J. Functional Programming **6** (3): 527–534, May 1996

FUNCTIONAL PEARLS

Drawing Trees

ANDREW J. KENNEDY

University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street
Cambridge CB2 3QG, United Kingdom

Abstract

This article 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 (due to Radack), independent of programming language implementation. We then describe a Standard ML program which reflects the structure of the abstract solution much better than an imperative language implementation. We conclude with an informal discussion on the correctness of the implementation and some changes which improve the algorithm's worst-case time complexity.

1 The problem and its solution

The problem is this: given a labelled tree, assign to each node a position on the page to give an aesthetically pleasing rendering of the tree. We assume that nodes at the same depth are positioned on the same horizontal line on the page, so the problem reduces to finding a position horizontally for each node. But what do we mean by "aesthetically pleasing"? The various papers on the subject (Radack, 1988; Wetherell and Shannon, 1979; Vaucher, 1980; Reingold and Tilford, 1981; Walker, 1990) list *aesthetic rules* which constrain the positions in a number of ways. We adopt the same rules as Radack and Walker:

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. So, for example, Figure 1 shows 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. In Figure 2 the tree on the left fails the test, and the one on the right passes.

Finally, trees should be as narrow as possible without violating these rules.

Fig. 1. A symmetric tree rendered in two ways

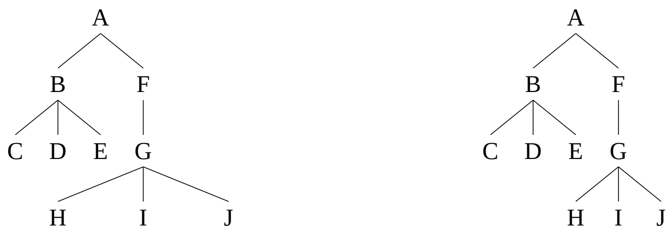

Fig. 2. A tree with two identical subtrees

The layout problem is solved as follows. First, draw all the subtrees of a node in such a way that none of the rules are broken. Fit these together *without* changing their shape (otherwise rule 4 is broken), and in such a way that rules 1 and 3 are not broken. Finally centre their parent above them (rule 2) and the layout is complete.

The critical operation is the fitting together of subtrees. Each subtree has an *extent*—an envelope around the subtree. Because the shape of the subtrees must not be distorted, their extents are simply fitted together as tightly as possible. Unfortunately, the overall positioning of the subtrees depends on the order we choose to perform this fitting. Figure 3 shows two different arrangements of the same extents.

We can choose a left bias for this 'gluing' effect, by starting with the leftmost subtree, or a right bias instead. To satisfy rule 3, we simply do both and take the average; this approach was also taken by Radack.

In the rest of the article some familiarity with a functional language is assumed. We use Standard ML (Paulson, 1991; Milner *et al.*, 1989), but any functional language, strict or lazy, would do just as well.

Fig. 3. Two arrangements

2 Representing trees

First we define a general tree datatype, using ML's polymorphism to parameterise 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 of type `'a Tree` and return *positioned* trees of type `('a*real) Tree`. The second element of the node value represents the node's horizontal position, *relative to its parent*. Rule 2 suggests that we should use `real` values for this purpose; in fact, rationals with finite binary representations would suffice.

Because we have chosen to use relative positions, the operation of displacing a tree horizontally can be done in constant time:

```
fun movetree (Node((label, x), subtrees), x' : real) =
  Node((label, x+x'), subtrees)
```

3 Representing extents

The extent of a tree is represented by a list of pairs:

```
type Extent = (real*real) list
```

The first component of each pair records the leftmost horizontal position at a particular depth, and the second component records the rightmost. The head of the list corresponds to the root of the tree. In contrast with the tree representation, the positions in an extent are *absolute*.

A trivial function to move an extent horizontally will be useful:

```
fun moveextent (e : Extent, x) = map (fn (p,q) => (p+x,q+x)) e
```

It will also be necessary to *merge* two non-overlapping extents, filling in the gap between them. This is done simply by picking the leftmost positions of the first extent and the rightmost positions of the second:

```
fun merge ([], qs)                 = qs
  | merge (ps, [])                 = ps
  | merge ((p,_)::ps, (_,q)::qs) = (p,q) :: merge (ps, qs)
```

Notice how we must deal with extents of different depths.

This operation can be extended to a list of extents by the following function:

```
fun mergelist es = fold merge es []
```

This is a nice example of the functional style. The functional `fold` is used to apply the binary operation `merge` between all extents in the list. Informally, it is defined as:

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

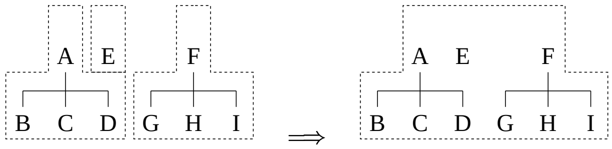

Fig. 4. Merging extents

where $\oplus$ is a two argument function written as an infix operator which associates to the *right*. We could have used a left-associating version of `fold` instead because `merge` is associative. Readers familiar with Haskell or another functional programming language should note carefully the order of the arguments to `fold`: this is the order used in most implementations of Standard ML.

An example of the use of `mergelist` is shown in Figure 4.

4 Fitting extents

First we define a function which determines how close to each other two trees may be placed, assuming a minimum node separation of 1. Of course when the tree is drawn this is scaled appropriately. The function accepts two extents as arguments and returns the minimum possible distance between the two root nodes:

```
fun rmax (p : real, q : real)   = if p > q then p else q
fun fit ((_,p)::ps) ((q,_)::qs) = rmax(fit ps qs, p - q + 1.0)
  | fit _           _           = 0.0
```

Now we extend this function to a list of subtrees, calculating a list of positions for each subtree relative to the leftmost subtree which has position zero. It works by accumulating an extent, repeatedly fitting subtrees against it. This produces an asymmetric effect because trees are fitted together *from the left.*

```
fun fitlistl es =
let
  fun fitlistl' acc [] = []
    | fitlistl' acc (e::es) =
    let val x = fit acc e
    in
      x :: fitlistl' (merge (acc, moveextent (e,x))) es
    end
in
  fitlistl' [] es
end
```

The opposite effect is produced from the following function which calculates positions relative to the rightmost subtree, which has position zero. The function `rev` is ordinary list reversal, and `~` is negation.

```
fun fitlistr es =
let
  fun fitlistr' acc [] = []
    | fitlistr' acc (e::es) =
    let val x = ~(fit e acc)
    in
      x :: fitlistr' (merge (moveextent (e,x), acc)) es
    end
in
  rev (fitlistr' [] (rev es))
end
```

Alternatively, it is possible to define `fitlistr` in terms of `fitlistl` by the following composition of functions:

```
val flipextent : Extent -> Extent = map (fn (p,q) => (~q,~p))
val fitlistr = rev o map ~ o fitlistl o map flipextent o rev
```

In order to obtain a symmetric layout we calculate for each subtree the mean of these two positionings:

```
fun mean (x,y) = (x+y)/2.0
fun fitlist es = map mean (zip (fitlistl es, fitlistr es))
```

5 Designing the tree

We are now ready to combine these elements into a single function `design` which accepts a labelled tree and returns a positioned tree with the root at zero. In fact, we will use an auxiliary function `design'` which also returns the extent of the tree. This saves us from recalculating extents unnecessarily.

```
fun design tree =
let
  fun design' (Node(label, subtrees)) =
  let
    val (trees, extents) = unzip (map design' subtrees)
    val positions        = fitlist extents
    val ptrees           = map movetree (zip (trees, positions))
    val pextents         = map moveextent (zip (extents, positions))
    val resultextent     = (0.0, 0.0) :: mergelist pextents
    val resulttree       = Node((label, 0.0), ptrees)
  in
    (resulttree, resultextent)
  end
in
  fst (design' tree)
end
```

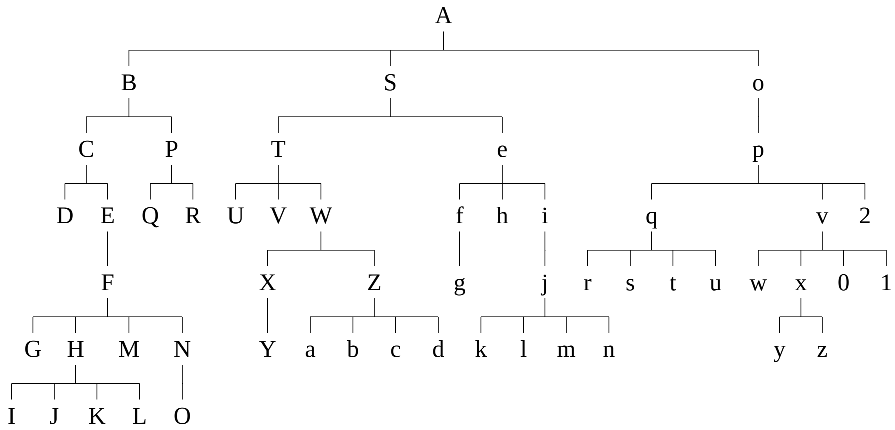

Fig. 5. An example rendering

It works as follows. First, recursively design all the subtrees. This results in a list of (*tree*, *extent*) pairs, which we `unzip` into two lists. All the subtrees' roots will be at position zero. Next fit the extents together using `fitlist`, giving a list of displacements in `positions`. Then move each subtree in `trees` by its corresponding displacement in `positions` to give `ptrees`, and do the same for the extents to give `pextents`. Finally calculate the resulting extent and resulting tree with its root at position 0. That's it!

Figure 5 shows a realistic example, in family tree form with all connecting lines horizontal or vertical. Incidentally, the PostScript used to produce these diagrams was generated by a back-end ML program.

6 Correctness

In contrast with previous algorithms which solve the tree-drawing problem using an imperative language, it is clear from the ML code that our aesthetic rules are not broken. Consider them each in turn.

1. The function `fit` ensures that the positioning of tree extents by `fitlistl`, `fitlistr` and `fitlist` places nodes at least a scale unit apart. A formal proof would entail showing that if the nodes are listed in breadth-first order then the positions $x_1, \ldots, x_n$ at any level have the property that for $1 \leq i < n$, $x_i + 1 \leq x_{i+1}$.
2. From the symmetry in the definitions of `fitlistl` and `fitlistr` it can be seen that if the positions assigned by `fitlistl` range between 0 and x then the positions assigned by `fitlistr` will range between $-x$ and 0. Hence when these are averaged by `fitlist` the parent (at position 0) will be centred above its children. This could be proved formally without much trouble; to do the same for imperative code would be much harder.

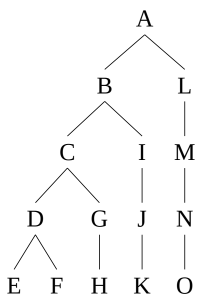

Fig. 6. A pathological case

It is possible to use integer values instead of reals if we are not concerned about truncation errors causing this rule to be broken. Alternatively, we can set the minimum separation between subtrees to 2^{n-1}, where n is the maximum depth of the tree. A pathological case, where we really *do* need a separation value of 2^{n-1}, is illustrated in Figure 6, scaled appropriately.

3. The mirror image property is forced by taking the mean of left and right-biased positionings of subtrees. We are asking for the following equation to be satisfied:

$$\text{For all trees } t,\ \texttt{design}\ t = \texttt{reflect}(\texttt{reflectpos}(\texttt{design}(t)))$$

where `reflect` is a function which reflects a tree structurally, and `reflectpos` is a function which reflects the node positions about zero. They are defined as follows:

```
fun reflect (Node(v, subtrees)) =
  Node(v, map reflect (rev subtrees))
fun reflectpos (Node((v,x : real), subtrees)) =
  Node((v,~x), map reflectpos subtrees)
```

Again this could be proved formally using equational reasoning and structural induction, as described in any good text on functional programming (Paulson, 1991; Bird and Wadler, 1988).

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

The tree designed could be no narrower without violating these rules because `fitlist` fits extents together as tightly as possible without distorting the shapes of the subtrees but leaving a gap of at least one unit between adjacent nodes.

7 Complexity

The program as presented uses $O(n^2)$ time in the worst case, where n is the number of nodes in the tree. Fortunately it is possible to transform the program to a linear-time one with some loss of clarity.

The inefficiency arises in the representation of extents. Moving a tree uses constant time, due to the use of relative positions, but moving an extent uses linear time because it is represented using absolute positions. Changing to relative positions would reduce the complexity of `mergelist` from quadratic to linear. Unfortunately the functions `fit` and `merge` become rather less elegant, though it is an easy exercise to define them. They are also good candidates for formal derivation (Gibbons, 1991; Gibbons, 1996).

Acknowledgements

I am grateful to Nick Benton for several fruitful discussions, and to one of the referees whose comments helped improve the presentation of this paper.

References

R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

J. Gibbons. Algebras for Tree Algorithms. DPhil thesis, Oxford University Computing Laboratory, 1991.

J. Gibbons. Deriving tidy drawings of trees. *Journal of Functional Programming*, this issue.

R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Mass., 1989.

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

G. M. Radack. Tidy drawing of M-ary trees. Technical Report CES-88-24, Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio, November 1988.

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

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

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

C. Wetherell and A. Shannon. Tidy drawings of trees. *IEEE Transactions on Software Engineering*, SE-5(5):514–520, September 1979.