Functional Pearl: Holey Generators!

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

An attractive 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 *validity conditions* by construction, such as ordering constraints on binary search trees, and to control the *distribution* of generated values.

How good are these hand-tuned distributions? Less good than one might imagine! We illustrate this surprising observation by looking carefully at the distributions produced by some familiar generation strategies for unlabeled binary trees.

- Summarize what we find that is disappointing / surprising
- Introduce the idea of local vs. global control

We propose, instead, a new generator abstraction that makes *global* choices about how to incrementally grow a tree.

- summarize the results

Our story concentrates on the case of binary trees (with and without validity constraints); at the end, we discuss prospects for generalizing to other data types.

Additional Key Words and Phrases: property-based testing, random generation

ACM Reference Format:
Joseph W. Cutler, Harrison Goldstein, John Hughes, Benjamin C. Pierce, and Koen Claessen. 2022. Functional Pearl: Holey Generators!. In *Proceedings of The 27th ACM SIGPLAN International Conference on Functional Programming*. ACM, New York, NY, USA, 21 pages.

1 INTRODUCTION

This book fills a much needed gap.
— Saul Gorn

Property-based testing is a popular bug-finding technique, especially in the Haskell community where QuickCheck [1] 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 always 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 constraints, like binary search trees (BSTs), the programmer must write *generators*: functions expressed using QuickCheck's DSL 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 [
```

ICFP'22, September 11–16, 2022, Ljublijana, Slovenia
2022.

```
    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.

As is well known, this rather naïve generator is not very useful for finding bugs. To see why, let's look at the *distribution* of values that it produces:

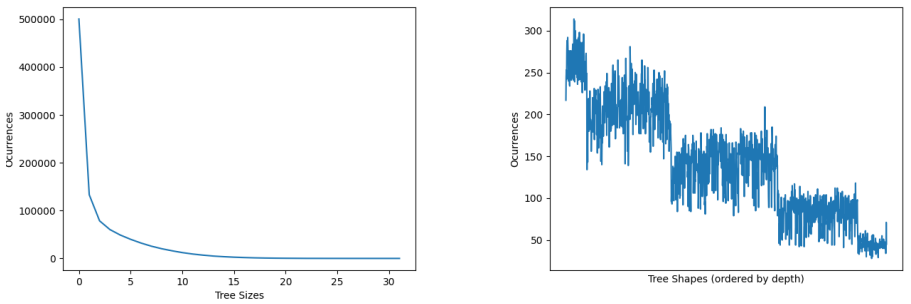

Fig. 1. Left: The size distribution of one million generated BSTs. Right: the shape distribution of BSTs of size 8, ordered shortest to tallest by depth .

Each of these plots points to a serious problem. The first shows that the values produced by this generator are mostly not of very useful sizes. Most of the trees are far too small: trees of size 1 and 2 are too small to catch many bugs—and even if they could, there is no need to try `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 and "bushy" trees found on the left of the graph over the tall and "stringy" trees on the right.

But 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 generator uses two different techniques to get better distributional control. First, it uses QuickCheck's hidden `size` parameter to ensure that values do not get too large. If default settings are used, this generator will not produce trees that are more than 5 nodes deep. 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:

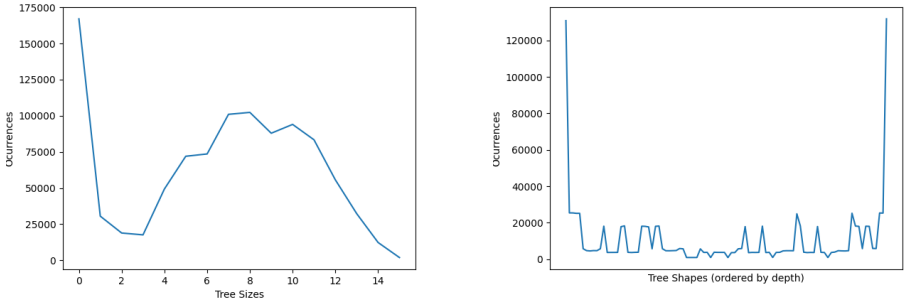

Fig. 2. Left: The size distribution of one million generated trees using the tuned generator. Right: the shape distribution of the tuned generator for trees of size 8, with x axis ordered by depth.

The sizes are a bit more reasonable. The `sized` prevents *very* large trees from ocurring, and puts a bit more weight on mid-sized trees between 6 and 12 nodes. This is a partial win, but the story of the shapes is less encouraging. The shape distribution horendous compared to how it was before: every size 8 tree generated had depth exactly 4, leading to only 95 of the possible 1430 shapes ocurring, with two of the shapes accounting for 240000 of the total draws! This is akin to a game of generator whack-a-mole: by improving one aspect of the generator, we have made the other *significantly* worse.

It might be somewhat surprising that this issue arises in even the simplest generators like the BST generator above. After all, generators with multiple orders of mangitude more complexity have been written by practitioners of property-based testing (PBT). Unfortunately, this problem is fundemental to the behavior of classic QuickCheck generators. In this paper, we forgo an attack on the broader problem and focus primarily on the case of writing generators for structures which are shaped like binary trees. Most of the discussion can straightforwardly be generalized to data structures with higher arity and multiple constructors, but to avoid extraneous details and provide the strongest case for our techniques, we'll stick with binary trees.

So why didn't QuickCheck's distributional control mechanisms fail to influence tree shapes? The problem, which we believe to be fundamental to classic QuickCheck generators, is that the only distributional control is 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" that their results will be put into a larger tree. Each recursive instance of a generator behaves *the same way*, leading to trees whose subtrees are, with all likelihood, very self-similar. In BSTs, this kind of "self-similarity" manifests itself as trees with a short and bushy structure—long and stringy trees are very unlikely to occur.

Instead of local control, we want *global* control over the distribution. Every choice that we make should be able to depend on the outcomes of all previous choices to ensure that the final distribution

can break the self-similarities that arise when every sub-generator runs independently. Concretely, we must be able to make decisions about how to proceed with generation based on choices made to generate other parts of the structure. Global control is difficult to achieve because it requires much 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 abstraction that allow for global control.

Again, we hear you yelling. If you want more control over the distribution of tree shapes, why not use a system like FEAT [2] instead of classic QuickCheck? To this we say, first, please stop yelling. But you do have a point: FEAT can generate values of any algebraic datatype with a controllable distribution, given any desired size, so it can certainly generate binary trees with a custom distribution. However, it has trouble generating binary *search* trees, because these must satisfy an additional invariant. Recall 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)
```

the calls to `aux` take `x` as an argument, which is only available after the call to `choose`. In contrast, FEAT does not allow its component generators to depend on one another, instead providing only an `Applicative` abstraction that runs component generators independently and combines their results. So it seems like existing approaches to generation with controllable distributions won't work out of the box: we need to find another approach.

So what can we do? Classic QuickCheck generators provide a powerful monadic interface, but give only local control over the distributions and sizes of the values generated. How do we build generators that both (a) allow users to write in an easy-to-use expressive monadic style, while (b) providing *global* control over the choices which influence the structures they can generate?

The first step (and this paper's first contribution) is to bring the problem to the attention to the larger community. The deficiencies of local control appear in lots of abstractions for generating random data, including a number of extension of the QuickCheck generator framework . Indeed, many of the complex QuickCheck generators that have appeared in the literature and that are used in practice likely suffer from the same issues that we've described, since they use the same problematic local distribution control methods. 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.

Accordingly, our second contribution is a first step in this direction: we present an approach to the problem of generation with global control and monadic invariants in the case of binary trees. Our technique, *Holey Generators*, 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 stage, Section 2 introduces the basic idea of our technique and demonstrates how to use it to generate unlabled binary trees using a simple applicative interface; Section 3 shows how to instantiate this interface to obtain a uniform distribution over tree shapes of a given size. In the second stage, Section 4 introduces the rest of the holey abstraction and explains how it gives global control over the distributions of labeled tree types with constraints on labels, like BSTs. Section 5 demonstrates the usefulness of the holey technique by testing a suite of QuickCheck properties taken from *How To Specify It!*[3]. Finally, Section 7

Fig. 3. Generating a tree with hole filling

discusses future directions, including ways of generalizing the holey approach beyond binary trees, as well as other methods for distributinal control.

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 to make random choices that depend on the global state of data that's so far been generated, but which still allows the programmer to write generators in a familliar 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—requiring this would destory 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 unabled 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 [1].

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

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

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

[1] In a `Holey` gen for *unlabled* 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 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 the `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` should only ever be called by the function which evaluates the holey generator.

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

Given a `Holey a`, we can repatedly 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. 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 <$>). 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`

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.

```
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)
      }
```

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 desiderada 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. 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.

```
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
```

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 encode their hypotheses about where in the input space a could be found. Perhaps the programmer believes that a potential bug will only make itself known on input trees which are long and stringy, with depth on the order of their size. . In this case, 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.

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

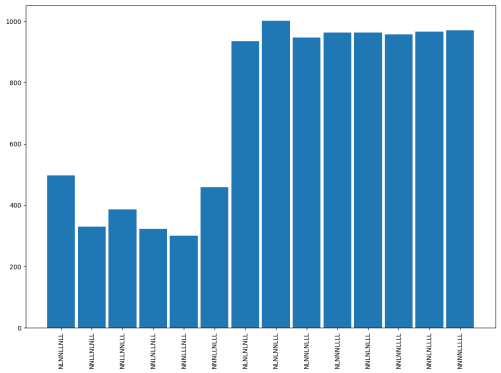

Fig. 4. Frequency of each of the size-4 trees, ordered by increasing depth, in a draw of 10,000 trees from a depth-weighted distribution

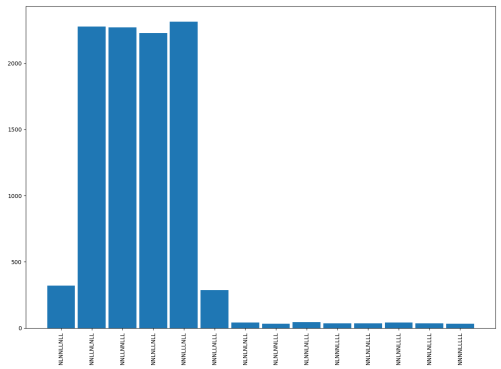

Fig. 5. Frequency of each of the size-4 trees, ordered by increasing depth, in a draw of 10,000 trees from an inverse-depth-weighted distribution

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 for this weighting is in Figure 6, where the x-axis is here ordered using the natural lexicographic order on binary trees[2].

```
leftWeighted :: HoleWeighting
leftWeighted t = (\h -> (f h, h)) <$> holes t
```

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

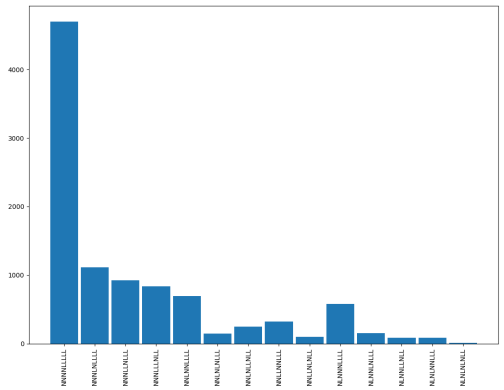

Fig. 6. Frequency of each of the size-4 trees, ordered lexicographically, in a draw of 10,000 trees from a left-weighted distribution

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

```
  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.

But in order to truly demonstrate the power and control we can harness with hole weightings, we will demonstrate a hole weighting which induces the *uniform* distribution over the set of shapes of possible values, for every fixed size. With traditional QuickCheck generators using local weights, this is nigh-on-impossible, even for the very simple case of binary trees that we consider in this paper. 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 distributions, let's try choosing to fill a 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 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, generates 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 holes n 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.

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 lowers 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 a node. What is the probablity 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 $1 \leq k \leq n-1$ 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 the only two possible ways that we could have arried at a tree

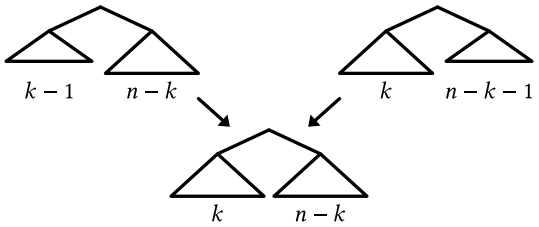

Fig. 8. The ways of getting to a binary tree of size $n+1$ with k nodes on the left, and $n-k$ nodes on the right.

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 know-able. 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:

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

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 $1 \le k < n-1$

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

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:

$$\frac{C_0 C_{n-1}}{C_n} \left(1 - P_n(0)\right) = \frac{C_0 C_n}{C_{n+1}}$$

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

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:

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

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 $\frac{C_n}{C_{n+1}} = \frac{n+2}{2(n+1)}$, 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

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

We then define the formal analogue of our Holey uniform generator as a sequence of random variables T_n defined[3] by iterating the add function.

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

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 we get to the proof of uniformity, a lemma about the add function is required.

Theorem 3.1. *For all $n \ge 0$ 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. □

With our lemma in place, we proceed to proving that the random variables T_n are in fact uniformly distributed over the trees of size n.

Theorem 3.1. *For all $n \ge 0$ and all t of size n, $P(T_n = t) = \frac{1}{C_n}$*

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

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,

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

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

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

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

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

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

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

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

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

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

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."
```

```
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.

Fig. 10. Frequencies of size-10 trees in a draw of 1,000,000

```
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-10 trees over a draw of 1,000,000 unlabled binary trees from the generator `recursively genUTree uniform`. In theory, this graph should be a straight line at $y = \frac{1{,}000{,}000}{C_{10}}$, and this line is drawn in through the center of the graph.

Once again, 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 sufficiently powerful that we can encode combinatorially complicated distributions like the uniform distribution over fixed sizes simply by weighting which holes to fill. Of course, it is clear that our story cannot stop here! Remarkably, this technique for generating uniform binary trees extends to generating nearly-uniform binary trees with invariants.

4 DECORATING HOLEY GENERATORS

So far, our recursive generator abstraction helps programmers generate unlabeled binary trees with flexible distributions. But most tree structures of interest (such as our BST running example) are labeled! Moreover, these trees usually have complex invariants which the generator must enforce, yet our current API provides no mechanism for encoding these constraints. The recursive generators that we have looked at so far only provide random choice in the form of `orFill`, which chooses whether or not to grow a tree at a particular leaf.

The main problem is that `Holey` does not provide any randomness for values in a tree. Consider trying to write a generator for labeled trees of integers:

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

badHoleyTree :: Holey (Tree Int)
badHoleyTree =
```

```
    Leaf `orFill` (Node <$> badHoleyTree <*> error "???" <*> badHoleyTree)
```

There is no way to get a random value to put in the `Node`. Clearly we need some more power.

The solution is surprisingly simple, and is best framed by thinking about *staging*. The plan goes like this: use random generation via normal QuickCheck to pre-determine values for the tree, and then use our `Holey` structure to pick out a shape of tree based on those values. Sound difficult? Actually, it isn't! The following generator does exactly what we want:

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

The function `hnode` here stands for *hypothetical* node. 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 in the previous section). 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 value that will appear in the node has already been chosen! It was pre-determined by `arbitrary` when the `Holey (Tree Int)` was constructed. That is the real power behind this approach: `Gen` sets the values that might appear in the tree, and then the `Holey` structure ensures that the shapes are chosen properly.

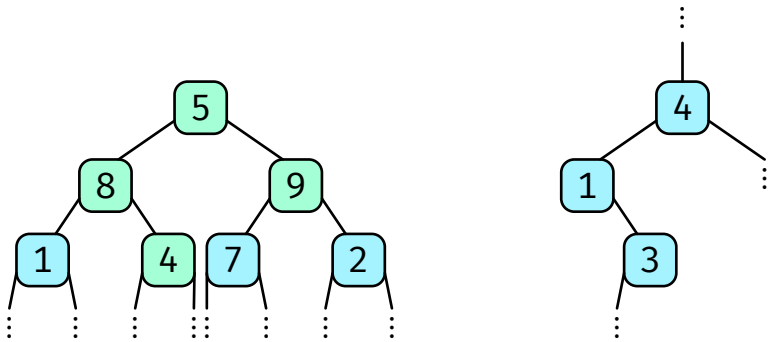

Fig. 11. Left: A single binary tree, chosen from an infinite `Holey` structure. Right: A `Holey` generator for BSTs, pruned to enforce the invariant (assumes `lo = 1`).

So what does a sample from `genHoleyTree` actually look like? Informally, it is an *infinite tree* of arbitrarily chosen integers that might be chosen as part of the tree, as shown on the left of Figure 11. Each node in the tree represents a choice: the recursion could end, resulting in a `Leaf`, or the recursion could continue with a new `Node`. Each `Node`, if chosen, already has a value courtesy of the generator. (And if it is never chosen the value is never computed, thanks to laziness.) In this particular tree, the nodes with values `5`, `8`, `9`, and `4` are all expanded into `Node`s when the holes are eventually filled, the rest of the nodes are not.

To actually 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
```

When actually run:

```
> analyze (QC.generate (QC.resize 10 genTree))
TODO
```

We can see that the distribution is exactly what we hoped for.

This programming scheme is not only useful, it is quite natural. 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 values from the recursive expansion procedure, it is relatively difficult to "get it wrong". Structure is handled by the `Holey` abstraction, values are handled by `Gen`, and the two are only interleaved at the last moment.

Perhaps surprisingly, this same scheme also works when generating values that satisfy a non-trivial semantic invariants as well. The following generator uses the usual technique (that tracks the minimum and maximum values allowed in a subtree) to generate binary search trees:

```
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 <- QC.choose (lo, hi)
      l <- aux (lo, x - 1)
      r <- aux (x + 1, hi)
      return (Leaf `orFill` (Node <$> l <*> pure x <*> r))
```

There are two differences between this generator and the last. First, the values are constrained by the bounds that are passed from one recursive call to the next—this is unsurprising, and does not change anything fundamental. The more interesting change is actually the first line of `aux` that forces a `Leaf` if there are no valid values to use for a node. This check means that, unlike in `genHoleyTree`, the `Holey` tree is not infinite.

There is nothing inherently wrong with the fact that the BST invariant prunes the recursive tree. A `Holey (Tree Int)` generated by `bstGenerator` looks like the right image in Figure 11. We assume that `lo = 1` (i.e., we are in a subtree where `1` is the smallest valid value). There are no valid values to the left of `1` nor are there any to the right of `3`. Accordingly, the branches of the `Recursive` are cut off at those points, meaning that those parts of the tree will never be expanded to a `Node`. When expanding holes, the `Recursive` will have fewer choices to make, but in most circumstances there will still be plenty of choices available.

Of course, this flexibility does come at a price. Since the `Recursive` tree may be cut off at arbitrary points to maintain the BST invariant, the sampled trees are not necessarily selected uniformly with respect to their shapes. Early on, when a subtree has an arbitrarily wide range of valid values, things look sufficiently like the unconstrained case and are nicely uniform. But as the tree gets more constrained, the values might fix a particular shape as the only valid shape.

Still, in practice we find that this generator does quite a good job at producing a wide variety of random values:

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

One more neat trick: the way that generators like this are staged makes size control a joy. A holey generator like `genHoleyBST` really has two sizes that one might care about. The first is the size of the values 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 values 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!

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 manfiested 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.

Does any of this actually help you with testing? In fact, it does.

5 PUTTING IT INTO PRACTICE

6 RELATED WORK

7 FUTURE WORK

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 `HTrees` 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.

NOTES

- Somewhere (in the discussion?) we should remark that a monadic interface gives the user more power, while an applicative interface restricts that power, but in so doing permits a wider range of implementations, thus giving the combinators more power. Indeed, this was one of the original motivations for introducing the applicative interface: "If you need a Monad, that is fine; if you need only an Applicative functor, that is even better!" [7]. For example, a library for building parsers may provide a monadic interface [4], enabling the user to parse context-sensitive grammars in which parsed values may affect the *syntax* of what follows, or a more restrictive applicative interface, which in turn enables the implementation to compute properties such as starter symbols, and thus parse more efficiently [8].

Any monad can be given an applicative interface—`(<*>)` can be defined in terms of `(>>=)`—and Haskell programmers are quite accustomed to using applicative combinators to invoke an underlying monad, without worrying about the difference. *But such applicative interfaces, by definition, cannot make use of the extra power than an applicative interface provides.* In some circumstances it thus makes sense for a library to provide *both* an applicative and a monadic interface, where the applicative interface is *not* just derived from the monadic one in the standard way. For example, Marlow et al.'s HAXL provides a monadic interface for fetching data over a network, together with an applicative interface that allows the implementation to perform independent fetches in parallel [5]. Nowadays Haskell's `do`-syntax can even be desugared into applicative operators, when consecutive operations are independent of each other, to enable this kind of library to be used conveniently [6].

Our library is another useful instance of this idea: we combine a monadic interface to express multi-step generation, with an applicative interface that enables our library to 'see' several recursive calls at once, and take control of their unfolding.

ACKNOWLEDGMENTS

Thanks to Steve for being really good at math

REFERENCES

[1] 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

[2] 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, 61–72. https://doi.org/10.1145/2364506.2364515

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

[4] Graham Hutton and Erik Meijer. 1998. Monadic parsing in Haskell. *Journal of functional programming* 8, 4 (1998), 437–444.

[5] Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is no fork: An abstraction for efficient, concurrent, and concise data access. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. 325–337.

[6] Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. 2016. Desugaring Haskell's Do-Notation into Applicative Operations. In *Proceedings of the 9th International Symposium on Haskell* (Nara, Japan) *(Haskell 2016)*. Association for Computing Machinery, New York, NY, USA, 92–104. https://doi.org/10.1145/2976002.2976007

[7] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of functional programming* 18, 1 (2008), 1–13.

[8] Niklas Röjemo. 1995. Efficient Parsing Combinators.

APPENDIX

THEOREM 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 = 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 $k = 0$[4]. 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. □

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