Drawing Trees

Andreas Halkjær From   Anders Schlichtkrull   Jørgen Villadsen

DTU Compute, AlgoLoG, Technical University of Denmark, 2800 Kongens Lyngby, Denmark

Abstract

We formally prove in Isabelle/HOL two properties of an algorithm for laying out trees visually. The first property states that removing layout annotations recovers the original tree. The second property states that nodes are placed at least a unit of distance apart. We have yet to formalize three additional properties: That parents are centered above their children, that drawings are symmetrical with respect to reflection and that identical subtrees are rendered identically.

1 Introduction

We consider the functional pearl Drawing Trees by Andrew J. Kennedy [1] and formalize it in Isabelle/HOL. The formalization is available online.[1] The paper presents a functional program for laying out trees in an "aesthetically pleasing" way according to four rules. Quoting Kennedy, the properties are as follows: [1, p. 527]

1. Two nodes at the same level should be placed at least a given distance apart.
2. A parent should be centered 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 [1, p. 528] 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 [1, p. 528] the tree on the left fails the test, and the one on the right passes.

We have implemented the algorithm in Isabelle using *Complex_Main* since the algorithm uses real numbers for node offsets. Our formalization includes a proof of property 1 as well as a proof of structural preservation. We construct a counterexample to property 3 as it is stated in the paper and suggest a remedy, which we leave to future work along with properties 2 and 4.

The paper is structured as follows. First we give a brief overview of the layout algorithm as implemented in Isabelle (§ 2). We show that the algorithm is equivalent to a slower but simpler definition (§ 3). This is used to show that layouting preserves the structure of the original tree (§ 4) and that nodes are spaced at least a unit apart (§ 5). Then we show a counterexample to property 3 as stated (§ 6) before finally concluding (§ 7).

We are unaware of other work on formally verifying aesthetic properties of layout algorithms.

[1] `http://www.student.dtu.dk/~s144442/Drawing_Trees.thy`

2 Algorithm

Kennedy presents an implementation in Standard ML. We have translated this to corresponding Isabelle definitions, cf. Figure 1 — these can be code generated back to Standard ML if needed.

The algorithm takes a tree as input and annotates each node with a horizontal offset relative to its parent. Vertical offsets between levels are implicit. The main function, *design'*, returns both the annotated tree and its extent. An extent is a list of horizontal ranges, one for each level of the tree, denoting the space taken up by the tree. Extents use absolute coordinates. The body of *design'* works by calculating the annotated subtrees and their extents recursively, then, based on these extents, calculating the horizontal offsets that make the subtrees fit after each other via *fitlist* and finally doing some bookkeeping to return the correct extent for itself.

For fitting, *fitlist* tries to fit the given extents from both the left and right, and then takes an average to produce a balanced layout.

We have to define *unzip* ourselves, as it is not currently included in Isabelle. We use the efficient definition given by Kennedy, but show it equivalent to a version using built-in functions on lists to reason about it more easily:

lemma *unzip* [*simp*]: ‹*unzip xs* = (*map fst xs*, *map snd xs*)›
by (*induct xs*) *auto*

The running time of the presented algorithm is quadratic in the size of the tree, as we use absolute positions for extents and move these in every recursive call. Kennedy notes that it can be made linear by using relative positions, as for the offsets, but that the definitions become "rather less elegant" [1, p. 534]. We have opted for the elegant version here.

3 Simpler Definition

The algorithm calculates the extent on-the-fly for the sake of performance, but this makes it harder to reason about than doing it explicitly every time. Therefore we prove it equivalent to the following slower but simpler definition.

First, we need a way to calculate the extent of a tree:

fun *extent-of-tree* :: ‹(*'a* ∗ *real*) *tree* ⇒ *extent*› **where**
‹*extent-of-tree* (*Node* (-, *offset*) *subs*) =
(*offset*, *offset*) # *mergelist* (*map* (λ*t*. *moveextent* (*extent-of-tree t*, *offset*)) *subs*)›

This can then be used to obtain the extents of subtrees in a simpler version of *design'*, dubbed *raw-design*:

primrec *raw-design* :: ‹*'a tree* ⇒ (*'a* ∗ *real*) *tree*› **where**
‹*raw-design* (*Node label subtrees*) = (
let trees = *map raw-design subtrees*;
extents = *map extent-of-tree trees*;
positions = *fitlist extents*;
ptrees = *map movetree* (*zip trees positions*)
in Node (*label*, *0*) *ptrees*)›

We prove that the two new definitions are functionally equivalent to *design'*:

theorem *design'-raw-design*:
‹*design' t* = (*raw-design t*, *extent-of-tree* (*raw-design t*))›

```
datatype 'a tree = Node 'a ‹'a tree list›

type-synonym extent = ‹(real * real) list›

fun movetree :: ‹('a * real) tree * real ⇒ ('a * real) tree› where
  ‹movetree ((Node (label, x) subs), x') = Node (label, x + x') subs›

primrec moveextent :: ‹extent * real ⇒ extent› where
  ‹moveextent (e, x) = map (λ (p, q) ⇒ (p + x, q + x)) e›

fun merge :: ‹extent ⇒ extent ⇒ extent› where
  ‹merge [] qs = qs›
| ‹merge ps [] = ps›
| ‹merge ((p1, p2) # ps) ((q1, q2) # qs) = (min p1 q1, max p2 q2) # merge ps qs›

primrec mergelist :: ‹extent list ⇒ extent› where
  ‹mergelist [] = []›
| ‹mergelist (e#es) = merge e (mergelist es)›

fun fit :: ‹extent ⇒ extent ⇒ real› where
  ‹fit ((p1,p2)#ps) ((q1,q2)#qs) = max (fit ps qs) (max p1 p2 − min q1 q2 + 1)›
| ‹fit - - = 0›

primrec fitlistl' :: ‹extent ⇒ extent list ⇒ real list› where
  ‹fitlistl' acc [] = []›
| ‹fitlistl' acc (e#es) = (let x = fit acc e in x # fitlistl' (merge acc (moveextent (e, x))) es)›

definition fitlistl where ‹fitlistl ≡ fitlistl' []›

definition flipextent :: ‹extent ⇒ extent› where
  ‹flipextent = map (λ(p, q). (−q, −p))›

definition fitlistr :: ‹extent list ⇒ real list› where
  ‹fitlistr = rev o map uminus o fitlistl o map flipextent o rev›

definition fitlist :: ‹extent list ⇒ real list› where
  ‹fitlist es = map mean (zip (fitlistl es) (fitlistr es))›

fun unzip :: ‹('a × 'b) list ⇒ 'a list × 'b list› where
  ‹unzip [] = ([], [])›
| ‹unzip ((a, b) # xs) = (case unzip xs of (as, bs) ⇒ (a # as, b # bs))›

primrec design' :: ‹'a tree ⇒ ('a * real) tree * extent› where
  ‹design' (Node label subtrees) = (
    let (trees, extents) = unzip (map design' subtrees);
        positions = fitlist extents;
        ptrees = map movetree (zip trees positions);
        pextents = map moveextent (zip extents positions);
        resultextent = (0, 0) # mergelist pextents;
        resulttree = Node (label, 0) ptrees
    in (resulttree, resultextent))›

definition design where ‹design t ≡ fst (design' t)›
```

Figure 1: The layout algorithm

4 Property 0 — Structure Preservation

We should be able to strip away the annotated offsets and get the same tree back. In other words, the following function should cancel *design*:

fun *strip-offsets* :: ⟨(′*a* × *real*) *tree* ⇒ ′*a tree*⟩ **where**
⟨*strip-offsets* (*Node* (*label*, *offset*) *subs*) = *Node label* (*map strip-offsets subs*)⟩

And it does:

theorem *strip-offsets-design*: ⟨*strip-offsets* (*design t*) = *t*⟩
unfolding *design-def* **using** *strip-offsets-design′* **by** (*metis prod.collapse*)

This immediately gives us that *design* is injective:

theorem *design-inj*: ⟨$t = t' \longleftrightarrow$ *design t* = *design t′*⟩
using *strip-offsets-design* **by** *metis*

5 Property 1 — Spacing

We prove that all nodes are offset in such a way that they are at least one unit apart from every other node on that level in the tree. This is proved first for extents and then for nodes. We use the following definition to check that two extents are properly spaced:

fun *spaced* :: ⟨*extent* ⇒ *extent* ⇒ *bool*⟩ **where**
⟨*spaced* ((*p1*, *p2*) # *ps*) ((*q1*, *q2*) # *qs*) = (*q1* − *p2* ≥ *1* ∧ *spaced ps qs*)⟩
| ⟨*spaced* - - = *True*⟩

It is useful to consider extents within other extents, as this gives us a more readily applicable induction hypothesis for functions that accumulate extents. We define the following:

fun *within-extent* :: ⟨*extent* ⇒ *extent* ⇒ *bool*⟩ **where**
⟨*within-extent* ((*p1*, *p2*) # *ps*) ((*q1*, *q2*) # *qs*) = (*q1* ≤ *p1* ∧ *p2* ≤ *q2* ∧ *within-extent ps qs*)⟩
| ⟨*within-extent* [] - = *True*⟩
| ⟨*within-extent* - - = *False*⟩

For instance, if *ps* is contained within *xs* and we fit *es* relative to *xs*, then *ps* automatically becomes spaced correctly with regards to *es*:

lemma *fitlistl′-spaced-within*:
⟨*within-extent ps xs* ⟹ *list-all* (*spaced ps*) (*map moveextent* (*zip es* (*fitlistl′ xs es*)))⟩

To check every extent relative to every other extent to the right of it, we use the function *all-pairs*:

primrec *all-pairs* :: ⟨(′*a* ⇒ ′*a* ⇒ *bool*) ⇒ ′*a list* ⇒ *bool*⟩ **where**
⟨*all-pairs* - [] = *True*⟩
| ⟨*all-pairs p* (*e* # *es*) = (*list-all* (*p e*) *es* ∧ *all-pairs p es*)⟩

definition *all-spaced* :: ⟨*extent list* ⇒ *bool*⟩ **where**
⟨*all-spaced* = *all-pairs spaced*⟩

Thus we can prove that *fitlistl* spaces extents correctly:

lemma *all-spaced-fitlistl*: ⟨*all-spaced* (*map moveextent* (*zip es* (*fitlistl es*)))⟩
unfolding *fitlistl-def* **using** *all-spaced-fitlistl′* **.**

We reverse the order of extents and flip them around in the definition of *fitlistr*, so the following property relating these operations is useful:

```
lemma spaced-move-flip:
  ‹spaced (moveextent (flipextent qs, y)) (moveextent (flipextent ps, x)) =
   spaced (moveextent (ps, −x)) (moveextent (qs, −y))›
  unfolding flipextent-def by (induct ps qs rule: spaced.induct) auto
```

This and a few other lemmas allow us to prove that *fitlistr* spaces extents correctly:

```
lemma all-spaced-fitlistr:
  ‹all-spaced (map moveextent (zip es (fitlistr es)))›
```

If moving an extent by either of two values spaces it correctly, then the mean of those two will also space it correctly:

```
lemma spaced-mean:
  ‹spaced (moveextent (ps, x)) (moveextent (qs, y)) ⟹
   spaced (moveextent (ps, a)) (moveextent (qs, b)) ⟹
   spaced (moveextent (ps, mean (x, a))) (moveextent (qs, mean (y, b)))›
  by (induct ps qs rule: spaced.induct) (simp-all add: add-divide-distrib)
```

And building on this gives us correct spacing for *fitlist*:

```
lemma all-spaced-fitlist: ‹all-spaced (map moveextent (zip es (fitlist es)))›
```

Finally we define what it means for a tree to be properly spaced:

```
fun get-offset :: ‹('a ∗ real) tree ⇒ real› where
  ‹get-offset (Node (-, x) -) = x›

definition spaced-offset :: ‹real ⇒ real ⇒ bool› where
  ‹spaced-offset x y = (x + 1 ≤ y)›

primrec spaced-tree :: ‹('a ∗ real) tree ⇒ bool› where
  ‹spaced-tree (Node - subs) =
    (all-pairs spaced-offset (map get-offset subs) ∧ list-all spaced-tree subs)›
```

If the extents of two trees are properly spaced, then so are the trees' offsets:

```
lemma spaced-extents-offsets:
  assumes ‹spaced (extent-of-tree s) (extent-of-tree t)›
  shows ‹spaced-offset (get-offset s) (get-offset t)›
  unfolding spaced-offset-def using assms
  by (cases s rule: get-offset.cases, cases t rule: get-offset.cases) simp-all
```

Next we prove that *raw-design* spaces correctly. We get proper spacing for the subtrees by the induction hypothesis, then we show that fitting them together preserves that spacing:

```
lemma spaced-raw-design: ‹spaced-tree (raw-design t)›
proof (induct t)
  case (Node v subs)
  define trees where ‹trees ≡ map raw-design subs›
  define extents where ‹extents ≡ map extent-of-tree trees›
  define positions where ‹positions ≡ fitlist extents›

  have ‹list-all spaced-tree trees›
```

```
    unfolding trees-def using Node by (induct subs) simp-all
  then have ‹list-all spaced-tree (map movetree (zip trees positions))›
    using move-spaced-trees by blast
  moreover from this have ‹all-spaced (map extent-of-tree (map movetree (zip trees positions)))›
    unfolding extents-def positions-def
    using Node all-spaced-fitlist all-spaced-extent-of-fitted-tree by blast
  moreover have ‹?case = spaced-tree (Node (v, 0) (map movetree (zip trees positions)))›
    unfolding trees-def extents-def positions-def by simp
  ultimately show ?case
    using all-spaced-tree by metis
qed
```

And through the equivalence between the two definitions, we also obtain the property for the faster one:

```
theorem spaced-design: ‹spaced-tree (design t)›
  using design-raw-design spaced-raw-design by metis
```

6 Property 3 — Mirror Image Property

Kennedy defines the following two functions:

```
fun reflect :: ‹'a tree ⇒ 'a tree› where
  ‹reflect (Node v subtrees) = Node v (map reflect (rev subtrees))›

fun reflectpos :: ‹('a ∗ real) tree ⇒ ('a ∗ real) tree› where
  ‹reflectpos (Node (label, offset) subtrees) = Node (label, −offset) (map reflectpos subtrees)›
```

And states that for all trees t, it should hold that *design t = reflect (reflectpos (design t))* [1, p. 533]. The function *reflect* reverses the positions of subtrees, while *reflectpos* mirrors the tree's offsets horizontally. But since the drawing is based on the offsets, and only *reflectpos* changes these, this equation will not hold for any asymmetric tree. We can therefore prove the negation of Kennedy's claim using a specific counterexample:

```
lemma ¬(∀ t. design t = reflect (reflectpos (design t)))
proof −
  let ?t = Node undefined [Node undefined [Node undefined [Node undefined [], Node undefined []]],
              Node undefined [Node undefined []]]
  have ¬ (design ?t = reflect (reflectpos (design ?t)))
    by normalization
  then show ?thesis
    by blast
qed
```

What is actually meant is probably that *design t* $\approx$ *reflectpos (design (reflect t))* should hold for all trees t. Here we first reflect the tree structurally, then design it, and then mirror the produced offsets. By $\approx$ we mean that the trees are drawn equally, not that they are equal as Isabelle/ML values, since the children will be in opposite order because of the structural reflection. Formalizing this property is a work-in-progress.

7 Conclusion

It is notable how easily a functional algorithm as the one given by Kennedy [1] can be formalized in Isabelle. Furthermore, Isabelle can easily deal with real numbers in a practical application as this one.

Proving the algorithm equivalent to a slower but simpler version helped tremendously in the formalization. It would have helped us if *unzip* was included in Isabelle along with various lemmas similar to how *zip* or *map* as in the Isabelle distribution.

For future work, the rest of the properties should be formally proved as well. While Kennedy states that this can easily be done for most of them [1, p. 533], doing so in practice has turned out to be non-trivial and even revealed an error in the specification of property 3. This showcases the value of formalizing seemingly obvious properties.

References

[1] Andrew J. Kennedy. "Functional Pearls: Drawing Trees". In: *Journal of Functional Programming* 6.3 (1996), pp. 527–534. DOI: `10.1017/S0956796800001830`. URL: `https://doi.org/10.1017/S0956796800001830`.