Technical Report

UCAM-CL-TR-303
ISSN 1476-2986

Number 303

Drawing trees — a case study in functional programming

Andrew Kennedy

June 1993

ISSN 1476-2986

Drawing Trees —
A Case Study in Functional Programming

Andrew Kennedy
University of Cambridge
Computer Laboratory
Pembroke Street
Cambridge CB2 3QG
United Kingdom
`ajk13@cl.cam.ac.uk`

June 7, 1993

Abstract

This report describes the application of functional programming techniques to a problem previously studied by imperative programmers, that of drawing general trees automatically. We first consider the nature of the problem and the ideas behind its solution, independent of programming language implementation. The functional language implementation is described in a bottom-up style, starting with very general functions over trees and then narrowing in on the particular tree layout algorithm. Its correctness is considered informally. Finally we discuss the implementation's computational complexity and possible improvements.

1 Introduction

This short article is an attempt to demonstrate functional programming as a vehicle for the abstract description of algorithms. Whilst the functional programming community is aware of many programming techniques and ingenious data structures, little of this has found its way into papers on algorithms. This is a shame, because functional languages are an ideal means of explaining the operation of algorithms at a level which abstracts away from nitty-gritty details of pointers and loops.

Many of the papers which do describe neat functional solutions choose examples such as lambda-reduction or theorem proving—programs which imperative programmers are unlikely to write anyway. There are exceptions [BY90, FL89, Hug89, JS89, Pey85, Tri89]. The problem described in this paper, that of drawing trees, clearly is not an 'in' problem. But the functional implementation is almost embarrassing in its conciseness!

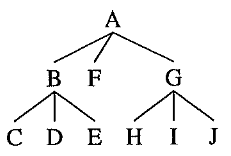
The particular functional language used is Standard ML. Any functional language could have been chosen, lazy or strict, but ML is popular, standardized and has widely-available implementations. The language is defined in [MTH89].

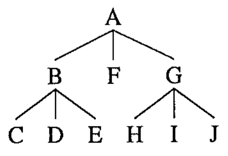
For the reader unfamiliar with ML, Paulson's book [Pau91] provides an excellent introduction to the language, functional programming, and indeed algorithms in general. Bird and Wadler's book [BW88] is particularly good at explaining *equational reasoning* about functional programs. They use a popular lazy functional language but most of the examples are easily translated into Standard ML.

2 The problem and its solution

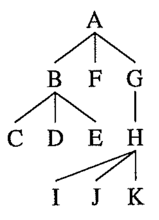
The problem is this: given a general tree, assign to each node a position on the page to give an aesthetically pleasing rendering of the tree. What do we mean by "aesthetically pleasing"? The various papers on the subject [WS79, Vau80, RT81, Wal90] list *aesthetic rules*, as follows:

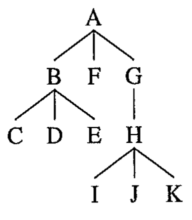
1. Nodes at the same level of the tree should be placed in the same position vertically on the page.
2. A parent should be centred over its offspring.
3. A tree and its mirror image should produce drawings that are reflections of each other. Thus symmetric trees will be rendered symmetrically. So, for example, here are two renderings, the first bad, the second good:





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

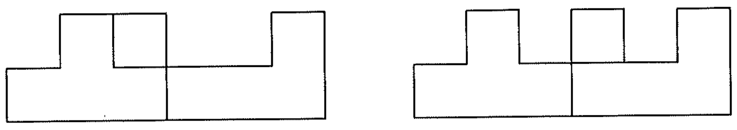




Incidentally, the Postscript used to produce these diagrams was generated by a back-end ML program. This is a messy problem, and the ML solution is messy, though not as messy as an imperative version would be.

The layout problem is solved by a decomposition into subproblems. First, draw all the subtrees of a node. Fit these together *without* changing their shape (otherwise we break rule 4). Place the subtrees one level below their parent (rule 1) and centre the parent above them (rule 2).

The critical operation is the fitting together of subtrees. Each subtree has a *profile*: an envelope around the subtree. Because we cannot distort the shape of subtrees, we simply fit these together as tightly as possible. Unfortunately, the overall positioning of the subtrees depends on the order in which we perform this fitting. For example, here are two different arrangements of the same profiles:



We can choose a left bias for this 'glueing' effect, by starting with the leftmost subtree, or a right bias instead. To satisfy rule 3, we simply do both and take the average!

3 Some general functions over trees

We first define a general tree datatype, using ML's polymorphism to parameterize the type of the node values.

```
datatype 'a Tree     = Node of 'a * ('a Tree list)
```

This simply says that a node consists of a value (of type `'a`) and a list of subtrees.

Our algorithm will accept trees with arbitrary labels (`'a Tree`) and return *positioned* trees of type (`'a*real*int Tree`). The second element of the node value represents the node's position horizontally, and the third element represents the node's depth. The root of a tree has position $(0.0, 0)$. We must use values of type `real` for the horizontal displacement because of rule 2 which forces exact division by two.

A frequently-used operation on lists is that of applying a function to each element of the list and returning the result. In most functional languages this is given the name `map`. We can write an equivalent function `maptree` for trees which we will find useful. In drawing trees it is common to apply some function to the X co-ordinate, so we have a special version of `maptree` to do just this. We then use this to define a function `translatetree` which shifts a complete tree horizontally by a fixed amount.

```
fun maptree f (Node(v, subtrees))
  = Node(f v, map (maptree f) subtrees)

fun maptreex f
  = maptree (fn (v,x,y) => (v,f x,y))

fun translatetree (x : real, t)
  = maptreex (fn x' => x+x') t
```

Another frequently-used list operation is `zip`, which takes a pair of lists and returns a list of pairs, corresponding elements paired with each other. The two lists must have the same length.

```
fun zip ([], [])         = []
  | zip (x::xs, y::ys) = (x,y) :: zip (xs,ys)
```

The analogous tree function, ziptree, takes a pair of trees and returns a tree of pairs. The two trees must have the same *shape*.

```
fun ziptree (Node(x, xs), Node(y, ys))
  = Node((x,y), map ziptree (zip (xs,ys)))
```

These two functions will have respective polymorphic types

```
'a list * 'b list -> ('a * 'b) list
```

and

```
'a Tree * 'b Tree -> ('a * 'b) Tree
```

A reflect function is very easy to define: we simply reverse the subtree list in each node using ML's built-in list function rev.

```
fun reflect (Node(v, subtrees))
  = Node(v, map reflect (rev subtrees))
```

A function to reflect a tree physically is even easier: just negate all the X co-ordinates.

```
val reflectx = maptreex (~ : real->real)
```

4 Profiles

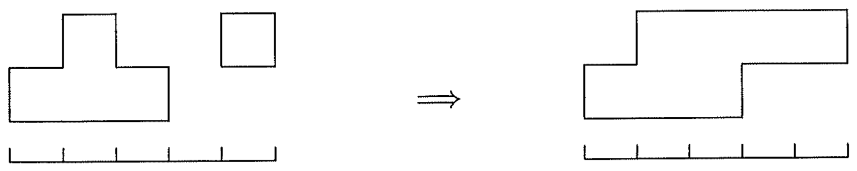
We need some way of representing the 'shape' of a tree: its *profile*. For this we use a list of pairs, the first element of which records the minimum X co-ordinate at a particular depth, and the second element records the maximum. The head of the list corresponds to the root of the tree.

It is useful to *merge* two profiles, combining their extents at each level. This we do simply by picking maxima and minima:

```
fun rmin (x1 : real, x2 : real)   = if x1<x2 then x1 else x2
fun rmax (x1 : real, x2 : real)   = if x1>x2 then x1 else x2

fun merge ([], qs)                = qs
  | merge (ps, [])                = ps
  | merge ((a,b)::ps, (c,d)::qs)
  = (rmin(a,c), rmax(b,d)) :: merge (ps, qs)
```

Notice how we must deal with profiles of different depths. We will not assume that the profiles overlap or are centred about zero. The gap between two separated profiles is filled in, as in the case shown below.



Now given a positioned tree, the following function determines its profile:

```
fun treeprofile (Node((_, x, _), subtrees))
  = (x, x) :: fold merge (map treeprofile subtrees) []
```

This is a nice example of the functional style. The extent of the root is simply `(x,x)` and we tack this onto the result of merging the profiles of all the subtrees. The functional `fold` is used to apply the binary operation `merge` between all subtrees. Informally, it is defined as:

$$\texttt{fold}\ (\oplus)\ [x_1, x_2, \ldots, x_n]\ a = x_1 \oplus (x_2 \oplus (\cdots(x_n \oplus a)\cdots))$$

where $\oplus$ is a two argument function written as an infix operator which associates to the *right*. We could have used a left-associative version instead because `merge` is associative.

A trivial function to shift a profile horizontally is convenient:

```
fun translate (x : real) = map (fn (a,b) => (a+x,b+x))
```

Now the real work begins. First we define a function which determines how close two trees may be placed next to each other, given that the minimum separation between two nodes in different subtrees is a constant value of 1.0. Of course when we actually draw the tree this is scaled appropriately. The function accepts two profiles as arguments and returns the minimum distance between the two root nodes.

```
fun findgap ((_,b)::xs) ((a,_)::ys) = rmax(findgap xs ys, b-a + 1.0)
  | findgap _           _           = 0.0
```

Now we extend this function to a list of subtrees. Given an initial profile p, the function repeatedly fits trees against this profile and returns a list containing the minimum distance between each tree and this initial profile. The asymmetry in the tree layout algorithm derives from this function: trees are fitted together *from the left.*

```
fun findgaps p []
  = []

  | findgaps p (q::qs)
  = let val gap = findgap p q
        val newp = merge p (translate gap q)
    in
      gap :: findgaps newp qs
    end
```

5 Drawing the tree

We will use `findgaps` to give us the positions of offspring relative to the first child. We must centre these to satisfy rule 2. The function `centre` assumes that these offsets start at zero.

```
fun centre xs =
  case rev xs of
    []     => []
  | (w::_) => map (fn x => x - w / 2.0) xs
```

Now we wrap everything together in one function.

```
fun planleft tree =
let
  fun plan' level (Node(v, subtrees))
  = let val ptrees  = map (plan' (level+1)) subtrees
        val offsets = centre (findgaps [] (map treeprofile ptrees))
        val ctrees  = map translatetree (zip (offsets, ptrees))
    in
      Node((v, 0.0, level), ctrees)
    end
in
  plan' 0 tree
end
```

The local function `plan'` produces a positioned tree with the root at position $(0.0, \texttt{level})$. It does this in the following stages. First, recursively draw all the subtrees (in `ptrees`). All the subtrees' roots will be at position $(0.0, \texttt{level} + 1)$. Calculate their profiles and fit them together using `findgaps`. Then adjust the offsets so that they are centred around zero, and translate each subtree by the appropriate offset to give `ctrees`. That's it!

To produce a layout with the asymmetry in the other direction, we do the following:

1. Reflect the tree structurally.
2. Lay out the tree using the left-biased algorithm.
3. Reflect the tree structurally.
4. Reflect the tree physically.

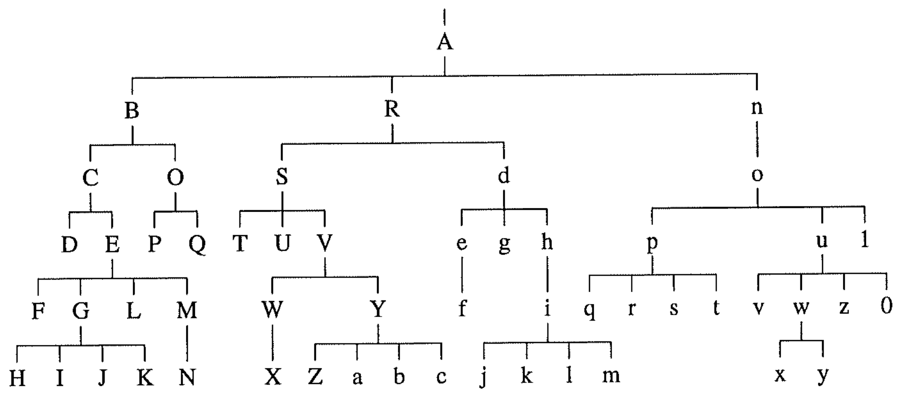
Function composition is used to glue these processes together:

```
val planright = reflectx o reflect o planleft o reflect
```

Now we just take the average of the two cases to generate an unbiased layout with the properties we require.

```
fun average ((v,x1,y), (_,x2,_)) = (v, (x1 + x2) / 2.0, y)
fun plan tree =
  maptree average (ziptree (planleft tree, planright tree))
```

Here is a realistic example, in family tree form with all connecting lines horizontal or vertical.



6 Correctness

In contrast with a coding of this algorithm in Pascal[Wal90], it is clear from the ML code that our aesthetic rules are not broken. Consider them each in turn.

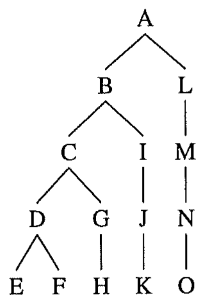
1. The `plan'` function produces a tree with its root at vertical position `level`. It calls itself recursively with the same value (`level+1`) for all subtrees.

   It is clear that reflecting the tree either structurally or physically leaves the Y co-ordinates alone, as does the final averaging operation. Hence the condition is satisfied.

2. The centreing of parent over offspring occurs in `centre`.

   It is possible to use integer values instead of reals if we're not concerned about truncation errors causing this rule to be broken. Alternatively, to guarantee that `w` is always even, we can set the minimum separation between subtrees to 2^{n-1}, where n is the maximum depth of the tree. This is somewhat messy, and could be done automatically by `plan` before passing it in as an argument to `planleft` and `planright`.

   A pathological case, where we really *do* need a separation value of 2^{n-1}, is illustrated below.



3. The mirror image property is forced by our final averaging function. We are asking for the following equation to be satisfied:

   $$\texttt{plan}\, t = \texttt{reflect}\ (\texttt{reflectx}\ (\texttt{plan}\ (\texttt{reflect}\ t)))$$

   There is a fairly easy proof of this using the standard principles of equational reasoning and structural induction as described in [BW88, Pau91].

4. The subtree consistency property is self-evident from the recursive nature of the algorithm. A recursive call to `plan'` is used to draw the subtrees, and the subsequent manipulation using `translatetree` does not affect their physical structure.

7 Complexity

The function `plan` has worst case $O(n^2)$ time behaviour, where n is the number of nodes in the tree. This is due to the repeated use of `translatetree`, which traverses the entire tree in order to displace it horizontally. To reduce this complexity to $O(n)$ we can store *relative* displacements in the nodes instead of absolute ones, so that `translatetree` uses constant time. This requires a certain degree of plumbing in the other functions to accumulate an absolute displacement when one is required.

The motivation behind this article was to show how a first-attempt, *correct* solution to the problem could be knocked up quickly and elegantly. It would be interesting to apply some of the *program transformation* techniques that have been developed to introduce the improvement just mentioned.

8 Conclusion

It is hoped that this case study has once more highlighted the elegance and versatility of functional programming. For readers unfamiliar with the ideas of functional languages, the code must seem very strange but suggestive—a flavour to entice you to the references listed below!

References

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[BY90] F. Warren Burton and Hsi-Kai Yang. Manipulating multilinked data structures in a pure functional language. *Software—Practice and Experience*, 20(11):1167–1185, November 1990.

[FL89] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, 32(2), April 1989.

[Hug89] R. J. M. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

[JS89] S. B. Jones and A. F. Sinclair. Functional programming and operating systems. *The Computer Journal*, 32(2), April 1989.

[MTH89] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Mass., 1989.

[Pau91] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[Pey85] Simon L. Peyton Jones. Yacc in Sisal—an exercise in functional programming. *Software—Practice and Experience*, 15(8):807–820, August 1985.

[RT81] Edward M. Reingold and John S. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, SE-7(2):223–228, March 1981.

[Tri89] Phil Trinder. Referentially transparent database languages. In *Glasgow Workshop on Functional Programming*, Fraserburgh, Scotland, August 1989. Springer-Verlag.

[Vau80] Jean G. Vaucher. Pretty-printing of trees. *Software—Practice and Experience*, 10:553–561, 1980.

[Wal90] John Q. Walker II. A node-positioning algorithm for general trees. *Software—Practice and Experience*, 20(7):685–705, July 1990.

[WS79] Charles Wetherell and Alfred Shannon. Tidy drawings of trees. *IEEE Transactions on Software Engineering*, SE-5(5):514–520, September 1979.