. The other is that the original tree and the reflected tree are symmetric to each other. We prove this by validating that rule 1 and 2 still holds after reflecting positions and the order of subtrees. (See PropertytestingThree.fs)

4. The final rule complies that identical subtrees should be rendered identically. To show the position in the larger tree does not affect their structure, we put two identical trees under a hard coded tree, which we extract after the tree has been designed. Then we compare the two trees and see if their position and label are always the same since even though we put them in different places in the hard coded tree. (See PropertytestingFour.fs)

3 Translation to Post Script

This section will cover how translation from a Tree<'a> to postscript was implemented. Two distinct concatination strategies will be explored, hence there will be a risk of duplicated code. Implementing two functions that behaviorally do the same, but work different internally is very likely to cause problems if not careful.

3.1 Implementation and repeated code

To ensure consistency for the generated postscript commands, functions are made to be used regardless of version.

```
(* Command for move cursor *)
let moveTo = sprintf "%f %f moveto\n"
(* Command for displaying string *)
let label = sprintf "(%s) dup stringwidth pop 2 div neg 0 rmoveto show\n"
(* Command for drawing line *)
let lineTo = sprintf "%f %f lineto\n"
```

In order for both version to make the same exact drawings ie. movements, lines, labels etc. Functions for drawing line and labels are generalised as follows:

```
(* Creates move down, draw label, move down commands and a new position *)
let createLabel (s : 'a) (pos : position) : string*string*string*position =
    let posIn = moveTo pos.x (pos.y - labelHeight)
    let printLabel = label s
    let moveDown = moveTo pos.x (pos.y - labelHeight*1.5)
    let positionTo = {x = pos.x ; y = pos.y - labelHeight*1.5}
    (posIn, printLabel, moveDown, positionTo)

(* Creates move to below parrent, draw line and a new position *)
let createLine (relativePos : float) (pos : position) : position*string*string =
    let newPos = {x = pos.x + relativePos * 50.0 ; y = pos.y - 50.0}
    let line = lineTo newPos.x newPos.y
    let moveLine = moveTo pos.x pos.y
    (newPos, line, moveLine)
```

toPSslow was then implemented by traversing the tree, concatinating strings from the usage of createLabel and createLine along the way. toPSfast was implemented allmost the same way, but with passing a single stringbuilder around to append strings along the way. These functions can be found in PostScripRendering.fs.

An idea to reduce even more repeated code in the context of these implementations, could be to generalise the traversal of the tree, such that the only thing that differs would be the concatination function. This idea could perhaps be implemented with a higher order function. However, the way it was implemented this time was by passing a StringBuilder around extending the functions signature.

As the same experiment for functions with the same signature was to be explored, toPSfast and toPSslow was used as higher order functions to reduce time experiment code. The following code example showcases how the functions were used as higher order functions:

```
(* Time a single compuation of a function *)
let benchmarkStuff (eval:Tree<'a>->string) x : float =
    let stopWatch = System.Diagnostics.Stopwatch.StartNew()
    let _ = eval x
    stopWatch.Stop()
    stopWatch.Elapsed.TotalMilliseconds
```

To setup an experiment, we have utilized FsCheck's capabilities to generate test samples. These samples have been generated by the following code:

```
let treeGenerator = Arb.generate<Tree<string>>
let generateSamples =
    let sample = Gen.sample sampleSize sampleLength treeGenerator
    List.map (fun a -> (fst (design_tree a))) sample
```

Generalising the benchmarking function to use a higher order function trivialises running the experiment with different functions and with different samples as the following example showcases:

```
(* benchmark_PostScriptRendering runs all generated samples in "sample"
   and prints results such as mean etc. *)
let benchmark_slow sample = benchmark_PostScriptRendering toPSslow "toPSslow" sample
let benchmark_fast sample = benchmark_PostScriptRendering toPSfast "toPSfast" sample
```

3.2 Evaluation

We conducted a time experiment with different sample sizes and maximal tree sizes for toPSslow and toPSfast. The experiment reveals that using a StringBuilder is many times faster than using infix (+) string concatenation operator.

We have additionally conducted time experiments with varying length of tree sizes, which is organized below:

Sample size	Tree size	slow	fast
300	10	0.16	0.01
300	300	2.9	0.3

Table 1: Measuring time by tree size

To see the time experiment runs, see file: outputfile.txt.

4 Rendering of AST

Transforming the abstract Syntax tree provided in AST.fs to a general tree of a type like Tree<string> sounds like a lot of challenging work in a non-functional programming language. However using a functional programming language can make the transformation almost trivial. One thing that is very important, is that each discriminated type has to have a corresponding transformation function. The transformation functions will need to be recursive because the data structure of AST is recursive in nature.

1. Pattern matching is used to succinctly parse which discriminated type a discriminated union type is, and apply the correct transformation. Pattern matching is a very powerful language construct in functional programming which includes F#. The following example showcase pattern matching is used to transform the discriminated union type Exp from AST.fs.

   ```
   let rec generateExp (e:Exp) : Tree<string> =
       match e with
       | N(i) -> Node("N", [Node(string(i),[])])
       | B(b) -> Node("B", [Node(string(b),[])])
       | Access(a) -> Node("Access", [generateAccess a])
       | Addr(a) -> Node("Addr", [generateAccess a])
       | Apply(s,el) -> Node("Apply", [Node(s, [])]@List.map generateExp el)
   ```

   Using pattern matching this way ensures that all discriminated types in the given AST.fs is covered for transformation, as static analysis can warn if not all cases are covered.

2. Recursive function declaration is used to facilitate transformation of a recursive data structure. The data structure of the abstract syntax tree in AST.fs is recursive, which is why the functions that transform the AST also has to be recursive. To declare a function recursive, the keyword "rec" is used. To split recursive functions, the keyword "and" can be used such that a function declaration order do not matter.

5 Extensions

As previously stated in 1.1 we optimized the memory efficiency of the function FITLIST1 by making it tail recursive.

6 Evaluation

The final product is a working implementation of the design from [1] that produces an aesthetically pleasing rendered tree.
The translation of the program from the paper to F# went without any big obstacles, and although we deviated a bit from the implementation in the article, the semantics of the functions have stayed the same.
With property testing, we made sure that the tree is designed correctly such that when rendered, it is following the rules of pleasant visuals.
The program can translate a generel tree into PostScript. We think the resulting PostScript is absolutly adequate for a pleasant viewing experience, as we are able to easily read and understand the trees drawn with the program. There are some parts of the PostScipt translation where the functional paradigm could have been abused more, to avoid some repeated code.
Transforming an AST into a general tree was easy to do with the pattern matching language construct in F#.

References

[1] Andrew J. Kennedy, *FUNCTIONAL PEARLS - Drawing Trees*. Cambridge University Press 527-534, May 1996.

7 Appendix

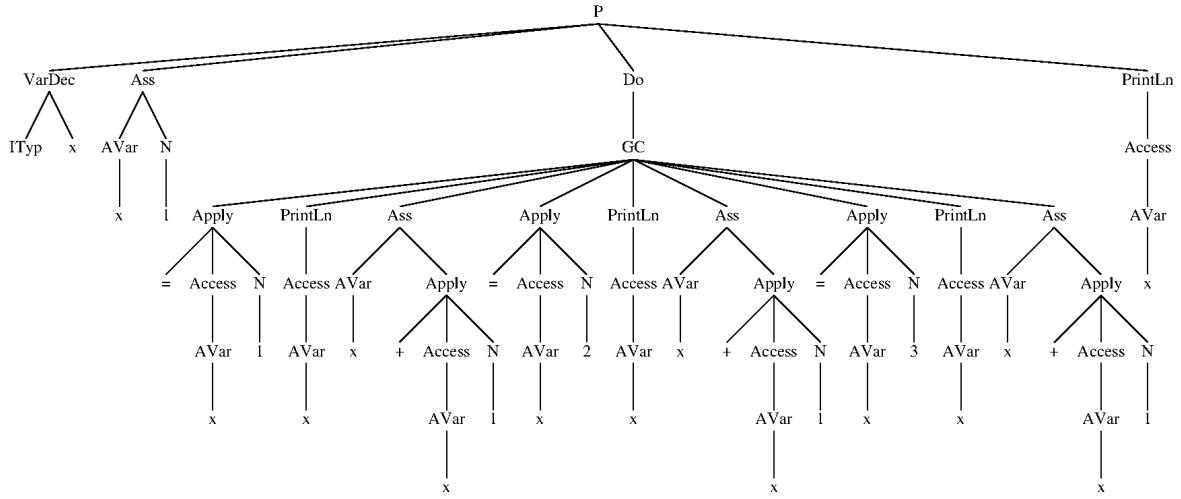

Figure 1: Rendering of an AST