Functional Pearl: Holey Generators!

JOSEPH W. CUTLER, University of Pennsylvania, USA
HARRISON GOLDSTEIN, University of Pennsylvania, USA
KOEN CLAESSEN, Chalmers University of Technology, Sweden
JOHN HUGHES, Chalmers University of Technology, Sweden
BENJAMIN C. PIERCE, University of Pennsylvania, USA

A nice feature of testing frameworks like QuickCheck is that they provide domain-specific languages that can be used to build custom generators for random test data. Programmers value such handcrafted generators for two reasons: they can be used to ensure *invariants*, such as ordering constraints on binary search trees, by construction, and they allow control over the *distributions* of generated values.

How easy is it to control these hand-tuned distributions? Surprisingly hard, as it turns out! We investigate the distributions produced by some familiar generation strategies for unlabeled binary trees and observe that it is quite difficult to achieve both a good distribution of tree sizes and a good distribution of tree shapes. The fundamental issue is *locality of control*—the fact that QuickCheck generators for recursive data structures are naturally written as recursive functions makes it *un*natural to implement the kind of control flow that gives global influence over distributional properties.

We propose instead a novel abstraction, *holey generators*, that makes a sequence of *global* random choices about where to incrementally extend a tree. This abstraction supports much more direct control over distributions; we show how changing a single parameter yields bushy, stringy, left-leaning, right-leaning, and (most challenging) uniformly distributed trees. Moreover, the core (applicative) combinators for building holey generators can be extended with a monadic interface, supporting generators for data structures with invariants, like binary search trees and heaps. We concentrate on the case of data structures shaped like binary trees, with and without invariants; at the end, we discuss prospects for generalizing our results to other data types. Finally, we evaluate our holey generators by showing that they make it painless to achieve testing performance comparable to expert-written classic generators.

1 INTRODUCTION

This book fills a much needed gap.
— Saul Gorn

Property-based testing (PBT) is a popular bug-finding technique, especially in the Haskell community where QuickCheck [2] is the *de facto* testing tool of choice. QuickCheck users define *properties*—Boolean-valued functions that validate a system's behavior on a single given input—and QuickCheck tests that these properties return `True` on many randomly generated arguments. This random generation process can be automated much of the time, but to efficiently generate data structures with invariants, like binary search trees (BSTs), the programmer must write *generators*: functions expressed using QuickCheck's domain-specific language which return randomly-chosen structures that are valid with respect to the invariants. QuickCheck generators for data types like BSTs are commonly written like this:

```
data Tree = Leaf | Node Tree Int Tree

genBST :: (Int, Int) -> Gen Tree
genBST (lo, hi) | lo >= hi = return Leaf
genBST (lo, hi) =
  oneof [return Leaf,
         do
           x <- choose (lo, hi)
```

```
              l <- genBST (lo, x - 1)
              r <- genBST (x + 1, hi)
              return (Node l x r)]
```

This generator produces valid BSTs by generating a value in a range and then recursively generating children whose keys fall in appropriate smaller ranges. It uses QuickCheck's monadic `Gen` abstraction and combinators such as `choose` (which samples from a discrete range) and `oneof` (which invokes a generator chosen at random from a list) to build up larger generators from smaller ones.

This rather naïve generator is not very useful for finding bugs, as many in the community have observed. To see why, let's look at the *distribution* of values that it produces:

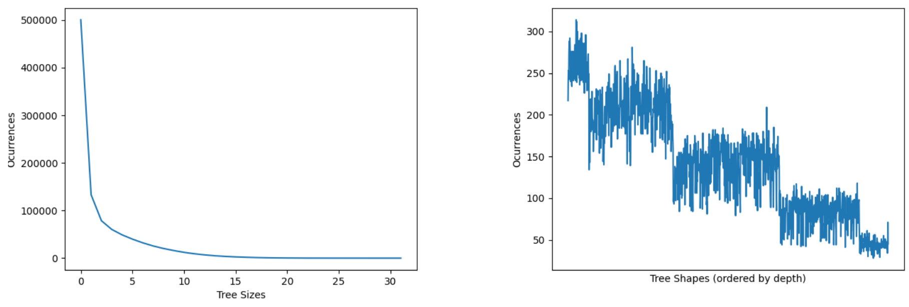

Fig. 1. Left: The size distribution of one million generated BSTs. Right: the shape distribution of BSTs of size 8, ordered shortest to tallest by depth – note that the shortest possible depth is 4. .

Each of these plots points to a serious problem. The first shows that most of the values produced by this generator are far too small: trees of size 1 and 2 are too simple to catch many bugs—and even if they could, there is no point in trying `Leaf` 500,000 times! The second plot, showing the different shapes of size-8 trees that were generated, highlights that some shapes of trees are more popular than others. The generator seems to prefer the short, bushy trees on the left of the graph over the tall, stringy trees on the right.

But wait—doesn't QuickCheck provide tools for addressing exactly these kinds of issues? Well, yes. The "real-world" version of the above generator looks closer to this:

```
genBST :: (Int, Int) -> Gen Tree
genBST bnd = sized (aux bnd)
  where
    aux (lo, hi) n | lo >= hi || n <= 1 = return Leaf
    aux (lo, hi) n =
      frequency [
        (1, return Leaf),
        (5, do
          x <- choose (lo, hi)
          l <- aux (lo, x - 1) (n `div` 2)
          r <- aux (x + 1, hi) (n `div` 2)
          return (Node l x r))
      ]
```

This version of the generator uses two different techniques to get better distributional control. First, it uses QuickCheck's hidden `size` parameter, here named `n`. By cutting off generation when `n` reaches one and halving it as the generator recurses, we ensure that the tree does not get too large. If the size parameter is set to 30 for example, this generator will not produce trees that are more than $\log_2(30) \approx 5$ nodes deep. [1] Additionally, it replaces the `oneof` combinator (which makes uniform choices) with `frequency` (which annotes choices with weights), preferring `Node`s to `Leaf`s at a 5-to-1 ratio.

Does this fix our problems? Sadly, not really. Take a look at the graphs now:

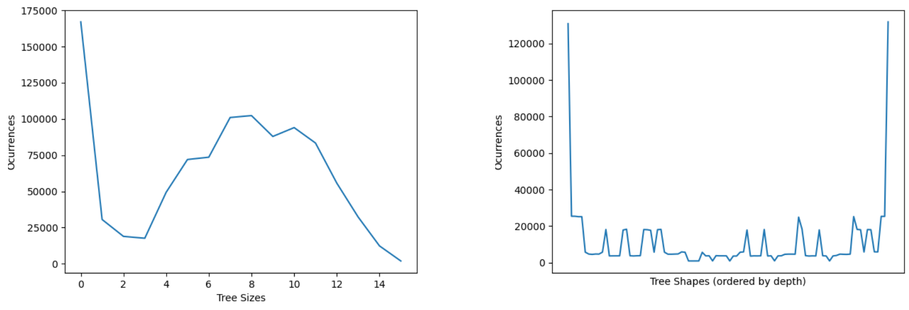

Fig. 2. Left: The size distribution of one million generated trees using the tuned generator. Right: the shape distribution of the tuned generator for trees of size 8, with x axis ordered by depth.

The sizes are a bit more reasonable: using `sized` prevents *very* large trees from ocurring and puts a bit more weight on mid-sized trees between 6 and 12 nodes. But the shape distribution is *much worse* than it was before: every size 8 tree generated has depth exactly 4, leading to only 95 of the possible 1430 shapes ocurring, with two of the shapes accounting for just under a quarter of the total draws! We are playing a game of generator whack-a-mole here: by improving one critical aspect of the generator (size distribution), we have significantly degraded the other (shape distribution).

So why have QuickCheck's distributional control mechanisms been unhelpful in actually controlling tree sizes and shapes? The problem is that control over distributions is traditionally applied *locally*, without regard for broader context. When we recursively build a binary search tree by building left and right subtrees, the recursive calls to the generators that build those subtrees don't know how their results will be used in assembling a larger tree. In BSTs, this lack of communication results in generated trees with a short and bushy structure; long and stringy trees are very unlikely to occur.

What we need is a generic way for different parts of the generation process to depend on one another—decisions made to generate one subtree should be able to influence the decisions made to generate the next ones. In other words, instead of local control, we want *global* control over the distribution. Global control is difficult to achieve because it requires more complex program structure than straightforward recursive functions. We will need a better abstraction than classic QuickCheck generators if we hope to have a usable interface that allows for global control.

But why (you might ask), if we want more control over the distribution of tree shapes, do we not use a system like FEAT [3] instead of classic QuickCheck? This is a reasonable question! FEAT can generate values of any algebraic datatype at any specified size, with a controllable distribution, so in particular it can certainly generate binary trees that are better distributed than what we've

[1]During testing the size parameter ranges from 0 to 99; even at its maximum value, generated trees cannot be more than 6 nodes deep

seen. The problem with FEAT is that it has trouble with generators for data satisfying additional invariants (like binary *search* trees).

To see why, notice that the classic QuickCheck generator above relies on Haskell's `do`-notation to sequence generators that depend on one another:

```
do
  x <- choose (lo, hi)
  l <- aux (lo, x - 1) (n `div` 2)
  r <- aux (x + 1, hi) (n `div` 2)
  return (Node l x r)
```

Here, the calls to `aux` take `x` as an argument, which is only available after the call to `choose`. FEAT does not support this kind of monadic dependency: it only provides an `Applicative` abstraction that runs two component generators independently and combines their results.[2] It appears that FEAT and similar interfaces do not provide enough power for our use cases.

So... what to do? Classic QuickCheck generators provide a powerful monadic interface but give only local control over the distributions of shapes and sizes of the values generated. We do not seem to have a way to build generators that (a) allows users to write in an easy-to-use expressive monadic style, while also (b) providing *global* control over the distributional choices that influence the structures they generate.

The first step in addressing a problem is to understand that you have it. Accordingly, the first contribution of this paper is to bring this problem to the attention of the larger community. The deficiencies of local control appear in lots of abstractions for generating random data. Indeed, many of the complex QuickCheck generators that have appeared in the literature and that are used in practice suffer from the same issues that we've described, since they use the same inherently problematic local distribution control methods like splitting the size parameter for recursive calls. This is clearly not a satisfactory state of affairs, and we hope this paper serves as a call to arms for PBT researchers to study the problem of distributional control more closely.

Solving this problem in general appears quite challenging (we discuss why in Section 7). In the remainder of the paper, we focus on the case of binary trees, with and without invariants, developing an approach to the problem of generation with global control and monadic invariants that solves this case. This *holey generator* technique is a new method for writing generators that combines global control with a monadic interface for invariant maintenance.

The presentation proceeds in two stages. In the first (Section 2), we introduce the basic idea of our technique and demonstrate how to use it to generate unlabeled binary trees using a simple applicative interface; Section 3 then shows how to instantiate this interface to obtain a uniform distribution over tree shapes of a given size. In the second stage (Section 4), we introduce the rest of the holey abstraction and explain how it gives global control over the distributions of labeled tree types with constraints on labels, like BSTs. Section 5 demonstrates that our newfound distributional control makes it painless to write a holey generator that performs as well as a painstakingly-tuned classic one on a slate of tests drawn from *How To Specify It!* [7]. Finally, Section 7 discusses future directions, including ways of generalizing the holey approach beyond binary trees, as well as other methods for distributional control.

[2]Technically, FEAT *can* support a monadic interface, but random generation using it is entirely intractable. The bind (`>>= :: Enumerate a -> (a -> Enumerate b) -> Enumerate b`) would need to eagerly enumerate all of the values of type `a` to generate a single value of type `b`.

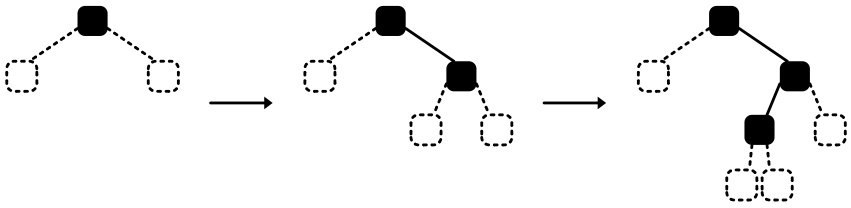

Fig. 3. Generating a tree with hole filling

2 HOLE-FILLING GENERATORS

As we discovered through our experimentation in Section 1, the fundemental problem behind the poor distributions generated by classic QuickCheck generators is that they make random decisions about the shape of their outputs *locally*, without taking into account the entire context that the generated value will be placed into. Our goal is to design an abstraction that empowers users to make random choices that depend on the global state of data that's been generated so far, but which still allows the programmer to write generators in a familiar recursive style. Of course, this is quite difficult! Try as you might, a generator (and indeed, any function) cannot observe the context in which it has been invoked without the programmer explicitly passing around the generator state, which would interrupt the "recursive style" of generator writing.

To solve this conundrum, we subtly shift how we think about generating values. Instead of generating entire structures recursively in one go, we take a step-by-step approach where structures are generated one constructor at a time. Then, after each step, we can "step back", observe the partial structure that has so far been constructed, and make a random choice about how to continue.

We can operationalize this new perspective by defining a recursive generator type `Holey a` as a sort of state machine whose states are binary tree structures of type `a` with "holes at the leaves". Each transition in this state machine represents the "filling" of a hole by replacing the hole with either a new node constructor (and hence two new holes), or a leaf constructor. The state machine presentation of `Holey a` is suggestive of how it can be run: to generate a value of type `a`, we repeatedly transition the system some number of times, and then take the state element as the resulting value. As an example, Figure 3 shows the process of building an unlabeled binary tree through repeated hole-filling.

Concretely, we store this machine state as (1) a value of type `a` representing the tree that has been constructed so far, and (2) a "skeleton" of a binary tree that we call an `HTree`, whose structure mirrors that of the value `a` exactly. The `HTree` has two kinds of leaves, `HoleLeaf`s, which mark the location of a hole, and `DoneLeaf`s, which mark leaves in the current tree which cannot be further extended with a new node. [3]

Formally, a `Holey a` is a record with three fields.

```
data Holey a = Holey
  { done :: a,
    treeOfHoles :: HTree,
    fill :: Hole -> Holey a
  }
```

[3]In a `Holey` gen for *unlabeled* trees, all of the leaves in the `HTree` are `HoleLeaf`s. Because of this, the content of this section works just as well with `Holey a` defined without the `HTree` field, using the only `done` field as the state. While we will not see trees with labels until Section 4, it seems best to introduce the full `Holey` abstraction here.

The first is `done :: a`, the current state of the partially-completed tree of type `a`. The second is an `HTree` whose structure mirrors that of the `done` value, but whose leaves mark either completed sub-trees, or holes which can yet be filled.

```
data Hole = Here | L Hole | R Hole

data HTree = HoleLeaf | DoneLeaf | HNode HTree HTree deriving (Eq,Ord,Show)
```

The final field is the transition function `fill :: Hole -> Holey a`, which, when given the location of a hole in the `HTree` as a path from its root to the leaf, returns a new state where the hole in question has been filled with a new structure, possibly containing more holes. In the case that `fill` is called with a path to a hole that is not actually present in the `HTree`, the generator will fail. In practice, this will never happen, since `fill` is part of the internal API.

Given a `Holey a`, we can repeatedly fill randomly chosen holes from the hole tree to build up a final value of type `a`. Crucially, since we have access to the `HTree` when we pick which hole to fill, the choice of hole can depend on the structure of the hole tree, and by proxy, the entire structure that has been generated so far. Moreover, since hole-filling adds one node at a time, we can control *exactly* how large our structures will be! This holey design has allowed us to untangle the knot that classic QuickCheck generators are constantly tied up in, attempting to balance shape and size control. A `Holey a` generator separately gives control over the shapes of your trees using global hole choices, and control over the sizes by choosing the number of holes to be filled.

2.1 Generators 'n Combinators

To help users effectively build holey generators, we provide an instance for the Haskell `Applicative` typeclass, which defines a way of combining two independent holey generators. A precondition for writing this is that we also provide an instance for the `Functor` typeclass: a function `fmap` which lifts a function between binary tree types `a -> b` to a `Holey a -> Holey b`. Given `f :: a -> b` and `r :: Holey a`, we update `r` by applying `f` to the partial tree `done r`, and post-composing the transition function `fill r` with a recursive call `fmap f` to transform the next states.

```
instance Functor Holey where
  fmap f r = r {fill = fmap f . fill r, done = f (done r)}
```

The applicative interface for `Holey` is much more interesting, as it substantiates much of the discussion that, so far, has been mostly hand-waving. The applicative "combination function" `<*> :: Holey (a -> b) -> Holey a -> Holey b` is where all the action happens.

```
instance Applicative Holey where
  pure x =
    Holey
      { treeOfHoles = DoneLeaf,
        fill = (error "No holes left to fill!"),
        done = x
      }
  rf <*> rx | isDone rf = done rf <$> rx
  rf <*> rx | isDone rx = ($ done rx) <$> rf
  rf <*> rx =
    Holey
      { treeOfHoles = HNode (treeOfHoles rf) (treeOfHoles rx),
        fill = \case
          L h -> fill rf h <*> rx
          R h -> rf <*> fill rx h
          Here -> error "No holes left to fill!",
        done = done rf (done rx)
      }
```

When we combine two holey generators into one with `<*>`, the combined generator's remaining holes are those of the two arguments: the tree of holes of the combined generator has as subtrees the two trees of the argument generators. Since the `treeOfHoles` of the combined generator has two subtrees, a call to `fill` will provide a `Hole` which is either `L h` or `R h`. In the former case, the hole `h` on the left argument `rf` is filled, in the latter case, we fill the hole `h` in `rx`. If either of the two argument generators have no holes, they can be trivially combined with the other using the `fmap` function from the functor instance (written infix with `<$>`).

The applicative interface also requires a function `pure :: a -> Holey a`, which constructs a "trivially holey" generator from a value. Given `x :: a`, it returns the generator which has no holes to fill.

We next define a combinator we call `orFill`, which emulates a common use of the `oneOf` function in QuickCheck: to provide a base case to a recursively-defined generator. For a base case value (usually a leaf) `x` and a holey generator `r`, we define ``x `orFill` r`` to be the generator whose current tree is a leaf node `x`, and a single hole, which when filled, results in the generator `r`. This combinator commonly provides the outermost structure of a generator, providing a choice to either stop the recursion at a leaf, or continue with a recursive call.

```
orFill :: a -> Holey a -> Holey a
orFill x r = Holey {treeOfHoles = HoleLeaf, fill = \Here -> r, done = x}
```

With our holey generator combinators in hand, we can write a simple generator for unabled trees.

```
data UTree = ULeaf | UNode UTree UTree deriving (Eq, Ord, Show)

holeyUTree :: Holey UTree
holeyUTree = ULeaf `orFill` (UNode <$> holeyUTree <*> holeyUTree)
```

2.2 But Why Does it Work?

Back at the beginning of Section 2, we set out our desiderata of allowing for global choices to be made about the structure of the binary trees being generated, while maintaing an API which lets programmers write functions that look like they're directly recursive in the usual style. So how does our `Holey` abstraction accomplish this? To make global choices, we use a state-machine generator type that tracks the shape of the *entire* binary tree structure that has been generated up until that point. But the key insight that allows this all to work is that we can *determine* the state of the binary tree being generated by keeping track of when recursive generators are combined using `<*>`. In a classic generator, the use of a `<*>` to run two recursive generators independently when generating binary trees usually signals the construction of a new node! We leverage this pattern to reconstruct the tree that's already been generated, under the hood. Of course, not *all* uses of `<*>` signal a new node. When binary tree structures include labels or any other nontrivial data, a `<*>` is used for each argument of the constructor. This is precisely the reason that we only add a new `HNode` to the tree of holes when both arguments to the `<*>` include holes, which only holds in the case where both arguments are in fact subtrees.

2.3 Semantics of Holey Generators

We now know how to construct holey generators and why they ought to work in principle, but how do they work in practice? In other words, given a generator `Holey a`, how do we sample values of type `a`? As discussed previously, the process boils down to repeatedly choosing random holes to fill based on the current state of the hole tree. How one makes these random choices determines the distribution over shapes that the generator will denote, and a good distribution makes all the difference in finding bugs quickly. To this end, we define a `HoleWeighting` to be a function `HTree -> [(Int,Hole)]` mapping states of the hole tree to a list of weighted holes in that tree to choose from. Holes with higher weight are chosen with higher probability, and lower weights chosen with lower probability. Formally, when given the weighted list `[(n1,h1),(n2,h2), ... (nk,hk)]` we will sample the next hole to fill from the categorical distribution over the holes `h1 ... kh` with probabilities equal to each weight divided by their sum.

```
type HoleWeighting = HTree -> [(Int,Hole)]
```

With a `HoleWeighting` in hand, we can begin to sample from our random generators. In practice, we will accomplish this by *interpreting* holey generators into standard QuickCheck generators using QuickCheck's `Gen` monad, and then sample from those. The interpretation function `recursively` from `Holey a` into `Gen a` is a straightforward translation of the intuitive semantics of holey generators: iteratively fill holes until done.

```
recursively :: HoleWeighting -> Holey a -> Gen a
recursively f p = sized $ go p 0
  where
    go r _ _ | isDone r = return (done r)
    go r n target | n == target = return (done r)
    go r n target = do
      i <- frequency (second pure <$> f (treeOfHoles r))
      go (fill r i) (n + 1) target
```

The auxiliary function `go :: Holey a -> Int -> Int -> Gen a` defines a single iteration of a loop that will run until either all of the holes have been filled (the first guard), or the structure has reached the desired size (the second). The "body" of the loop passes the tree of holes to the user-specified hole-weighting function, and then samples a hole from it. This hole is then filled, and the loop proceeds.

2.4 Controlling the Distribution

Of course, this begs the question: how does one choose the `HoleWeighting` function? Ideally, one chooses the `HoleWeighting` function to ensure that the important parts of the input space are adequately explored. If, for example, the programmer realizes that some bugs may hide in input trees which are long and stringy, they could assign weights to the holes that are exponentially growing with the depth of each hole. This way, in a partially completed tree, the deeper-down holes in the tree are exponentially more likely to be filled, leading to a "chain reaction", where deep trees beget deeper trees.

```
depthWeighted :: HoleWeighting
depthWeighted t = (\h -> (f h, h)) <$> holes t
  where
    f h = 4 ^ holeDepth h
```

In Figure 4, we present a graph of all of the trees of size 4, and their relative frequencies in a draw of 10,000 trees from a holey generator using `depthWeighted`. From left to right, the trees are ordered by depth.

Conversely, suppose if the programmer believes that their bug will make itself known if fed enough trees which are short and squat? They could consider weighting deeper holes lower, like in `inverseDepthWeighted` below.

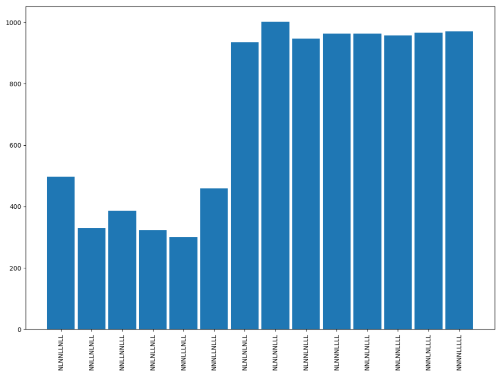

Fig. 4. Frequency of each of the size-4 trees, ordered by increasing depth, in a draw of 10,000 trees from a depth-weighted distribution

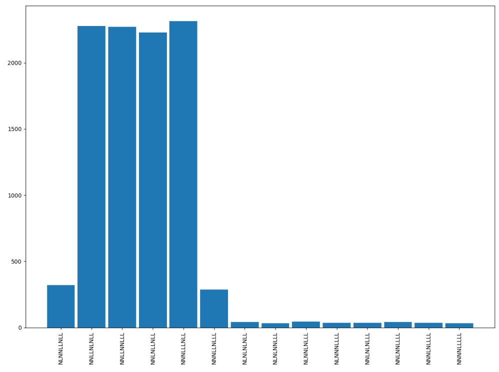

Fig. 5. Frequency of each of the size-4 trees, ordered by increasing depth, in a draw of 10,000 trees from an inverse-depth-weighted distribution

```
inverseDepthWeighted :: HoleWeighting
inverseDepthWeighted t = (\h -> (f h, h)) <$> hs
  where
    hs = holes t
    maxDepth = maximum (holeDepth <$> hs)
    f h = 4 ^ (maxDepth - holeDepth h)
```

This hole weighting gives holes exponentially more weight the closer they are to the root, when compared to the current deepest hole. The graph in Figure 5 shows the opposite story as the graph in Figure 4: shorter, more squat trees are much more likely.

What if the programmer believes that the bug will rear its ugly head on inputs which are severely left or right skewed? In this case, they could heavily weight holes which are left-leaning – the weighting function below operationalizes this by weighting holes exponentially based on how many "left turns" there are on a path from the root down to the hole. The corresponding histogram

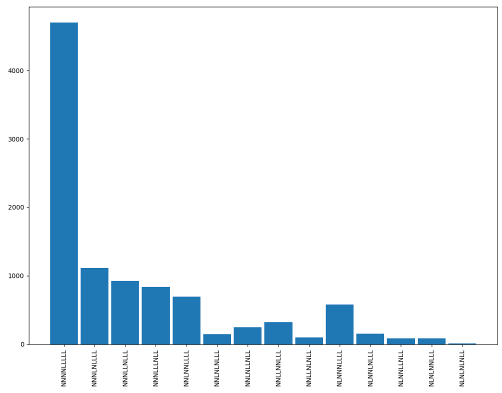

Fig. 6. Frequency of each of the size-4 trees, ordered lexicographically, in a draw of 10,000 trees from a left-weighted distribution

for this weighting is in Figure 6, where the x-axis is ordered using the natural lexicographic order on binary trees.[4]

```
leftWeighted :: HoleWeighting
leftWeighted t = (\h -> (f h, h)) <$> holes t
  where
    f Here = 1
    f (L h) = 4 * f h
    f (R h) = f h
```

All three of these hole weightings induce QuickCheck generators which are very difficult to express using the classic recursive generation methods. Again, this boils down to the lack of local control: all of the hole weightings described above leverage the global view of the `HTree` to choose which holes to fill next.

The ultimate demonstration of the power and control we claim to be able to harness with hole weightings would be to give a hole weighting which induces the *uniform* distribution over the set of shapes of possible values, for every fixed size. It's not at all clear how to accomplish this with classic QuickCheck generators using traditional idiomatic tuning techniques—even for the simplest case of binary trees that we consider in this paper—and so demonstrating how uniformity can be accomplished with holey generators would truly show the flexibility of our technique.

So does the setting of holey generators allow us to encode the uniform distribution? It turns out that the answer is yes! But it will take a bit of explaining to get us there. As a first cut at generating the uniform distribution over trees of every size, let's try choosing to fill each hole in the tree uniformly at random.

```
unweighted :: HoleWeighting
unweighted t = map (1,) (holes t)
```

This gets much closer to uniformity than the tuned classic QuickCheck generator—as shown in Figure 7—but it doesn't *actually* give rise to the uniform distribution of trees of a fixed size. Why not? The crux of the problem is that there are almost always multiple ways to arrive at the same tree

[4] Actually, this is the opposite of the one that Haskell derives for `Ord`.

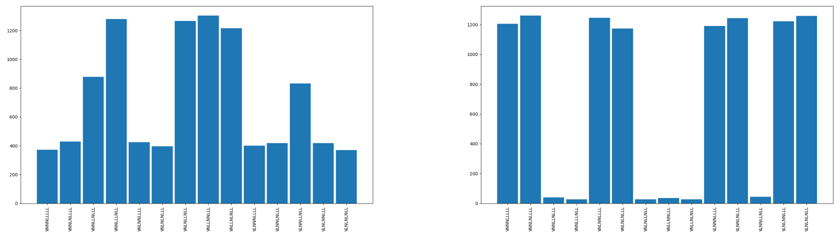

Fig. 7. Frequency Charts, $k = 4$, $n = 10000$. Left: `unweighted` holey generator. Right: classic QuickCheck Generator

through repeated node insertion which leaves some trees more heavily weighted in the distribution than others. But this failure brings important insight: correcting for all of the possible ways that a tree could have been reached is the core of our eventual solution.

3 UNIFORM TREES

Now with a clearer picture of the challenge ahead, let's more precisely define the goal. We need to derive a function `uniform :: HoleWeighting` which, when plugged into the recursive generator for binary trees, yields the uniform distribution on trees of every fixed size. More specifically, `uniform` needs to assign weights to every hole in every possible tree so that filling n holes according to those weights generates a uniformly chosen tree of size n. Formally, we would like that, for every $n \geq 0$, the distribution denoted by `generate (resize` n `(genUTree uniform))` is the uniform distribution on trees of size n. That is, the probability of each tree should be $\frac{1}{C_n}$, where C_n is the n-th Catalan Number: the number of unlabled, ordered, binary trees with n nodes [5].

For the case of binary trees, this problem is tractable. The key insight is to think of choosing the next hole as a *random walk* down the hole tree: a process that starts at the root of the tree and makes independently random choices to "go left" or "go right" until it reaches a leaf. The weight (or probability) of each hole in `uniform h` will be the probability of that random walk ends up at that hole: the product of the probabilities of the choices it made along the way. The challenge of constructing `uniform` then reduces to the challnge of setting the random walk probabilites in such a way that the resulting hole weighting induces the uniform distribution on trees of each size.

Somewhat surprisingly, we can derive an efficiently computable solution for these random walk probabilites, where the probability of taking a left or right turn at a specific node during the walk depends only on the *sizes* of the left and right subtrees rooted at that node.

We also impose the helpful invariant that the generation process will be uniform "at every step." This need not be the case—if you know that you intend to produce a uniformly random tree of size n by repeated node insertion, there is no reason a priori to insist that the terminating this process after k steps yield a uniformly random tree of size k. However, we will adopt this invariant as it greatly reduces the difficulty of computing the probabilities.

Calculating the Weights. To find the right random walk probabilities, let's examine what happens when we add a node to a partially built tree by filling a hole chosen by the random walk. By determining what must be true of the walk probabilities in order for the outcome of this addition to be a uniformly chosen tree, we will derive constraints on the probabilities that can be turned into a method for computing them.

Suppose we have so far generated a tree `t` of size n, and that we've chosen the next hole to fill by taking a random walk down the tree. Let `t'` be the new tree (of size $n + 1$) after filling this hole with

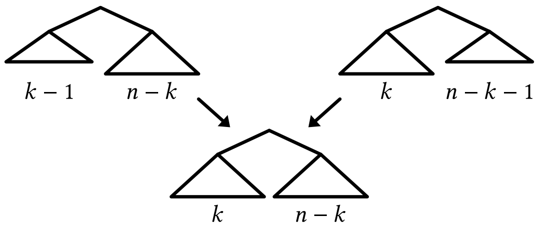

Fig. 8. The ways of getting to a binary tree of size $n+1$ with k nodes on the left, and $n-k$ nodes on the right.

a node. What is the probability that this hole is on the left side of the tree (i.e., that the first step in the random walk was to the left)? Formally, we would like to find $P_n(d \mid l, r)$: the probability that, when our walk encounters a tree of size n with left subtree of size l and right subtree r, it turns in the direction $d \in \{\mathrm{L}, \mathrm{R}\}$. These probabilites are sufficient to define the random walk, and hence the probabilites of filling each hole in every possible tree. To fix some more notation, let $P_n(l, r)$ be the overall probability of generating a tree of size n with left and right subtrees of size l and r, and let $P_n(l, r, d)$ be the probability of generating a tree of size n with a left subtree of size l and a right subtree of size r *and then* taking the first step in the walk down the tree to a hole in direction d.

For the moment, let's suppose that both the left and right subtrees of `t'` are nonempty—we will return to the edge cases later. Let $1 \le k \le n-1$ be the size of the left subtree of `t'` (and $n-k$ the size of the right subtree). Given this setup, there are only two possibilities for the sizes of the two immediate subtrees of `t`: either the hole was chosen on the left, and it has a left subtree of size $k-1$ and right subtree of size $n-k$, or the hole was chosen on the right and it has a left subtree of size k and a right subtree of size $n-k-1$. These two options are shown in diagram form in Figure 8.

This realization about the specific trees `t` and `t'` gives an important insight about the random walk probabilities. Since there are only two possible ways that we could have arried at a tree with a left/right split like `t'`, namely $(k, n-k)$, the probability of generating a tree with that split has to be equal to the sum of the probabilities of (1) generating a tree of size n with left/right split $(k-1, n-k)$ and then the random walk going left, and (2) generating a tree of size n with left/right split $(k, n-k-1)$ and then the random walk going right. Symbolically, we have:

$$P_n(k-1, n-k, \mathrm{L}) + P_n(k, n-k-1, \mathrm{R}) = P_{n+1}(k, n-k) \tag{1}$$

But of course, the choices made during shape generation are all independent, and so the probability of encountering a tree with a particular left/right balance and then going left is the product of the probabilities of seeing such a tree, times the probability of going left at such a tree. So, we can re-write equation (1) as:

$$P_n(k-1, n-k)P_n(\mathrm{L} \mid k-1, n-k) + P_n(k, n-k-1)P_n(\mathrm{R} \mid k, n-k-1) = P_{n+1}(k, n-k) \tag{2}$$

But some of these probabilities are knowable. By the uniform-at-every-step assumption, the probability of encountering an n-node tree with left/right subtrees of size l and r is precisely the fraction of n node trees with subtrees of those size:

$$P_n(l, r) = \frac{C_l C_r}{C_n}$$

Moreover, the probabilities of going left and right are complements, since the walk always goes somewhere with probability 1:

$$P_n(\mathrm{L} \mid l, r) + P_n(\mathrm{R} \mid l, r) = 1$$

Using these facts we can simplify equation (2) to:

$$\frac{C_{k-1}C_{n-k}}{C_n}P_n(\mathrm{L} \mid k-1, n-k) + \frac{C_kC_{n-k-1}}{C_n}\left(1 - P_n(\mathrm{L} \mid k, n-k-1)\right) = \frac{C_kC_{n-k}}{C_{n+1}} \tag{3}$$

To simplify notation somewhat, we define $P_n(k)$ to be $P_n(\mathrm{L} \mid k, n-k-1)$. Rewriting (3), we can plainly see that this equation defines a recurrence relation on $P_n(k)$, for $1 \le k < n-1$

$$\frac{C_{k-1}C_{n-k}}{C_n}P_n(k-1) + \frac{C_kC_{n-k-1}}{C_n}\left(1 - P_n(k)\right) = \frac{C_kC_{n-k}}{C_{n+1}} \tag{$\star$}$$

Given a base case for this recurrence, we can solve it and find the values of $P_n(k)$, as desired. The base case for this recurrence is derived from the two cases we ignored in our original analysis: when the left or right subtree of the tree `t'` that resulted after the node addition were empty, or in other words, in the cases $k = 0$ and $k = n$. By an analysis similar to the one that brought us to the above solution, we find that the following two equations must hold on the boundary:

$$\frac{C_0C_{n-1}}{C_n}\left(1 - P_n(0)\right) = \frac{C_0C_n}{C_{n+1}}$$

$$\frac{C_{n-1}C_0}{C_n}P_n(n-1) = \frac{C_nC_0}{C_{n+1}}$$

Intuitively, these equations hold because there is exactly one way to reach a tree with left/right split $(0, n)$ or $(n, 0)$: by having a tree with split $(0, n-1)$, and $(n-1, 0)$, and going to the right and left, respectively. Solving, this yields the base case:

$$P_n(0) = 1 - \frac{C_n^2}{C_{n-1}C_{n+1}} \tag{$\star\star$}$$

Using ($\star$) and ($\star\star$), we can easily compute $P_n(k)$ for $0 \le k \le n-1$. Through some serious algebraic simplification using the combinatorial fact $\frac{C_n}{C_{n+1}} = \frac{n+2}{2(2n+1)}$, we arrive at the equation:

$$P_n(0) = \frac{3}{(n+1)(2n+1)}$$
$$P_n(k) = \frac{2n-2k-1}{n-k+1}\left(\frac{n+2}{2n+1} - P_n(k-1)\frac{k+1}{2k-1}\right)$$

Correctness. While we hope the derivation above is intuitive, it does not yet constitute a proof. To derive the equations above, we chose a generation scheme—pick holes to fill via a random walk down the graph—and inferred constraints based on what must be true for iterating that process give a uniform distribution over trees of every size. But this by no means proves that using a hole-weighting function that picks holes by a random walk using the probabilites $P_n(k)$ *must* induce the uniform distribution! To prove this, we need to be a bit more formal about our calculations with probabilities.

We begin by defining a random function called add. When given a tree, add takes a random walk down it by making independent left/right choices with our probabilites $P_n(k)$, and inserts a `Node` with `Children` children in place of the leaf at the bottom of its path. Formally, we define

$$\mathsf{add}(\mathsf{Leaf}) = \mathsf{Node\ Leaf\ Leaf}$$

$$\mathsf{add}(\mathsf{Node}\ l\ r) = \begin{cases} \mathsf{Node}\ \mathsf{add}(l)\ r & \text{with probability } P_n(|l|) \\ \mathsf{Node}\ l\ \mathsf{add}(r) & \text{with probability } 1 - P_n(|l|) \end{cases}$$

We then define the formal analogue of our Holey uniform generator as a sequence of random variables T_n defined[5] by iterating the add function.

$$T_0 = \delta_{\mathsf{Leaf}}$$
$$T_{n+1} = \mathrm{add}(T_n)$$

By a routine induction we can see that the add function always sends trees of size n to trees of size $n+1$, and so the trees T_n have support in the set of trees of size n, which we denote Tree_n.

Before prove uniformity, we need a lemma about the way the add function "balances" probabilities between different trees:

Lemma 3.1. *For all $n \geq 0$ and all t of size $n+1$, we have $\sum_{t' \in Tree_n} P(add(t') = t) = \frac{C_n}{C_{n+1}}$*

Proof. See Appendix. □

The lemma must hold in order to show that, in aggregate, the add function behaves $P(\mathrm{add}(t') = t) = \frac{1}{C_{n+1}}$: that every tree t' of size n has an equal chance of getting to any tree of size $n+1$.

Now we can prove that the random variables T_n are in fact uniformly distributed over the trees of size n:

Theorem 3.2. *For all $n \geq 0$ and all t of size n, $\mathbb{P}(T_n{=}t) = \frac{1}{C_n}$*

Proof. The proof proceeds by induction on n. The base case is trivial, since there is only one tree of size 0, namely `Leaf`. For the inductive step, let t be a tree of size $n+1$. Then, unrolling definitions,

$$\mathbb{P}(T_{n+1}{=}t) = \mathbb{P}(\mathrm{add}(T_n){=}t)$$

Because the events $\{T_n = t'\}_{t' \in \mathrm{Tree}_n}$ are disjoint, we have that

$$\mathbb{P}(\mathrm{add}(T_n){=}t) = \sum_{t' \in \mathrm{Tree}_n} \mathbb{P}(T_n{=}t', \mathrm{add}(t'){=}t)$$

Because the randomness in T_n and the add function is independent, we have

$$\sum_{t' \in \mathrm{Tree}_n} \mathbb{P}\,(T_n{=}t', \mathrm{add}(t'){=}t) = \sum_{t' \in \mathrm{Tree}_n} \mathbb{P}(T_n{=}t') \cdot \mathbb{P}(\mathrm{add}(t'){=}t)$$

By the induction hypothesis, $P(T_n{=}t') = \frac{1}{C_n}$, and so:

$$\begin{aligned}\sum_{t' \in \mathrm{Tree}_n} \mathbb{P}(T_n{=}t') \cdot \mathbb{P}(\mathrm{add}(t'){=}t) &= \sum_{t' \in \mathrm{Tree}_n} \frac{1}{C_n} \cdot \mathbb{P}(\mathrm{add}(t'){=}t) \\ &= \frac{1}{C_n} \sum_{t' \in \mathrm{Tree}_n} \mathbb{P}(\mathrm{add}(t'){=}t)\end{aligned}$$

But by Lemma 3.1, $\sum_{t' \in \mathrm{Tree}_n} \mathbb{P}(\mathrm{add}(t'){=}t) = \frac{C_n}{C_{n+1}}$, and so we arrive at

$$\begin{aligned}\frac{1}{C_n} \sum_{t' \in \mathrm{Tree}_n} \mathbb{P}(\mathrm{add}(t'){=}t) &= \frac{1}{C_n} \frac{C_n}{C_{n+1}} \\ &= \frac{1}{C_{n+1}}\end{aligned}$$

Stringing these equalities all together, we have that $\mathbb{P}(T_{n+1}{=}t) = \frac{1}{C_{n+1}}$, as desired. □

[5] δ_{Leaf} is the "delta" random variable with law $P(\delta_{\mathsf{Leaf}} = \mathsf{Leaf}) = 1$

```
leftProbs :: Int -> [Ratio Integer]
leftProbs n = take n $ snd <$> iterate go (1,p0)
  where
    n' = toInteger n
    p0 = 3 % ((n' + 1) * (2 * n' + 1))
    go (k,pk_pred) =
          let k' = toInteger k in
          let a = (2 * n' - 2 * k' - 1) % (n' - k' + 1) in
          let b = (n' + 2) % (2 * n' + 1) in
          let c = (k' + 1) % (2 * k' - 1) in
          let pk = 1 - a * (b - c * pk_pred) in
          (k+1,pk)
```

Fig. 9. Code to compute the random walk probabilities.

Implementation. Now that we know it's correct, there are a few things to remark about our solution to the uniform generation problem for binary trees. The first is that, in principle, these numbers need only be computed once! While there are $O(n^2)$ probablities required to generate a tree of size n, the same random walk probabilities can be used to generate any kind of binary tree of any size less than or equal to n, forever. This is incredibly convenient for using this distribution in practice: the combinatorics only need to be done once, and then can be re-used for any number of generation runs, in any number of tests, for any binary tree data type.

The second thing to remark is that the assumptions that we made in deriving these probabilities — uniform at every step, and that holes are chosen by a random walk from root to leaf — are not only simplifying assumptions that helped us derive the solution, but they also lend themselves nicely to a simple implementation of the `HoleWeighting` corresponding to the uniform distribution. Given a program to compute the probabilities $P_n(k)$ (Figure 9), we use the fact that each step in the random walks are independent to compute the probability of a given hole being filled by multiplying out the probabilities of walking down the path to that particular hole. We can then use these hole probabilities to compute the `HoleWeighting` function for the uniform distribution by simply enumerating the holes in a tree, and computing their probabilities.

```
calcProbs :: HTree -> Hole -> Rational
calcProbs HoleLeaf Here = 1
calcProbs t@(HNode l _) (L h) = (leftProbs (size t) !! (size l)) * (calcProbs l h)
calcProbs t@(HNode l r) (R h) = (1 - ((leftProbs (size t) !! (size l)))) * (calcProbs r h)
calcProbs DoneLeaf _ = error "Impossible: can't walk down to a done."
calcProbs _ _ = undefined

uniform :: HoleWeighting
uniform t = let hs = holes t in zip (weightify $ map (calcProbs t) hs) hs
```

With the `HoleWeighting` in hand, we can an plug it into our `Holey` generator for the `UTree` type, and read off uniformly-at-random trees to our heart's content.

Figure 10 shows the frequencies of all size-4 trees over a draw of 10,000 from the generator `recursively genUTree uniform`. In expectation, one should find $y = \frac{10,000}{C_4} \approx 714$ of each of the $C_4 = 14$ trees. Of course, estimated probabilities will vary between draws, but we can see that all of the empirical frequencies deviate only slighlty from this line, which is drawn in red through the top of the chart.

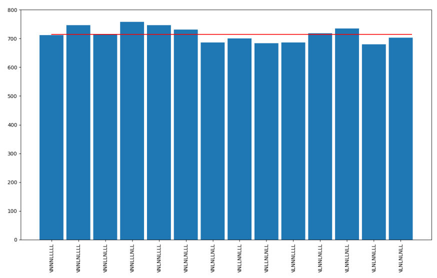

Fig. 10. Frequencies of size-4 trees in a draw of 10,000 from the uniform distribution: compare to Figure 7.

It's worth stepping back to consider what we've demonstrated here. Global control of the choices made during generation has given us complete access to shape the underlying distribution. This mechanism is powerful enough to encode combinatorially complex distributions like the uniform distribution over fixed sizes, simply by weighting which holes to fill.

4 GENERATING HOLEY GENERATORS

There is no doubt now that `Holey` generators give powerful control over distributions, but at this point they can't yet generate interesting structures. So far they have really only been able to produce unlabeled trees. Let's fix that! In this section, we give an elegant approach for *generating holey generators* that reintroduces the monadic power of classic QuickCheck generators without giving up the control we fought so hard to obtain.

4.1 Decorating the Trees

The `holeyUTree` generator from Section 2 is rather bare. It is boringly self-similar, defininig an infinite tree of generators that all generate simple `UNode`s. What we need is a way to *decorate* that tree with data! We can do it manually in few different ways:

```
data Tree a = Leaf | Node (Tree a) a (Tree a) deriving (Eq, Ord, Show)

badGenHoleyTree1 :: Holey (Tree Int)
badGenHoleyTree1 = Leaf `orFill` (Node <$> badGenHoleyTree1 <*> pure 0 <*> badGenHoleyTree1)

badGenHoleyTree2 :: Holey (Tree Int)
badGenHoleyTree2 =
  Leaf `orFill`
    (Node <$> (Node <$> leaf <*> pure 1 <*> leaf)
          <*> pure 3
          <*> (Node <$> leaf <*> pure 5 <*> leaf))
  where
    leaf = pure Leaf
```

The first generator *technically* produces a labelled tree, but all of the labels are `0`, so that's no fun. The second is at least a bit more interesting, but it bottoms out after a few levels because every label needs to be written in manually. But there's a good idea buried here: if we had a way of building holey generators with labels already embedded inside, we could sample shapes from those trees without an issue!

To further illustrate this point, consider this holey generator that produces trees whose labels increase with depth:

```
incHoleyTree :: Int -> Holey (Tree Int)
incHoleyTree k =
  Leaf `orFill` (Node <$> incHoleyTree (k + 1) <*> pure k <*> incHoleyTree (k + 1))
```

Sampling from `incHoleyTree 1 >>= recursively uniform` with size 3 would produce trees like

```
Node Leaf 1 (Node (Node Leaf 3 Leaf) 2 Leaf)     and
Node (Node Leaf 2 Leaf) 1 (Node Leaf 2 Leaf)
```

with a nice uniform distribution over their shapes! The key here is that `incHoleyTree 1` represents an infinite tree of *hypothetical* nodes. These nodes are not part of a tree per se; instead, they are part of the `Holey` structure that will eventually produce a tree (via the hole-filling procedure Section 2). Each node is hypothetical because `recursively` might choose to not expand that hole, leaving only a `Leaf`. But if the procedure *does* expand the node, the label that will appear in the node has already been chosen. This process is illustrated in Figure 11. All we need now is to replace these silly deterministic algorithms with random ones.

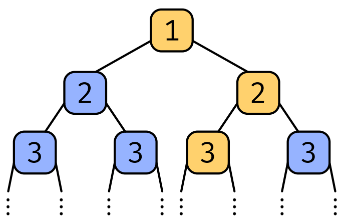

Fig. 11. A single binary tree, chosen from an infinite `Holey` structure.

Of course, we know how to do this—we can just use classic QuickCheck! This program in QuickCheck's `Gen` monad randomly generates a `Holey` generator:

```
genHoleyTree :: Gen (Holey (Tree Int))
genHoleyTree = do
  x <- arbitrary
  l <- genHoleyTree
  r <- genHoleyTree
  return (Leaf `orFill` (Node <$> l <*> pure x <*> r))
```

This function works just like the deterministic algorithm above, but it uses monadic sequencing to thread randomness through the program. This generator works just like `incHoleyTree`, but instead of incrementing the decorated labels down the tree it samples a new one each time using `arbitrary`. Every random seed gives a different holey generator with a different set of labels, and shapes can be sampled from that tree as desired.

Now we can use `genHoleyTree` to get a distribution over labeled trees that is uniform over their shapes, we first run `genHoleyTree` and then we fill holes with `recursively`:

```
genTree :: Gen (Tree Int)
genTree = genHoleyTree >>= recursively uniform
```

As we hoped, this generator produces trees with totally random labels and a nice uniform distribution over shapes:

4.2 What have we gained?

Before continuing, let's reflect on the many benefits of layering `Gen` and `Holey`.

A Cleanly Staged Interface. The type `Gen (Holey a)` is really a staged approach to generation. The first stage of generation controls the labels that might be in the tree, and the second controls the shapes of the actual trees generated. This is quite a natural interface to work with. Since these generators ultimately live in the usual `Gen` monad, they are entirely compatible with existing QuickCheck generators. Furthermore, by separating the random generation of labels from the recursive expansion procedure, it is relatively difficult to "get it wrong". Structure is handled by the `Holey` abstraction, labels are handled by `Gen`, and the two are only interleaved at the last moment. Additionally, staging generation like this can lead to efficiency improvements. If a variety of shapes is deemed more important than a variety of labels, the tester can sample a single `Holey a` and use it to sample many different shapes of trees.

Distributional Control. This was illustrated above—it is easy to see that the distribution of decorated trees is exactly the same as the one that we got from `genHoleyUTree`. We'll need to relax this control a bit when preconditions come into play, but in many cases generating a holey generator gives the same level of control that was provided by the original abstraction discussed in Section 2.

Size Control. The way that generators like this are staged also makes size control a joy. A holey generator like `genHoleyTree` really has two sizes that one might care about. The first is the size of the labels in the tree—the range that they are chosen from—which is controlled by `sized` on the first line of the generator. The second is the size of of the tree itself, which as we know is controlled by `Gen`'s size parameter when `recursively` is called. At first, it would seem that controlling both of these sizes with the same size parameter is a poor choice. Should we make one an explicit argument to the generator?

Actually, there is no need! Since generation of labels happens before generation of trees, we can leverage the holey generator's staging to stage the sizes too. Take a look at a modified version of `genBST`:

```
genBSTResize :: Gen (Tree Int)
genBSTResize = do
  g <- resize 30 genHoleyBST
  resize 5 (recursively uniform g)
```

We call QuickCheck's `resize` function twice to resize the two different aspects of BST size!

4.3 Enforcing Preconditions

Finally, we arrive at generators that enforce preconditions. The ones we present here, for BSTs and for binary heaps, are scarcely more complicated than `genHoleyTree`, but they are quite a bit more interesting and useful.

Our generator for BSTs follows the same strategy that testers use in classic QuickCheck, tracking the minimum and maximum labels allowed in each subtree:

```
genHoleyBST :: Gen (Holey (Tree Int))
genHoleyBST = sized (\n -> aux (-n, n))
  where
    aux (lo, hi) | lo > hi = return (pure Leaf)
    aux (lo, hi) = do
      x <- choose (lo, hi)
      l <- aux (lo, x - 1)
      r <- aux (x + 1, hi)
      return (Leaf `orFill` (Node <$> l <*> pure x <*> r))
```

The labels in this tree are constrained by the bounds that are passed from one recursive call to the next, and the tree is forced to "bottom out" with a `Leaf` if the range for the given subtree is empty. Similarly, this generator for binary heaps ensures ordering from the top to the bottom of the tree:

```
genHoleyHeap :: Gen (Holey (Tree Int))
genHoleyHeap = sized aux
  where
    aux hi | hi <= 0 = return (pure Leaf)
    aux hi = do
      x <- choose (0, hi)
      l <- aux x
      r <- aux x
      return (Leaf `orFill` (Node <$> l <*> pure x <*> r))
```

We can run these holey generator generators just like `genHoleyTree`, either with fancy staged sizing or by simply inheriting QuickCheck's size control:

```
genBST :: Gen (Tree Int)
genBST = genHoleyBST >>= recursively uniform
```

The distribution for BSTs looks like this:

This distribution has nice shape variety, but it isn't quite uniform. This is a side-effect of the BST invariant that the generator is required to enforce. You see, while `genHoleyTree` produces infinite trees of hypothetical labels, `genHoleyBST` gives finite trees. This is obvious from a simple counting argument—BSTs do not allow duplicate elements and the initial range is finite, therefore the hypothetical tree must also be finite—but the mechanics of exactly what happens is interesting. We'll use Figure 12 as an example. If we assume that `lo = 1` (i.e., we are in a subtree where `1` is the smallest valid label), then there are no valid labels to the left of `1` nor are there any to the right of `3`. Accordingly, the branches of the `Holey` generator are cut off at those points, meaning that those parts of the tree will never be expanded to a `Node`.

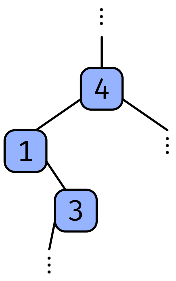

Fig. 12. A `Holey` generator for BSTs, pruned to enforce the invariant (assumes `lo = 1`).

What does any of this have to do with uniformity? Well, in the pathological case where the hypothetical tree has n labels and we want a tree of size n, there is only one tree to choose! So clearly there is no way to get a uniform distribution there. As the number of nodes in the tree increases things get much better, but recall from Section 3 that our uniformity argument still assumes that any node can be expanded to yield a new node with two children (this is how we get to C_n as the number of total trees of size n). If that is not true for some nodes, then the uniformity argument cannot continue to hold in general.

This sounds unfortunate, but it is not the end of the world. If the range of labels allowed in the tree is sufficiently large, then the tree will be as good as infinite from the perspective of hole-filling. After all filling n holes can't possibly explore more than n nodes deep in the hypothetical tree.

Additionally, while properties like uniformity suffer a bit from the addition of precondition constraints, your size almost certainly does not. This is because of the way hole-filling works: `recursively` tries to fill the appropriate number of holes no matter what, even if those holes are in unexpected places. If one part of the tree gets stuck due to a constraint, the rest of the size is allocated to another part of the tree without issue. The only potential problem arises if there are simply not enough nodes available in the hypothetical `Holey` tree, but this is rare. In our experiments, we found that as long as the range of potential labels is at least $\frac{3}{2}$ the total number of desired nodes, the tree will be large enough to avoid worries. Since such a tight bound would lead to distributional issues, there is no reason to cut it that close.

Does any of this actually help with testing? Yes, in fact it does...

5 PUTTING IT INTO PRACTICE

To find out how holey generators perform in a realistic scenario, we experimented with the properties in *How to Specify It!*, a tutorial on property-based testing that uses binary search trees as its running example [7], with all the code available online.[6] In this case, the trees represent finite maps, and so contain both keys and values; the implementation itself is small, but not trivial, consisting of 62 non-blank non-comment lines of Haskell code, with the following API:

```
find   :: Ord k => k ->       BST k v -> Maybe v
nil    ::                              BST k v
insert :: Ord k => k -> v ->  BST k v -> BST k v
delete :: Ord k => k      ->  BST k v -> BST k v
union  :: Ord k => BST k v -> BST k v -> BST k v
```

The code is accompanied by a test-suite of 49 properties from the paper—testing that each operation preserves the invariant, that expected postconditions hold, that various algebraic laws are satisfied, that behaviour is consistent with a 'reference implementation' using lists of key-value pairs, and so on. The respository also includes 31 properties generated by QuickSpec [13], of which 11 duplicate properties presented in the paper. Seven properties in the paper quantify over pairs of equivalent trees (containing the same keys and values), which require a special generator; we discard these and all duplicate properties, resulting in 57 different properties that we can use to evaluate binary tree generators.

The repository also contains eight buggy versions of the implementation, with bugs ranging from blatant (`insert` discards the tree it is passed, always returning a tree with a single node) to more subtle (when taking a `union` of trees with some keys in common, the specification says that values from the left argument should be preferred, but the buggy implementation does not do so consistently). We evaluate generators by measuring the *average number of tests needed to provoke failure*, for every failing property. In each case we took this average over 1,000 failing runs of QuickCheck, which translates into up to 200,000 tests per property (to find 1,000 counterexamples). We ran a very large number of tests of *non-failing* properties, so it is very unlikely that we missed a potential failure.

In all our tests, we generated *keys* in the range 0 to *size*, where *size* is the QuickCheck size parameter. Thus there are *size*+1 possible keys, and when *size* is small, two independently generated keys are quite likely to be equal, while, when *size* is large, this is unlikely.

How to Specify It! originally used the following generator for trees, which uses the `insert` function to insert-a random list of keys and values into the empty tree:

[6] `https://github.com/rjmh/how-to-specify-it`

Method	Total cost	Worst property
Original API-based	1607	193
Classic	1345	135
Holey	1075	103
Tuned API-based	1260	155
Tuned Classic	1006	96

Fig. 13. The holey generator competes with the others in terms of average number of tests needed to provoke all failures, and the failure of the most difficult property. Smaller is better.

```
instance Arbitrary Tree where
  arbitrary = do kvs <- arbitrary :: Gen [(Key,Val)]
                 return $ foldr (uncurry insert) nil kvs
```

We call this an 'API-based' generator, because it uses the API under test to generate test cases; this kind of generator is simple to write, and has the merit that generated trees are guaranteed to satisfy the invariant—provided `insert` is correct. There is a risk that testing may be incomplete, though, if some trees cannot be built at all using `insert` alone.

Using this generator we found 122 failing bug/property combinations; every bug provoked many properties to fail, ranging from 9 to 23 depending on the bug. Every bug can be found with fewer than 10 random tests *using the right property*—but this is a bit like saying that a bug is easy to provoke once you know the right test to run; it's true, but not useful. It is not obvious in advance which property will be most effective at finding each bug.

Not all failures are found fast, though. To give us an idea of how well the generator worked overall, we computed the *total* number of tests needed (on average) to falsify *all* failing bug/property combinations (see Figure 13). We also looked at the *hardest* properties to falsify: the three hardest all involved a buggy `delete`, and they were (in order)

```
prop_DeleteDelete k k' t = delete k (delete k' t) =~= delete k' (delete k t)
prop_DeletePost k t k'   = find k' (delete k t) === if k==k' then Nothing else find k' t
prop_qs_24 k t t'        = delete k (union (delete k t) t') ==== delete k (union t t')
```

The very hardest of these, `propDeleteDelete`, required over 190 random tests to find a counterexample, on average.

Next we replaced the generator with one in the classic style, very like `genBST` on page 2, except that we generate keys in the range $0 \ldots size$, and of course we also generate a random value in each node. As on page 2, we control tree size by halving the size bound in each recursion, and we favour branches over leaves by a ratio of 5-to-1.

The first surprise was that we found four more failures! Recall that the most blatant bug causes `insert` to return a single-node tree, no matter its arguments. When this version of `insert` is used in the API-based generator, it causes every generated tree to consist of only zero or one nodes, so properties that only fail for larger trees cannot be falsified. This is a graphic illustration of the risks of API-based generators.

As well as finding more failures, the classic generator did so at lower total cost (Figure 13); the same three properties were hardest to falsify, but fewer tests were needed to do so.

Finally, we implemented a holey version of the generator. First we wrote an auxiliary

```
holeyTree :: Int -> Int -> Gen (Holey Tree)
```

very like `aux` in the definition of `genHoleyBST` on page 18; `holeyTree lo hi` chooses keys randomly (consistently with the invariant), and returns a `Holey Tree` that can be filled to obtain a tree of any size from 0 to `hi-lo+1` (because there are `hi-lo+1` possible keys). It remains to decide what size of tree to generate.

It might be tempting to generate trees with exactly *size* nodes, but this would be a mistake, because then properties with *two* trees as arguments would only be tested with arguments of the same size. Instead it's customary to treat the size parameter as a *bound*, so that the values generated at *size* + 1 are a superset of those generated at *size*. We achieve this by choosing the tree size uniformly in the range 0 . . . *size*:

```
instance Arbitrary Tree where
  arbitrary = sized $ \n -> do
              ht <- holeyTree 0 n
              treeSize <- choose (0,n+1)
              resize treeSize (recursively uniform ht)
```

This results in a nice variety of test data, including small trees with keys drawn from a small set, small trees with keys drawn from a large set, and even trees containing every possible key.

Repeating our measurements, we found that the holey generator found all the failures at an even lower total cost, and falsified the hardest property (still the same one) in little more than half the number of tests we started with (Figure 13). These improvements are useful, if not dramatic. The second- and third-hardest properties to falsify are actually different in this case: they test the same bug in `delete` as before.[7]

```
prop_qs_27 t k t' = union t (delete k (union t t')) ==== union t (delete k t')
prop_qs_30 k t t' = union (delete k (union t t')) t ==== union t (delete k t')
```

5.1 Tuning our Instruments

One may wonder *why* this bug in `delete` is harder to find, and inspecting a generated counterexample provides a clue:

```
*BSTSpec> quickCheck prop_DeleteDelete
*** Failed! Falsified (after 2 tests):
Key 0
Key 1
Branch (Branch Leaf (Key 0) 0 Leaf) (Key 1) (-1) Leaf
[] /= [(Key 0,0)]
```

Recall that `prop_DeleteDelete` tests that deleting keys in either order yields the same result; in this case the buggy `delete` only works at the root of the tree, so if we try to delete `Key 0` first, then it is not deleted. To falsify the property, we need to choose two different keys, both present in the tree, with one of them located at the root.

What is the probability that a randomly chosen key is present in a generated tree? We can measure this and find that, for the holey generator, it is 50%. This makes sense: we chose the tree size uniformly in the range 0 to the maximum, so on average a generated tree contains half the possible keys. But for the API-based generator, the probability is 36%, and for the classic generator it is only 29%. Thus these generators may perform worse for tests involving `delete`, simply because they test deletion of a key that is not present in the tree more of the time.

With this in mind, we can *tune* the other two generators to generate larger trees, so that a random key is more likely to be present. Since tests in which a key *is* present in a tree, and *is not* present in

[7] These properties are not equivalent since `union` is not commutative.

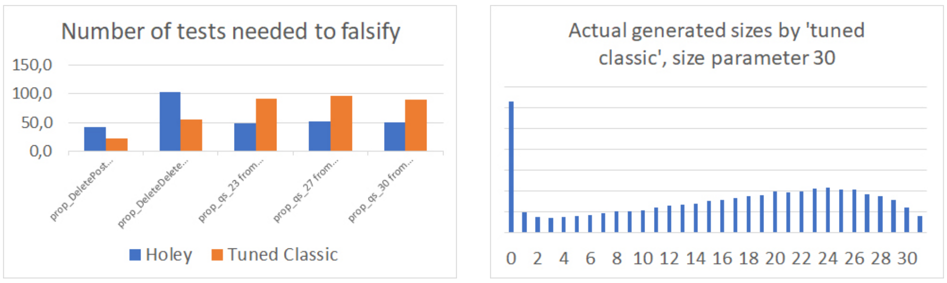

Fig. 14. The "tuned classic" generator: performance vs "holey" on the hardest properties, and distribution of generated tree sizes.

a tree, are arguably equally important, we aim for a 50% presence-probability.[8] For the API-based generator, one might have expected it to generate trees containing half the keys already, since it inserts a random list of keys and values, and QuickCheck generates lists with a length chosen uniformly from $0 \ldots size$. However, since the keys in the list are chosen independently, then some are duplicates. We can tune this generator by increasing the lengths of the lists, and we discovered empirically that lists $\frac{5}{3}$ longer resulted in trees containing half the keys on average.

The classic generator is a bit harder to tune. We can *(a)* reduce the size bound by a smaller factor in each recursion, instead of dividing by 2, and *(b)* weight branch nodes more highly. We discovered empirically that we needed to do both: we removed the size bound altogether, and weighted branch nodes 7× higher than leaves, and by so doing achieved a membership-probability of 48.4%, which was the best we could do.

Tuning did result in better performance (Figure 13), and indeed the tuned classic generator exhibits a slightly lower total cost, and a slightly better worst-property cost, than the holey generator. However, the picture is not clear-cut: if we look at the five hardest properties to falsify (now the same ones for both generators), then we see that while tuning has much improved the classic generator's performance for the two previously-hardest properties, it has also *worsened* it significantly for the three hard generated properties. Moreover, while the *average* tree size is now about right, the *distribution* of tree sizes is still skewed towards either empty or larger trees (Figure 14).

What have we learned? Firstly, that the natural way to write a holey generator resulted in better performance than the natural way to write either an API-based or classic generator. Secondly, while it is *possible* to tune a classic generator to get comparable fault-finding power, it is quite painful to do so because the effect of tweaking each parameter is unpredictable; one is reduced to performing trial-and-error experiments to see if the goal is achieved. This process is fraught, and may be difficult for non-experts. By contrast, it is much easier to specify the desired outcome with holey generators, making them more accessible for average users.

6 RELATED WORK

FEAT and Friends. As discussed in the introduction, FEAT [4] is *almost* a solution to the problem of local distributional control. A FEAT enumerator can generate interesting structures with whatever size and shape distribution the programmer desires, but it cannot do so efficiently under complex

[8]The reader may feel it is less important to test `delete` often with keys that are not present in the tree, but consider `insert`: both inserting a *new* key, and updating the value associated with an *existing* key, are important cases. If you are tempted to suggest using different distributions to test `insert` and `delete`, be aware that some properties call both `insert` and `delete` *with the same key*.

semantic constraints. In the next section, we discuss potential ways to bring FEAT in the fold, combining it with the ideas we present here, but for now it is not quite what we're after.

However, FEAT is not the only tool that gives attractive, global distributional control. Rather than rely on a correct-by-construction generators, [1] use a backtracking scheme that narrows a generator's distribution, avoiding values that fall outside of an executable constraint. This technique is made surprisingly efficient with the help of Haskell's laziness, but fundamentally this approach can still run into trouble when constraints are particularly *sparse* (i.e., when few values satisfy them relative to the total number of values). In those cases, a manual generator, tuned using our techniques, is preferable.

Derived Generators. Distributional properties like size and shape often require manual tuning to get just right, but some properties can be enforced automatically with the help of some pre-computation. The DRAGEN tool uses metaprogramming to automatically derive generators for algebraic data types, which ensure that the constructors in the data type being generated appear at a predetermined rate [11]. For example, a DRAGEN generator for a tree with multiple `Node` constructors can be derived to ensure a particular ratio between $\texttt{Node}_1$ and $\texttt{Node}_2$. A related tool gives similar control over any algebraic data type defined in the "à la carte" tradition [10]. These automatically derived generators are quite impressive, but as with FEAT and its ilk, they cannot express generators that enforce complex preconditions.

Automatic Distributional Control. While we focus on manually tuned generators in this discussion, there is a myriad of automatic approaches that manipulating a generator's distribution. Tools exist that guide test distributions using code coverage [8, 15], a variety of optimization functions [9], common examples [14], and even machine learning[6, 12]. Many of these approaches still ultimately rely on local tuning, which makes certain distributions more difficult to obtain than others, but in specific use-cases these tools are sometimes the best option for low-effort high-reward test generation.

Probabilistic Methods.

7 FUTURE WORK

So what have we learned? In Section 1, we learned that the current state of the art for QuickCheck generators gives inadequate control to the user, even in the well-studied case of generating binary tree data structures. This lack of control manifested itself in a game of generator whack-a-mole, where attempting to improve one aspect of the generator's distribution would cause us to lose control of another. Then, in Section 2, we learned how to achieve this control in the setting of unlabeled binary trees with the help of `Holey` trees. Next, we took a detour though the combinatorics of uniform generation in Section 3 to demonstrate the flexibility our abstracion. And in this Section we learned that a staged approach—generating `Holey` generators—allows us to capture all of the binary tree data types that we could have hoped for.

Finally, this discussion hints at the main technical challenge behind extending this concept to more complex datatypes. Any attempt to generate ternary-or-larger trees using this API will cause a de-coupling of the data structure being generated and the underlying `HTree`. If the data type being generated has an arity three constructor, the recursive generator expression ``g = pure Leaf `orFill` Node `fmap` g <*> g <*> g`` will internally yield unbalanced `HTree`s of the form `HNode (HNode HoleLeaf HoleLeaf) HoleLeaf`, as the `Recursive` API attempts to interpret `<*>`s as binary nodes. Thus, in order to generate tree datatypes other than binary trees, the `HTree` must be changed to reflect the constructor arities of the type in question. In principle, this could be straightforwardly accomplished using Template Haskell metaprogramming or some other more principled means, but we leave this extension to future work.

REFERENCES

[1] Koen Claessen, Jonas Duregård, and Michal H. Palka. 2015. Generating constrained random data with uniform distribution. *J. Funct. Program.* 25 (2015). https://doi.org/10.1017/S0956796815000143

[2] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, Montreal, Canada, 268–279. https://doi.org/10.1145/351240.351266

[3] Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: Functional Enumeration of Algebraic Types. In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark) *(Haskell '12)*. Association for Computing Machinery, New York, NY, USA, 61âĂŞ72. https://doi.org/10.1145/2364506.2364515

[4] Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: Functional Enumeration of Algebraic Types. *SIGPLAN Not.* 47, 12 (sep 2012), 61âĂŞ72. https://doi.org/10.1145/2430532.2364515

[5] A. Cayley Esq. 1859. LVIII. On the analytical forms called trees.âĂŞPart II. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 18, 121 (1859), 374–378. https://doi.org/10.1080/14786445908642782 arXiv:https://doi.org/10.1080/14786445908642782

[6] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 50–59.

[7] John Hughes. 2019. How to Specify It! *20th International Symposium on Trends in Functional Programming* (2019).

[8] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage guided, property based testing. *PACMPL* 3, OOPSLA (2019), 181:1–181:29. https://doi.org/10.1145/3360607

[9] Andreas Löscher and Konstantinos Sagonas. 2017. Targeted Property-Based Testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 46âĂŞ56. https://doi.org/10.1145/3092703.3092711

[10] Agustín Mista and Alejandro Russo. 2019. Deriving compositional random generators. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*. 1–12.

[11] Agustín Mista, Alejandro Russo, and John Hughes. 2018. Branching processes for QuickCheck generators. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, Nicolas Wu (Ed.). ACM, 1–13. https://doi.org/10.1145/3242744.3242747

[12] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2020. Quickly generating diverse valid test inputs with reinforcement learning. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1410–1421. https://doi.org/10.1145/3377811.3380399

[13] Nicholas Smallbone, Moa Johansson, Koen Claessen, and Maximilian Algehed. 2017. Quick specifications for the busy programmer. *Journal of Functional Programming* 27 (2017).

[14] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. 2020. Inputs from Hell Learning Input Distributions for Grammar-Based Test Generation. *IEEE Transactions on Software Engineering* (2020).

[15] Michal Zalewski. 2018. AFL quick start guide. http://lcamtuf.coredump.cx/afl/QuickStartGuide.txt.

APPENDIX

Lemma 3.1. *For all $n \geq 0$ and all t of size $n+1$, we have $\sum_{t' \in Tree_n} P(add(t') = t) = \frac{C_n}{C_{n+1}}$*

Proof. By induction on t. The $t = \text{Leaf}$ case is vacuous. Suppose $n \geq 1$. Let $t = N(l, r)$, and suppose that $|l| = k$ (and so $|r| = n - k$.)

First, consider the case where $1 \leq k \leq n-1$. If l and r both have nonzero size, then there are only two ways to arrive at t by adding a node to a tree t' of size n. Either t' has a left subtree of size $k-1$ (and add took the left branch), or t' has a right subtree of size $n-k-1$ (and add took the right branch). For any tree of size n with any other split of nodes in its left/right subtrees, $P(\text{add}(t') = t) = 0$. Therefore, denoting by $T(a, b)$ the set of $a + b + 1$-node trees with left subtrees of size a and right subtrees of size b, we have:

$$\sum_{t' \in \text{Tree}_n} P(\text{add}(t') = t) = \sum_{t' \in T(k-1, n-k)} P(\text{add}(t') = t) + \sum_{t' \in T(k, n-k-1)} P(\text{add}(t') = t)$$

Re-writing t' inside the sums as $N(l', r')$, and the t as $N(l, r)$, the right hand side of the above is equal to:

$$\sum_{N(l',r') \in T(k-1,n-k)} P(\text{add}(N(l',r')) = N(l,r)) + \sum_{N(l',r') \in T(k,n-k-1)} P(\text{add}(N(l',r')) = N(l,r))$$

Again, most of these terms drop away. On the left side, $P(\text{add}(N(l', r')) = N(l, r)) = 0$ if either $r \neq r'$ or the add goes right: the only way to get to a tree in $T(k, n-k)$ by adding a node to a tree in $T(k-1, n-k)$ is if you fill on the left, and the right subtrees were the same in the first place. A similar argument goes for the right hand side. Thus, we have:

$$\sum_{N(l',r') \in T(k-1,n-k)} P_n(k-1) P(\text{add}(l') = l) P(r = r') + \sum_{N(l',r') \in T(k,n-k-1)} (1 - P_n(k)) P(\text{add}(r') = r) P(l = l')$$

Again, yet more terms drop out: the $P(r = r')$ and $P(l = l')$ are zero for $r \neq r'$ and $l \neq l'$, and so we are left with

$$\sum_{l' \in \text{Tree}_{k-1}} P_n(k-1) P(\text{add}(l') = l) + \sum_{r' \in \text{Tree}_{n-k-1}} (1 - P_n(k)) P(\text{add}(r') = r)$$

Pulling out the constants from both sides, we have

$$P_n(k-1) \sum_{l' \in \text{Tree}_{k-1}} P(\text{add}(l') = l) + (1 - P_n(k)) \sum_{r' \in \text{Tree}_{n-k-1}} P(\text{add}(r') = r)$$

But by the induction hypothesis,

$$\sum_{l' \in \text{Tree}_{k-1}} P(\text{add}(l') = l) = \frac{C_{k-1}}{C_k}$$

and

$$\sum_{r' \in \text{Tree}_{n-k-1}} P(\text{add}(r') = r) = \frac{C_{n-k-1}}{C_{n-k}}.$$

Substituting in, we have

$$P_n(k-1) \sum_{l' \in \text{Tree}_{k-1}} P(\text{add}(l') = l) + (1 - P_n(k)) \sum_{r' \in \text{Tree}_{n-k-1}} P(\text{add}(r') = r)$$

$$= P_n(k-1) \frac{C_{k-1}}{C_k} + (1 - P_n(k)) \frac{C_{n-k-1}}{C_{n-k}}$$

But, we have picked the $P_n(k)$ to satisfy ($\star$), the equation from Section 3! Multiplying ($\star$) through by $\frac{C_n}{C_k C_{n-k}}$ yields

$$P_n(k-1)\frac{C_{k-1}}{C_k} + (1 - P_n(k))\frac{C_{n-k-1}}{C_{n-k}} = \frac{C_n}{C_{n+1}}$$

as required.

Now consider the case where $k = 0$.[9] If t has an empty left subtree and a right subtree of size n, then the only trees t' for which $P(\text{add}(t') = t)$ are trees of the form $N(\texttt{leaf}, r')$, where $|r'| = n - 1$. Thus,

$$\sum_{t' \in \text{Tree}_n} P(\text{add}(t') = t) = \sum_{r' \in \text{Tree}_{n-1}} P(\text{add}(N(\texttt{leaf}, r')) = N(\texttt{leaf}, r))$$

But just like the last case, $\text{add}(N(\texttt{leaf}, r')) = N(\texttt{leaf}, r)$ exactly when add goes right, and $\text{add}(r') = r$. So,

$$\begin{aligned}\sum_{r' \in \text{Tree}_{n-1}} P(\text{add}(N(\texttt{leaf}, r')) = N(\texttt{leaf}, r)) &= \sum_{r' \in \text{Tree}_{n-1}} (1 - P_n(0))P(\text{add}(r') = r) \\ &= (1 - P_n(0)) \sum_{r' \in \text{Tree}_{n-1}} P(\text{add}(r') = r)\end{aligned}$$

but by the IH,

$$(1 - P_n(0)) \sum_{r' \in \text{Tree}_{n-1}} P(\text{add}(r') = r) = (1 - P_n(0))\frac{C_{n-1}}{C_n}.$$

By ($\star\star$), $1 - P_n(0) = \frac{C_n^2}{C_{n-1}C_{n+1}}$, and so

$$\begin{aligned}(1 - P_n(0))\frac{C_{n-1}}{C_n} &= \frac{C_n^2}{C_{n-1}C_{n+1}}\frac{C_{n-1}}{C_n} \\ &= \frac{C_n}{C_{n+1}}\end{aligned}$$

as desired. □

[9] We will omit the $k = n$ case, as it is symmetric.