Ode on a Random Urn (Functional Pearl)

Leonidas Lampropoulos
University of Pennsylvania
llamp@seas.upenn.edu

Antal Spector-Zabusky
University of Pennsylvania
antals@seas.upenn.edu

Kenneth Foner
University of Pennsylvania
kfoner@seas.upenn.edu

Abstract

We present the *urn*, a simple tree-based data structure that supports sampling from and updating discrete probability distributions in logarithmic time. We avoid the usual complexity of traditional self-balancing binary search trees by not keeping values in a specific order. Instead, we keep the tree maximally balanced at all times using a single machine word of overhead: its size.

Urns provide an alternative interface for the `frequency` combinator from the QuickCheck library that allows for asymptotically more efficient sampling from dynamically-updated distributions. They also facilitate backtracking in property-based random testing, and can be applied to such complex examples from the literature as generating well-typed lambda terms or information flow machine states, demonstrating significant speedups.

CCS Concepts • **Theory of computation → Data structures design and analysis**;

Keywords Data Structure, Sampling, Random Testing, QuickCheck, Urn

ACM Reference Format:
Leonidas Lampropoulos, Antal Spector-Zabusky, and Kenneth Foner. 2017. Ode on a Random Urn (Functional Pearl). In *Proceedings of 10th ACM SIGPLAN International Haskell Symposium, Oxford, UK, September 7-8, 2017 (Haskell'17)*, 12 pages.
https://doi.org/10.1145/3122955.3122959

1 Introduction

Back in your introductory math classes, you may have encountered word problems about urns containing balls of different colors – like the urn in Fig. 1 – where you had to calculate the probability of ending up with specific colors after a few draws:

> Suppose you have an urn containing two red balls, four green balls, and three blue balls. If you take three balls out of the urn, what is the probability that two of them are green?

This process, often referred to as sampling without replacement, can be seen as a particular instance of a more general problem: sampling from *updatable discrete distributions*. Sampling from such distributions has many applications, ranging from distribution-tuning in property-based random testing à la QuickCheck [2, 15] to randomized search algorithms that need to try many different options once [9, 13]. But what is an efficient, persistent, purely functional representation of updatable discrete distributions?

To begin with, let's consider how to represent static (non-updatable) distributions. One of the simplest representations is a list of weighted elements. For instance, our example urn can be represented by the list

```
[(2,Red), (4,Green), (3,Blue)]
```

Haskell'17, September 7-8, 2017, Oxford, UK
2017. ACM ISBN 978-1-4503-5182-9/17/09...$15.00
https://doi.org/10.1145/3122955.3122959

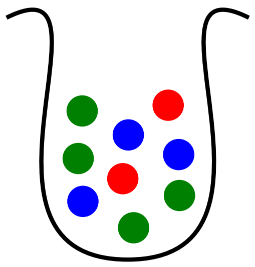

Figure 1. A sample urn.

which tells us that the `Green` ball will be selected with probability $4/(2+4+3)$. Indexing into a list always takes $O(n)$ time, so this sampling procedure will have to be linear. (We will formalize this notion in §2.) While simplistic, this list-based representation has been used successfully for a long time in QuickCheck, whose `frequency` combinator uses this representation. In practice, while the asymptotics of sampling from a list-based representation seem inefficient, the common inputs to `frequency` are small, and so the linear-time traversal of the list is inconsequential.

Unfortunately, this representation is not powerful enough when working with *updatable* distributions: sampling without replacement is only interesting if repeated, which leads to repeated traversals. Moreover, each traversal requires modifying the list to update the distribution. This produces quadratic time (and space) overhead, which can lead to noticeable slowdowns even with relatively small distributions.

In this paper, we present the *urn*, an immutable persistent data structure that supports efficiently sampling from a distribution, as well as efficiently updating it: inserting new (weighted) values, removing them, or updating their weights. Our contributions are:

- We define the urn data structure and provide its Haskell implementation[1] (§3).
- We propose an efficient alternative implementation of the core QuickCheck combinator `frequency` that uses urns instead of lists and empirically evaluate the two versions (§4.1).
- We introduce a novel combinator for prioritized random search, `backtrack`, and explore and evaluate its application in random testing examples from the literature [7, 13] (§4.2).

We discuss related work in §5, and future directions in §6.

2 Sampling Discrete Distributions

At their core, urns represent discrete distributions. We can represent a discrete distribution D over a set A as a nonempty set of pairs of positive weights $w_i \in \mathbb{N}^+$ and of values $x_i \in A$; that is,

$$D = \{\, (w_1, x_1),\ \ldots,\ (w_n, x_n) \,\} \subseteq (\mathbb{N}^+ \times A),\ \ n \geq 1.$$

[1] Full code available on Hackage: https://hackage.haskell.org/package/urn-random

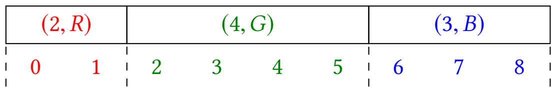

Figure 2. Indexing into the discrete distribution $\{(2, R), (4, G), (3, B)\}$; natural number indices are placed below their corresponding weight-value pair (the order is arbitrary).

Let $W = \sum_{i=1}^{n} w_i$ be the sum of all the weights; then, to sample a value from D is to pick a random x_k with probability w_k/W.

To sample from such a distribution, we use the range $[0, W)$ as indices into it. If we pick a natural number uniformly at random from $[0, W)$, we can map it to the x_i: the first w_1 natural numbers correspond to x_1, the next w_2 natural numbers correspond to x_2, and so on. Intuitively, we are breaking the range $[0, W)$ into n buckets: $[0, w_1)$, $[w_1, w_1 + w_2)$, and so on up through $[w_1 + \cdots + w_{n-1},\ w_1 + \cdots + w_{n-1} + w_n = W)$. Then the kth bucket, which corresponds to x_k, is

$$\left[\sum_{i=1}^{k-1} w_i,\ \sum_{i=1}^{k} w_i\right)$$

and has size

$$\sum_{i=1}^{k} w_i - \sum_{i=1}^{k-1} w_i = w_k.$$

Thus, there are w_k different values for each index that result in picking this bucket. Since each index is equally likely, the total probability of picking x_k is w_k/W. An instantiation of the buckets for the distribution $\{(2, R), (4, G), (3, B)\}$ – which will be a running example for the remainder of the paper – appears in Fig. 2.

This approach is the basis of the urn sampling algorithm, as well as the standard `frequency` combinator in QuickCheck and the array-based binary search variant used in `random-fu` [16] (for more about the latter two, see §5).

3 The Urn Data Structure

Since urns represent discrete distributions, their interface must provide support for (a) *constructing* urns from a list of pairs of weights and values (i.e., the list-based representation described in §1), and (b) *sampling* from urns. Additionally, we want to support (c) *modifying* urns in three different ways: (i) sampling without replacement; (ii) inserting new (weighted) values; and (iii) updating the weight and value of a sampled item.

3.1 The Urn API

The full API for urns is presented in Fig. 3; the interface is split into five categories.

Types: These include `Urn`, the type of discrete distributions; `Weight`s in those distributions; and `MonadSample`, a type class for monads that support random number generation to enable sampling from urns, such as `IO` or QuickCheck's `Gen` type for random generators.[2]

Construction: The `singleton` and `fromList` functions create distributions where the given values (of type `a`) have the corresponding weights. The `fromList` function produces a `Maybe (Urn a)` because distributions cannot be empty.

[2]While the `MonadRandom` type class [4] would seem to be a good fit for this purpose, `Gen` is unfortunately not an instance of it.

Sampling: The `sample` and `remove` functions both pick a value from the `Urn` at random, with probability proportional to its weight. The `remove` function also takes that value out of the urn ("sampling without replacement"), and so returns the `Weight` of that value and `Maybe` the updated `Urn`. Note that `remove` takes out the whole weight-value pair, rather than taking out one copy of the value and reducing its weight by 1.

Modification: The `insert` function simply adds a new value to the distribution with a given weight. The `update` and `replace` functions both choose a random value to modify, as per `sample`. The `update` function will modify that value and its weight, returning the old weighted value, the new weighted value, and the new `Urn`; the `replace` function simply overwrites the chosen value and its weight, returning the old weight and value along with the new `Urn`.

Properties: `Urn`s keep track of how many values they contain (`size`) and their total weight (`weight`).

Coding Conventions As we saw in §2, sampling from a distribution D with total weight W can be done by sampling a natural number uniformly from $[0, W)$ and using it as an index into D. The `MonadSample` type class provides the index-generation functionality; its only method is

```
randomWord :: MonadSample m => (Word,Word) -> m Word
```

where `randomWord (low,high)` generates a random `Word` chosen uniformly from the range `[low,high]`.

In the remainder of this section, we implement all the functions from Fig. 3 that require randomness by phrasing them instead in terms of indices into the urn. Every such function now requires an additional argument of type `Index`, where `Index` is a type synonym for `Word`. When formulated this way, these functions are deterministic. Thus, although the user-facing version of `sample` has type `MonadSample m => Urn a -> m a`, the implementation that we show in this section has type `Urn a -> Index -> a`, and similarly for `remove`, `update`, and `replace`. We connect the randomized and deterministic versions of these functions by generating random indices with `randomWord (0, w-1)`, where `w` is the total weight of the input `Urn`.

3.2 A Weighty Matter

The `Urn` abstract data type, behind the scenes, is implemented as a balanced binary tree – the functional programmer's go-to choice for logarithmic-time operations. However, before we consider how the trees are balanced, we need to consider how they represent discrete distributions in the first place; we save balancing concerns for §3.4.

As distributions must be nonempty, we use a nonempty binary tree that stores data – values in the distribution – at the leaves. In addition, we must also store information about the weights of each value: each location in the tree, leaf and (internal) node alike, stores a weight. We maintain the invariant that the weight of a tree or subtree is the total weight of every value in the corresponding distribution. This means that the weight of a leaf is simply the weight of the value it holds, and the weight of a node is the sum of the weights of its children. Such a tree can be represented by the following data type:

```
data Urn a                       -- a discrete distribution; abstract
type Weight = Word               -- nonzero
class Monad m => MonadSample m   -- provides randomness

singleton :: Weight -> a -> Urn a
fromList  :: [(Weight,a)] -> Maybe (Urn a)

sample    :: MonadSample m => Urn a -> m a
remove    :: MonadSample m => Urn a -> m ((Weight,a), Maybe (Urn a))

insert    :: Weight -> a -> Urn a -> Urn a
update    :: MonadSample m => (Weight -> a -> (Weight,a)) -> Urn a -> m ((Weight,a), (Weight,a), Urn a)
replace   :: MonadSample m => Weight -> a -> Urn a -> m ((Weight,a), Urn a)

size      :: Urn a -> Word
weight    :: Urn a -> Weight
```

Figure 3. The API for urns: the types, constructors, sampling functions, and updating functions.

```
-- Simple tree representation
Node 9 (Node 6 (Leaf 2 R)
               (Leaf 4 G))
       (Leaf 3 B)
```

```
-- An alternate grouping
Node 9 (Leaf 2 R)
       (Node 7 (Leaf 4 G)
               (Leaf 3 B))
```

```
-- Order doesn't matter
Node 9 (Node 5 (Leaf 2 R)
               (Leaf 3 B))
       (Leaf 4 G)
```

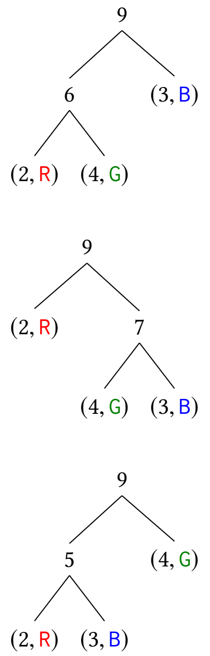

Figure 4. Three different tree representations of the distribution $\{(2, R), (4, G), (3, B)\}$.

```
type Weight = Word

data Tree a = Node Weight (Tree a) (Tree a)
            | Leaf Weight a

weight :: Tree a -> Weight
```

Our example distribution, $\{(2,R),(4,G),(3,B)\}$, can be represented as a Tree in multiple different ways by altering the grouping or the ordering of values. Three possible tree representations of this distribution are shown in Fig. 4.

The rationale behind storing the aggregate weights at the internal nodes comes from thinking about the buckets from §2. We can think of each Node w l r as representing a single "super-bucket" of size w, where the "super-bucket" spans the buckets of every value

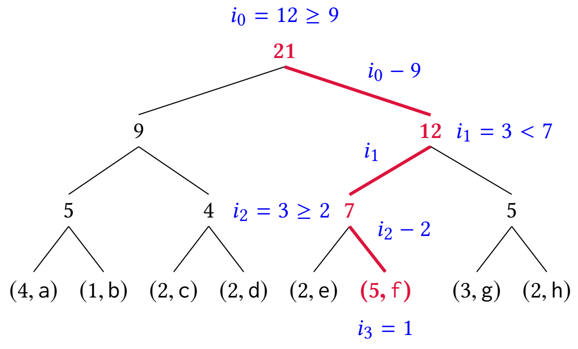

Figure 5. What happens when sampling from an urn. This example looks up the index 12 in a tree representing the distribution $\{(4,a),(1,b),(2,c),(2,d),(2,e),(5,f),(3,g),(2,h)\}$. The path taken through the Tree is in **bold red**; the changes to the index i_x at the xth recursive call are in blue. As this shows, adjusting is only done when recursing into the right-hand child of a node.

at the leaves. If the total range covered by this tree is $[b, b+\mathtt{w})$, then its two subtrees l and r split it into $[b, b+\mathtt{wl})$ and $[b+\mathtt{wl}, b+\mathtt{wl}+\mathtt{wr})$, where wl = weight l, wr = weight r, and wl + wr = w by the invariant on weights. An index i into this range falls in the left super-bucket if i < b + wl, and the right super-bucket otherwise; applying this recursively, we end up in the correct bucket, which is to say at the correct leaf. This algorithm can be slightly simplified by always adjusting the super-buckets to start at 0. This allows every Tree to be considered in isolation, without any need to keep track of the super-bucket's base b; to do so, we simply adjust i if it would fall in the right-hand bucket, subtracting wl. This leads to the following Haskell implementation:

```
sample :: Tree a -> Index -> a
sample (Leaf _ a)    _ = a
sample (Node w l r) i
  | i < wl     = sample l i
  | otherwise = sample r (i - wl)
  where wl = weight l
```

The result of running this algorithm on an 8-leaf Tree is presented in Fig. 5.

3.3 Turning Over a New Leaf

The `update` and `replace` functions from §3.1 are similar to `sample`, but they return a modified `Urn` in addition to the randomly chosen value. We consider `update` first: given a function `upd :: Weight -> a -> (Weight,a)` and an urn `u`, the call `update upd u` randomly chooses a value `a` in the urn with some weight `w`, applies `upd w a` to get the result `(w',a')`, and returns a triple `((w,a), (w',a'), wt')` consisting of the old weighted value, the new weighted value, and the new `Urn`, which has had `w` and `a` replaced by `w'` and `a'`. (We return both `(w,a)` and `(w',a')` in case we need the new values, as this avoids recomputing them when `upd` is expensive.)

The way that `update` uses an index to traverse a tree is the same as `sample`. However, as `update` modifies the `Tree`, we need to update the weights as we rebuild the tree on the way back up: every weight above the updated leaf must be adjusted by the difference `w'-w`. We do not need to worry about rebalancing, since the *structure* of the `Tree` and the number of values it contains is unchanged.

```
update :: (Weight -> a -> (Weight,a)) -> Tree a
       -> Index -> ((Weight,a), (Weight,a), Tree a)
update upd (Leaf w a)   i =
  let (w',a') = upd w a
  in ( (w,a), (w',a')
     , Leaf w' a' )
update upd (Node w l r) i =
  | i < wl =
    let (old, new, l') = update upd l i
    in ( old, new
       , Node (w - fst old + fst new) l' r )
  | otherwise =
    let (old, new, r') = update upd r (i-wl)
    in ( old, new
       , Node (w - fst old + fst new) l r' )
  where wl = weight l
```

The function `replace w' a'` is essentially the same as `update (\_ _ -> (w',a'))`, so we elide its implementation; the difference is that since `w'` and `a'` are statically known, we only need to return a pair `((w,a), wt')` containing the old weighted value and the new urn.

3.4 A Balancing Act

As mentioned at the beginning of §3.2, if we want logarithmic runtime for all our operations, we need to make sure our trees stay balanced when we add or remove values from the distribution. However, the `Tree` type presented thus far does not contain enough information to stay balanced if we change the size or layout dynamically. Because there is no natural ordering to the values contained within an urn, using a self-balancing binary search tree such as an AVL or red-black tree is unnecessarily complex and a poor fit for the problem we wish to solve. Such an implementation would force us to impose an ordering on the values contained in the urn, and some values we frequently wish to store in an urn – such as QuickCheck generators, which are wrappers around functions – cannot be given an ordering at all.

The key insight to balancing `Trees` in the simplest way is to realize that, unlike for binary search trees, *order is truly irrelevant*; we first encountered this in Fig. 4. The efficiency of `sample` also does not depend on ordering, as the only invariant we have imposed on our `Trees` is that `weight (Node w l r) == weight l + weight r`. Thus, if we always insert values at the second-deepest level of the tree until we must start a new level, we will maintain the balance.

When we wish to insert a new value into the tree, we take some path to get there, which involves going left or right at each `Node`. If we always go in the *opposite direction* on each successive insertion, we will distribute our updates evenly throughout the tree. We can do this by storing a *direction* at each `Node`: either left (⟵) or right (⟶). To decide where to insert, we recurse into the child we are directed to, and toggle the direction.

This allows us to implement the self-balancing insertion function `insert :: Weight -> a -> Tree a -> Tree a`, where `insert w a` inserts the value `a` into a distribution with weight `w`. As we go down the tree, we add the to-be-inserted weight `w` to the weight at every node we pass; to decide which way to go, all we need to do is follow the directions.

It is easiest to see what this means by looking at how this approach iteratively builds up a new tree from a singleton distribution, one insertion at a time; we present an example of this in Fig. 6. Each successive tree has a new leaf at the location found by following the arrows down from the root in the previous tree (on the right of the old leaf), and all the arrows that were followed in that traversal are flipped from said previous tree. We can see that this "evenly distributes" new leaves, rather than filling them in from left to right.

3.5 Losing Direction

We can look at the insertion pattern shown in Fig. 6 and record the directions we take, using `L` for left and `R` for right; every such path ends with an `R`, as new leaves are added to the right of old ones. The sequence of insertions we get is shown in the following table:

Inserting	Path	Inserting	Path
(2, b)	R	(5, e)	LLR
(3, c)	LR	(6, f)	RLR
(4, d)	RR	(7, g)	LRR
		(8, h)	RRR

The pattern of `L`s and `R`s is a familiar one: if `L` is 0 and `R` is 1, then we can read any given path backwards as a binary number. Enumerating our paths in this way counts from $1_{10} = 1_2$ through $7_{10} = 111_2$. This means that the path we must take to find a new insertion location is given exactly by the binary representation of the *size* of the `Tree` before insertion!

Thus, all we need beyond our original `Tree` type is a single `Word` keeping track of the size, and we have all the balancing information we need. It is this composite data type consisting of a size and a `Tree` that we call an `Urn`:

```
data Urn a = Urn { size :: Word
                 , tree :: Tree a }
```

All the functions that we defined on `Trees` are lifted to `Urns` by operating on the `tree` field.

This also saves space! A `Tree` holds the minimum amount of information that we need to sample from a discrete distribution; if we had to include directions in a `Tree` with n values, we would incur $O(n)$ overhead to store them. With an `Urn`, our space overhead, with respect to an ordinary tree representation, is reduced to a single machine word.

We have to change the insertion algorithm to use the size of the urn to perform traversal instead of embedded directions. At every

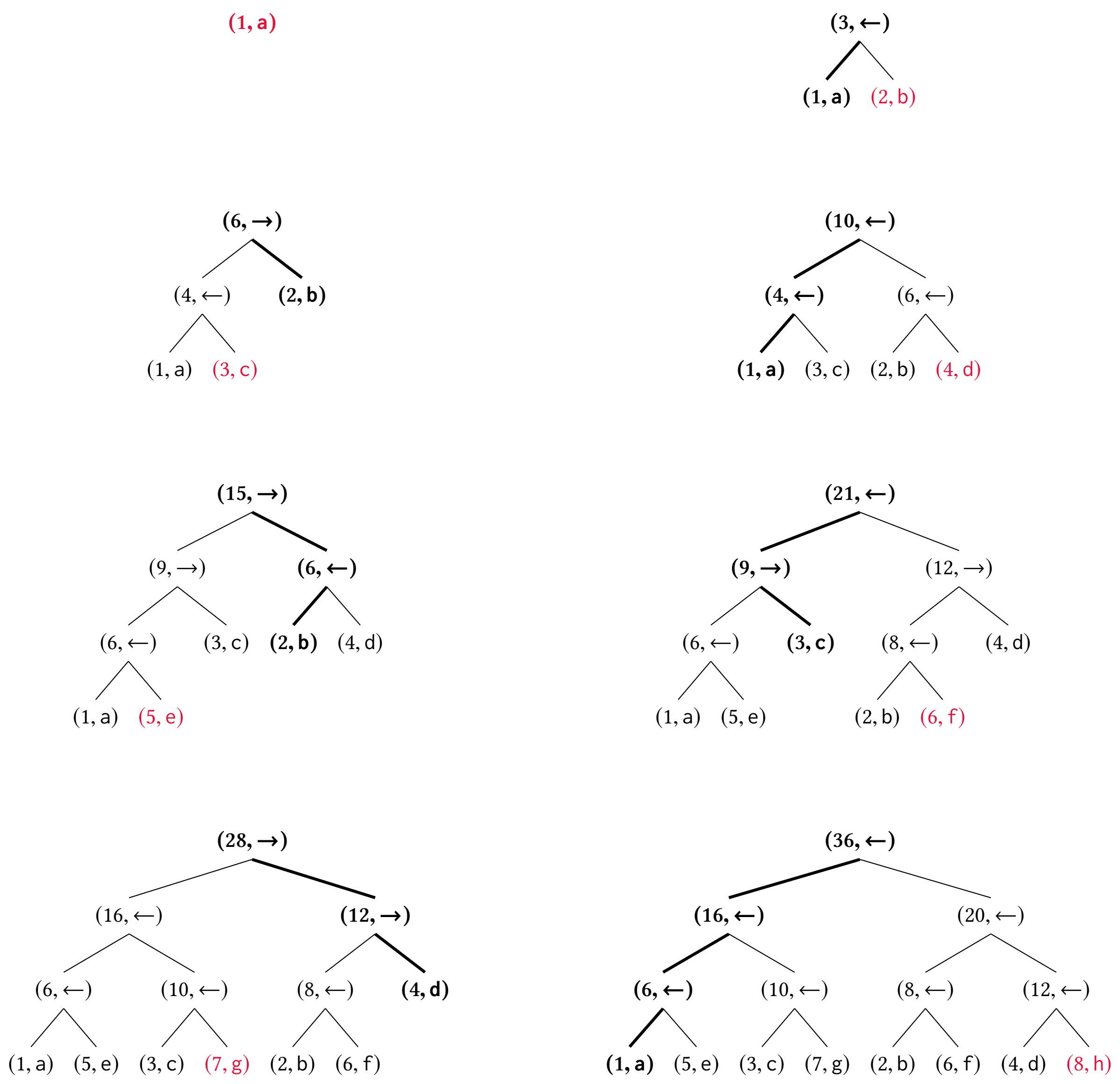
(1, a)
(3, ←)
(1, a) (2, b)
(6, →)
(4, ←) (2, b)
(1, a) (3, c)
(10, ←)
(4, ←) (6, ←)
(1, a) (3, c) (2, b) (4, d)
(15, →)
(9, →) (6, ←)
(6, ←) (3, c) (2, b) (4, d)
(1, a) (5, e)
(21, ←)
(9, →) (12, →)
(6, ←) (3, c) (8, ←) (4, d)
(1, a) (5, e) (2, b) (6, f)
(28, →)
(16, ←) (12, →)
(6, ←) (10, ←) (8, ←) (4, d)
(1, a) (5, e) (3, c) (7, g) (2, b) (6, f)
(36, ←)
(16, ←) (20, ←)
(6, ←) (10, ←) (8, ←) (12, ←)
(1, a) (5, e) (3, c) (7, g) (2, b) (6, f) (4, d) (8, h)

Figure 6. Iteratively constructing a directed tree, building up the distribution {(1, a), (2, b), (3, c), (4, d), (5, e), (6, f), (7, g), (8, h)} one value at a time. The **bold** paths indicate where the next value will be inserted; the red leaves are the most-recently-inserted leaf in each tree.

step, the low bit of the given size is the direction to travel – if the bit is 1, we go right, and if it is 0, we go left. In the recursive call, we shift off the lowest bit and recurse, getting access to the next lowest bit, which is the next direction in the path.[3]

```
insert :: Weight -> a -> Urn a -> Urn a
insert w' a' (Urn size tree) =
  Urn (size+1) $ go size tree
  where go _    (Leaf w a) =
          Node (w+w') (Leaf w a) (Leaf w' a')
        go path (WNode w l r)
          | path `testBit` 0 =
            WNode (w+w') l (go path' r)
          | otherwise =
            WNode (w+w') (go path' l) r
          where path' = path `shiftR` 1
```

As we see in `insert`, binary numbers correspond to root-to-leaf paths in an urn read backwards. In fact, for an urn of size n, all binary numbers less than n correspond to valid paths; vice versa, all paths to leaves correspond to binary numbers less than n. Moreover, just like `insert` uses n as the path to the insertion point, we will always be able to use this same n as the path to that location.

3.6 A Value Un-urned

Now that we have the definition of `Urn`s, the final piece of functionality we need to implement is deletion. In order to maintain balance in our trees, we cannot remove values from arbitrary locations. There is precisely one node whose removal would leave the tree compatible with further iterated insertion: the most recently

[3]Computing `n `testBit` b` determines whether the bth bit of `n` is set.

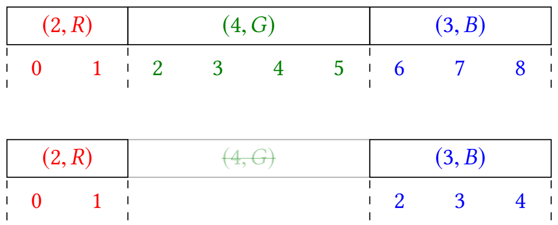

Figure 7. What happens to indices when uninserting a bucket: if `uninsert` returns `((4,G), 2, Just ...)`, then the indices into the subsequent *B* bucket must be shifted down.

inserted value. Removing this value would take us back to the previous, also-balanced tree.[4] We call this operation `uninsert`, and we can combine it with `replace` to implement `remove`: we `uninsert` the most-recently-inserted weighted value, and then `replace` the weighted value we want to `remove` with said `uninserted` weighted value.

However, as they say, the devil is in the details. The first thing we need to do is call `uninsert` to produce the value we need to pass to `replace`, as well as a new urn. Since urns cannot be empty, `uninsert` actually returns a `Maybe (Urn a)` – `uninserting` from an `Urn` of size one produces `Nothing`. Moreover, if the result *is* `Nothing`, we are done: there was only one possible value we could have removed, so we must have removed it.

On the other hand, if the result is a `Just`, we encounter a problem: `remove` had as an argument an index `i` into the urn that we had *before* calling `uninsert`, which means it cannot be used to index into its result – some of the indices may have shifted during uninsertion. We can see a visualization of what happens to the indices after an uninsertion in Fig. 7. So how can we update `i` to point to the correct place in the new urn? Again, looking at Fig. 7, we can see that the indices that fall *after* the removed bucket must be shifted down to fill in the uninserted bucket. We also have to address the case where we were supposed to remove the uninserted value; if `i` lay within the removed bucket, then we don't in fact need to call `replace` at all. This accounts for the extra `w'` indices that are valid for the old urn but not the new one.

Thus, `uninsert` must not only return the weight and the value that was deleted, but also enough information to completely identify the removed bucket: its lower bound. The type of `uninsert` is therefore

```
uninsert
  :: Urn a -> ((Weight,a), Weight, Maybe (Urn a))
```

The implementation of `uninsert` is very similar to `insert`. When inserting a value, we follow the path given by the bits of `size` itself and insert a `Leaf`, updating all the parent weights in the process; this produces an `Urn` of size `size+1`. In contrast, to uninsert a value, we just need to follow the path given by the bits of `size-1` and remove the `Leaf` there, again updating internal node weights to maintain the weight-sum invariant.

The bigger difference is that we also need to calculate the lower bound of the bucket of the value we removed. If our tree is just a leaf, `Leaf w a`, then the bucket for that leaf is just `[0,w)`. If our tree is a node, `Node w l r`, then the "super-bucket" for the whole tree is again `[0,w)`, and the two subtrees have "super-buckets" `[0, weight l)` and `[weight l, w)` (as we saw in §3.2). Therefore, since we know which direction to recurse to find the target, we know how to adjust the lower bound returned by the recursive call. If we recursed to the *left*, then the lower bound is unchanged; if we recursed to the *right*, then we need to add `weight l` to the lower bound.

As promised, we can now combine `uninsert` with `replace` to produce `remove`. This breaks down into the following three cases:

- If `i < lb`, then `i` lies before the removed bucket, so it pointed to the same value in `urn` as it now points to in `urn'`; thus, we can use the index `i` as-is when calling `replace`.
- If `lb <= i < lb + w'`, then `i` was in the removed bucket, so the uninserted item *was* the very item we had wanted to remove; this means we can return the pair `(old, Just u')` directly without calling `replace`.
- Finally, if `lb + w' <= i`, then `i` lies after the removed bucket, so the value `i` points to has been relocated by `uninsert`; we need to subtract off `w'` to get the new index `i-w'`.

The Haskell implementation reflects all of these cases directly.

```
remove :: Urn a -> Index
       -> ((Weight,a), Maybe (Urn a))
remove urn i =
  let ((w',a'), lb, maybeUrn') = uninsert urn
  in case maybeUrn' of
       Nothing -> ((w',a'), Nothing)
       Just urn'
         | i < lb ->
           Just <$> replace w' a' urn' i
         | i < lb + w' ->
           ((w',a'), Just urn')
         | otherwise ->
           Just <$> replace w' a' urn' (i - w')
```

3.7 Building Up To (Almost) Perfection

There is only one more nontrivial function from our API that we have not yet discussed: `fromList`.[5] Having already defined `insert`, we could define `fromList` in terms of it:

```
fromList :: [(Weight, a)] -> Maybe (Urn  a)
fromList ((w,a):was) =
  Just $ foldr (uncurry insert) (singleton w a) was
fromList [] = Nothing
```

Most of the time, this implementation will be fine; it runs in linearithmic – $O(n \log n)$ – time, but since each urn will only be initialized once, this overhead is not a problem in practice. Still, we can do better.

The `fromList` function constructs an urn all at once. Any binary tree with n leaves, such as an `Urn` with n values, also has exactly $n-1$ internal nodes. This means that if we could build an `Urn` while spending only constant effort at each node and leaf, we would have a construction algorithm which runs in linear time with respect to the length of the list.

The fold-based algorithm above constructs an urn by iteratively rebuilding it, traversing the tree from root to leaf and modifying its weights and values. Because any such traversal must cost at least logarithmic time, any top-down algorithm must take at least

[4]Even if `updates` have happened in the meantime, recall that changes to the weights do not affect the balance of the tree; only its leaf-node structure affects the balance.

[5]We elide the implementation of `singleton`.

linearithmic time. Instead, to construct an urn in linear time, we need to build it from the bottom-up, starting from the leaves.

A first useful observation is that all urns of a given size have an identical shape – they will only differ in their weights and leaf values. As a result, our construction algorithm need only compute the correct shape for urns with size equal to the length of the input list, summing weights to fill the internal nodes as it goes.

What, then, is the shape of an urn of a given size? We've seen, in Fig. 6, how this shape evolves as successive elements are inserted. At all times, we maintain the invariant that an urn is *almost perfect* – that is, that the difference in depth between any two leaves is at most one.[6] This means that the shape of an urn is restricted to being composed of a perfect tree with an additional "fringe" of pairs of leaves dangling beneath the last fully perfect row of that tree.

Computing an urn's shape boils down to computing the depth of the deepest full level of the tree, and the positions of all the dangling pairs of leaves beneath that level. Were there no such leaves – that is, were the list size a power of two – then we could build the tree in linear time by simple recursion. We present the algorithm parameterized over an arbitrary tree type `t` with node and `leaf` construction functions.

```
perfect :: (t -> t -> t) -> (a -> t)
        -> [a] -> t
perfect node leaf values =
  evalState values $ go perfectDepth
  where
    size         = length values
    perfectDepth = floorLog2 size

    go 0 = do
      [a] <- consume 1
      pure $ leaf a

    go depth =
      node <$> go (depth - 1)
           <*> go (depth - 1)

    consume :: Word -> State [a] [a]
    consume n = state $ splitAt n
```

This computation is `State`ful, storing a list of values that will become leaves; the `consume` operation removes and returns the first `n` elements of that list. The structure of the recursion in `perfect` looks like the desired tree, but we still only consume elements from the list one at a time. We recurse on the depth of the desired perfect tree; when this hits zero, we consume a single element from the input list and convert it into a leaf. At non-zero depths, we simply recurse twice and produce a node whose children are the two resulting perfect trees.

Urns, however, are merely *almost* perfect. When the input list is of size $2^d + r$, where $0 < r < 2^d$, we can handle the extra r elements by sometimes consuming two elements at once and building a node with two leaves as children instead of consuming one element and building a leaf. The tricky part is figuring out when to consume two elements. For example, if we wanted to build a *complete* tree, we could consume two elements the first r times, and then one element (as before) the remaining $2^d - r$ times. However, the urns built up by `fromList` will not be complete; they must have the shape that would have been produced by repeated insertion.

[6]This is weaker than the definition of a *complete* tree, which requires in addition that all leaves on the deepest level are as far left as possible.

What we need to determine, then, is whether we consume 1 or 2 values with the ith `consume` action. For concreteness, consider Fig. 8 which depicts an almost-perfect urn of size 11 and the sequence of `consume` actions that created it. Recall the invariant from the end of §3.5: every root-to-leaf path in this urn, read as a string of bits, corresponds to the reverse of a binary number less than 11; at the same time, every binary number less than 11 corresponds to a root-to-leaf path. We only perform a `consume 2` action when the leaves it produces satisfy this invariant. Let's focus on the node containing `d` and `e`, which is produced by the third `consume` action (with index $i = 2$). These two leaves have paths which correspond to $\texttt{0010}_2 = 2_{10}$ (for `d`) and $\texttt{1010}_2 = 10_{10}$ (for `e`). This was allowed to be a `consume 2` because both 2 and 10 are indeed less than 11. An urn of size 10, on the other hand, would instead contain a leaf (not a node) at this point.

This algorithm is reflected in the `almostPerfect` function below. The local variable `size` is the length of the input list, and is equal to `2^perfectDepth + remainder`. Calling `reverseBits count n` reverses the lowest `count` bits of the word `n`. Finally, we augment our state with a counter which is incremented every time `consume` is called, and use the `index` action to read its value.

```
almostPerfect :: (t -> t -> t) -> (a -> t)
              -> [a] -> t
almostPerfect node leaf values =
  evalState (0,values) $ go perfectDepth
  where
    size         = length values
    perfectDepth = floorLog2 size
    remainder    = size - 2^perfectDepth

    go 0 = do
      ix <- index -- 0 <= ix < 2^perfectDepth
      if reverseBits perfectDepth ix < remainder
        then do [l,r] <- consume 2
                pure $ leaf l `node` leaf r
        else do [a]   <- consume 1
                pure $ leaf a

    go depth =
      node <$> go (depth - 1)
           <*> go (depth - 1)

    index   :: State (Word, [a]) Word
    consume :: Word -> State (Word, [a]) [a]
```

In the code above, we decide whether to `consume 2` or 1 by checking if `reverseBits perfectDepth ix < remainder`, where `ix` is the index of the current `consume` action. This index is a path to the leaf *or node* that this action will produce. The check we described before would correspond to checking that both the reversals of $\texttt{ix} \cdot \texttt{0}$ and $\texttt{ix} \cdot \texttt{1}$ are both less than `size`. The first check is always trivially true, as `ix` is less than `2^perfectDepth` (which is also why our invariant is automatically satisfied in the `consume 1` case). For the $\texttt{ix} \cdot \texttt{1}$ case, the trailing `1` takes on a value of `2^perfectDepth`. We can thus compare the reversal of `ix` without that to `remainder`, which is the number of values beyond `2^perfectDepth`.

This function does indeed run in linear time: we access each list element once, and we perform a constant amount of work to create each leaf and node.

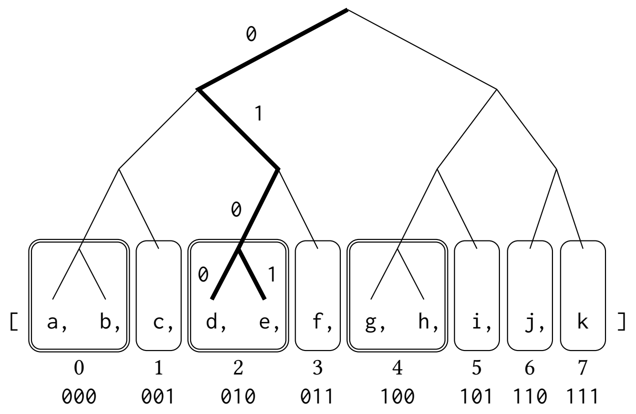

Figure 8. An almost-perfect tree in the shape of a size-11 urn, annotated with the details of our construction algorithm. The leaves are also the input list to the algorithm. The boxes indicate consumption steps: a double-lined box is a `consume 2` step, and a single-lined box is a `consume 1` step. The input list is consumed in order, one step at a time, from left to right; the box with the elements consumed at step i is annotated with i in base 10 and base 2. We also highlight the paths taken to reach into the second input chunk ($i = 2$).

4 Applications and Evaluation

Now that we have defined urns, we explore their applications to random testing and benchmark their performance against existing solutions from the literature.

4.1 An Alternative frequency Combinator

As mentioned in the introduction, the most expressive QuickCheck combinator, `frequency`, allows the user to combine different generators of the same type by choosing one of them based on a discrete distribution. Its implementation is presented in Fig. 9. Every time `frequency` is called, it calculates the sum `tot` of the weight components of the input list, generates a random number between 1 and `tot`, and indexes into the list linearly. For many applications, the input distribution has only a few values, so this approach is reasonably fast. However, the linear traversal of the list can cause unnecessary overheads for medium-to-large inputs.

We propose a new combinator `frequency'` that takes an `Urn (Gen a)` as input (where `Gen` is QuickCheck's type for random generators), using the random `sample` function from Fig. 3.

```
frequency' :: MonadSample m => Urn (m a) -> m a
frequency' = join . sample
```

Its functionality is identical to QuickCheck's `frequency`: we generate a number between zero and the total weight of the urn (`Urn`s are `0`-indexed where `frequency` is `1`-indexed) and index into our structure appropriately. The use of urns provides a lot of flexibility, allowing us to both use the very expressive combinator library of QuickCheck and dynamically change the distributions involved efficiently, as we will see in the rest of this section. Moreover, even in the static case – i.e., the case where we do not modify the distribution – we obtain better performance.

In 2012, Hriţcu et al. [7] explored different generation methods for information flow control stack machines, focusing for the most part on generating "good" instruction sequences; that is, sequences that lead to longer executions. The instructions for their

```
frequency :: [(Int, Gen a)] -> Gen a
frequency [] = error "... empty list"
frequency xs0 = choose (1, tot) >>= (`pick` xs0)
 where
  tot = sum (map fst xs0)

  pick n ((k,x):xs)
    | n <= k    = x
    | otherwise = pick (n-k) xs
  pick _ _  = error "... empty list"
```

Figure 9. The exact implementation of `frequency` from QuickCheck 2.8.2 (with abbreviated string literals) [2, 15].

simple machine are: `Push` and `Pop`, which manipulate the stack; `Add`, which sums the top two items on the stack; `Load` and `Store`, which are memory operations; `Jump`, `Call` and `Return`, which are control flow operations; `Halt`, which signifies a successful termination; and `Noop`, which does nothing. The `frequency` combinator is featured prominently in their development: for every generation strategy they explore other than the very first, naïve, one, individual instructions are generated using `frequency`.

Benchmarking We evaluated the performance of `Urn`s in the static case by testing one of Hriţcu et al.'s early generation strategies (which they call *genWeighted*), which crucially uses `frequency` to increase the probability of `Halt` and `Push` instructions, skewing the distribution of programs towards those that terminate (`Halt`) and do not crash (because they have a big enough stack to avoid underflows). We randomly generated 500 instructions in the form of instruction lists of size 10, and benchmarked the generation time using Criterion [11]. In this benchmark, sampling from `Urn`s is 2.64× faster than using `frequency`.

To further evaluate `Urn`s, we wanted to identify the cutoff point (in terms of input list size) where using an `Urn` for static sampling becomes more efficient. We used Criterion again to benchmark sampling uniformly from the first n integers (where n ranged from 1 through 10 000). For each n, we generated numbers using `frequency` and using `Urn`s; we ran each approach 10 000 times with QuickCheck's `sample'` (which generates 11 values using `IO`), and measured the performance. The results appear in Fig. 10. There was no cutoff: for small distributions ($n \leq 20$), the performance of `Urn`s and lists are the same within the margin of error; and for larger distributions, `Urn`s quickly outpace lists. The run time of `frequency`, as expected, scales linearly with the size of the input list, requiring more than 3 seconds to complete when $n = 10\,000$; on the other hand, the time taken to sample from an `Urn` grows at a much slower rate, rising logarithmically from roughly 50 ms for small inputs to roughly 80 ms for the larger ones. This logarithmic curve can be better seen on the right-hand side of Fig. 10, where we only plot the time needed to sample from urns.[7]

4.2 An Efficient backtrack Combinator

The real benefit of using urns, however, is not just a slight performance boost in the static case. When we wish to dynamically alter the input distribution, urns greatly improve the performance and

[7]All the benchmarks in this paper were run on a Dell XPS15 laptop with a 2.3 GHz Intel Core i7-4712HQ with 16 GB of RAM running Ubuntu 16.04.2 LTS; they were compiled with GHC 8.0.2 using `-O2 -funbox-strict-fields`.

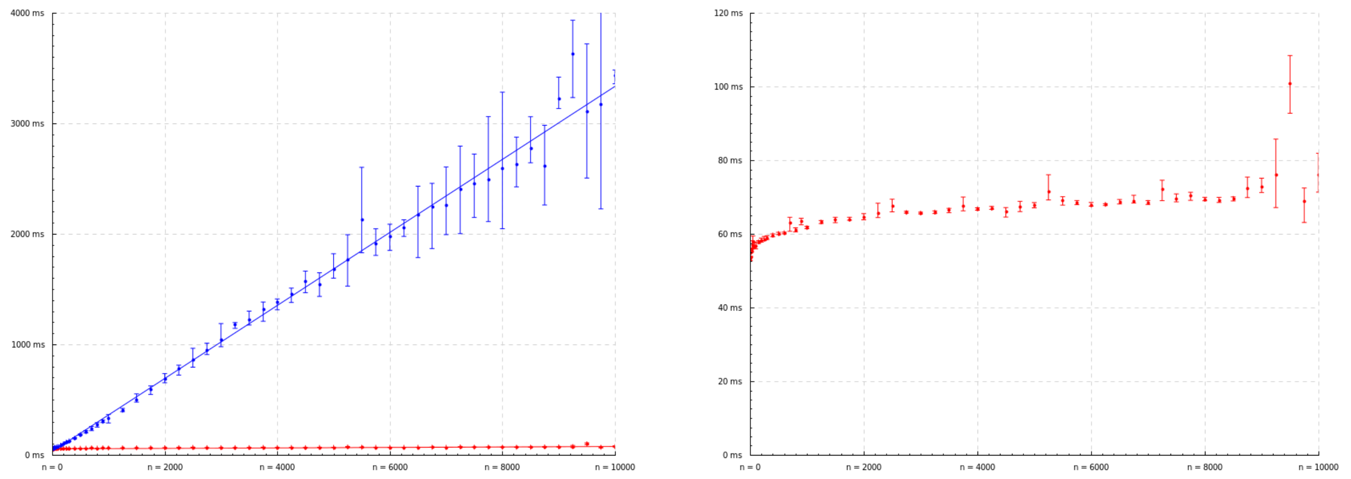

Figure 10. Left: Performance of `frequency` (blue, above) vs. `Urns` (red, below). Right: Zoomed-in performance of `Urns`.

conciseness of our code. This desire to update a distribution that is being sampled from often arises in random testing when some generators may fail to produce a value (i.e., return `Nothing`). If we sample from a generator and the generated value is not a `Just`, we must backtrack and try again.

As an example, consider the inspiring work of Pałka et al. [13] on generation of well-typed lambda terms. To generate well-typed terms, the authors use the typing rules of simply typed lambda calculus as generators. They assign an empirically-chosen weight to each rule; then, to generate a term with type `T`, they pick a rule whose conclusion has type `T` at random based on the weights. This rule may then have premises, which they attempt to satisfy recursively in the same way. For example, to generate a term of type `Int`, we could either use some `Int` constant like `0` or `1`; some variable `x` from our environment; or a function application `f e'` where `f :: T' -> T` and `e' :: T'` for any type `T'`.

However, the premises of these typing judgments may not be satisfiable. For instance, there might not be any `Int` variables in the environment. Worse, when using the application rules, `T'` is chosen arbitrarily, but there is no guarantee that we can generate a term of type `T'` within the constraints of the generation process. When a typing judgement is not satisfiable, Pałka et al. resort to backtracking: they randomly select the next applicable rule. When the remaining rules are exhausted, generation fails and they backtrack at some higher level if possible.

The way Pałka et al. choose a rule randomly from a weighted distribution is by permuting the entire list using a variant of permutation-by-sorting shown in Fig. 11, and then iterating through this permuted list as necessary. Standard permutation-by-sorting shuffles a list by generating a random number for each list item, and sorting the list by comparing these numbers.[8]

Pałka et al. extend permutation-by-sorting to take (positive, integral) weights w into account by generating w numbers for each item in the list and picking the minimum as the key for sorting. Intuitively, this approach simulates exploding a w-weighted item into w identical copies, using permutation by sorting to shuffle the exploded list and then keeping the first occurrence of each item in the result.

[8]One downside of permutation-by-sorting is that it only guarantees a fair shuffle if the generated comparison keys are unique. This is typically avoided by using a fair shuffling algorithm like Fisher-Yates [3]; however, this algorithm does not have a natural extension that takes weights into account. Thankfully, the unfairness is not a major concern, since the weights in random testing are typically tuned based on the observed behavior.

```
-- Weights must be positive!
permuteWeighted :: [(Int, a)] -> Gen [a]
permuteWeighted xs = do
  v <- mapM (\n  -> replicateM n arbitrary >>=
             \ns -> return $ minimum ns)
            (map fst xs) :: Gen [Int]
  let p = map snd $ sortBy (comparing fst)
                  $ zip v [0..]
  return $ map ((map snd xs)!!) p
  where l = length xs
```

Figure 11. The implementation of `permuteWeighted` from Pałka et al. [13] (reformatted).

The algorithm in Fig. 11 has several inefficiencies; apart from implementation details (the use of `!!` could be replaced by `zip`ping with `xs` directly), there are two more fundamental problems. First and foremost, the complexity of the algorithm is pseudo-polynomial; given the weights $\overline{w_i}$, the algorithm runs in $O(n \log n + \sum_i w_i)$ time, since we generate w_i keys for every item before sorting. Secondly, we only need the later items from the shuffled list if we actually backtrack. Thanks to laziness, we may be able to avoid spending the full $O(n \log n)$ time to permute the list, but this depends on the precise sorting algorithm used.

We can avoid completely shuffling a list of generators by putting them in an `Urn`, using the `backtrack` combinator:

```
backtrack :: Urn (Gen (Maybe a)) -> Gen (Maybe a)
backtrack urn = do
  ((_w,g), mUrn') <- remove urn
  ma <- g
  case ma of
    Just a  -> pure $ Just a
    Nothing -> maybe (pure Nothing) backtrack mUrn'
```

In `backtrack`, we `remove` a generator from the urn, sample from it, and test to see if we got a result. If so, we `Just` return that result; otherwise, we repeat this process until the urn is empty. Since each `remove` operation only takes $O(\log n)$ time, this combinator runs in $O(n \log n)$ time (with respect to the running time of the generators).

A similar construct exists in QuickChick [14], the Coq implementation of QuickCheck, but is based on lists; `backtrack` could be used to make this more efficient. In general, analogous situations arise in many other testing applications, such as in explicitly weighted narrowing approaches (e.g., Luck [9]). At a more abstract level, urns can be used to efficiently tune randomized search algorithms that choose between prioritized possibilities.

Benchmarking To evaluate urns in this context, we replaced `permuteWeighted` in Pałka et al.'s code [12] with a variation using urns, which inlines the aforementioned `backtrack` combinator:

```
permuteWeighted :: [(Int, a)] -> Gen [a]
permuteWeighted x =
    let Just u = Urn.fromList
               $ map (first fromIntegral) x
        aux u = do
          (w, a, mu) <- Urn.remove u
          case mu of
            Just u' -> (a:) <$> aux u'
            _       -> pure [a]
    in aux u
```

We benchmarked the generation time of 11 Haskell terms (as many as produced by QuickCheck's `sample'`), without altering anything else in their code. Criterion reported a 31.52 ms expected time (1.8 ms variance) for the original code; on the other hand, the urn-based version took 14.95 ms (1.6 ms variance).

While this 2.1× speedup is a victory in its own right, it doesn't measure the real difference between the two variants of `permuteWeighted`. For one, the permutation algorithm is clearly not the bottleneck amongst 1600 lines of complicated Haskell dealing with polymorphic unification. More importantly, because the Pałka variant of `permuteWeighted` has quasi-polynomial running time, it is not general purpose: it is only efficient if all the weights are small. On the other hand, the urn-based variant can be used as-is in any development. Indeed, if we were to directly benchmark the two variants, we could artificially inflate the difference as much as we wanted by choosing arbitrarily large weights.

5 Related Work

5.1 Alternative Representations of Discrete Distributions

The literature contains several extant representations for discrete distributions. For example, the QuickCheck [2, 15] and `random-fu` [16] packages both provide support for sampling from such distributions, using lists and arrays, respectively. For each of these approaches, we present in Fig. 12 its asymptotic run time for initialization and for performing the four operations we want to support, as compared with the run time of an urn for the same operation. The table highlights the trade-offs between these data structures: urns, like arrays, cost linear time to create, and take logarithmic time to sample from; on the other hand, lists are simple and so benefit from constant-time initialization (`id`) and insertion (`(:)`), but require linear time when sampling. Only urns, however, are designed to support the three dynamic update operations, and are the only structure to achieve consistent logarithmic performance.

Operation	Lists	Cumulative arrays	Urns
Create from list	$O(1)$	$O(n)$	$O(n)$
Sample	$O(n)$	$O(\log n)$	$O(\log n)$
Total weight	$O(n)$	$O(1)$	$O(1)$
Insert	$O(1)$	$O(n)$	$O(\log n)$
Remove	$O(n)$	$O(n)$	$O(\log n)$
Update/Replace	$O(n)$	$O(n)$	$O(\log n)$

Figure 12. Comparison of runtimes for operations on different functional distribution data structures; n is the number of values in the distribution.

The basic idea behind sampling from a discrete distribution was discussed in §2: given the distribution $\{(w_1, x_1), \ldots, (w_n, x_n)\}$, we break the range $[0, \sum_{i=1}^{n} w_i)$ into n subranges ($[0, w_1)$, $[w_1, w_1 + w_2)$, etc.) and generate a random number r from the total range; the index of the subrange r belongs to is the index of the desired value. Urns, QuickCheck and `random-fu` follow the same high-level approach, but use different data structure representations.

QuickCheck Perhaps the simplest representation of a discrete distribution over values of type `a` is `[(Weight,a)]` – a list of values paired with their weights, as discussed in §1. This is the representation used by QuickCheck's `frequency` combinator (Fig. 9) [2, 15], and is considered in column 1 of Fig. 12.[9]

This representation is very simple, uses only standard data types, and requires only simple, local invariants on the input data: that the list is nonempty and that its weights aren't all 0. However, as we saw in §4.1, while this simple representation works for small cases it has some obvious inefficiencies: recalculation of the total weight and worst-case linear traversal to generate samples.

On the plus side, inserting a new value into the distribution is constant-time: if we want to add the new value `x'` with weight `w'` to the distribution `d`, we can simply cons them onto the front to produce the new distribution `(w',x'):d`. Because all the invariants are local, no other computation is needed. Other modifications – deleting a value, replacing a value, or updating the weight of a value – take linear time in the worst case, however, as modifying the structure of a linked list always does.

`random-fu` The `random-fu` package [16] uses a similar representation for discrete distributions – also called categorical distributions – in its `Data.Random.Distribution.Categorical.Categorical` type. However, it instead uses an array (specifically, a `Vector` from the `vector` package [5]) and pairs values with their *cumulative* weights: the distribution

$$\{(w_0, x_0), (w_1, x_1), \ldots, (w_n, x_n)\}$$

becomes the array

```
[(w0,x0), (w0+w1,x1), ..., (w0+w1+...+wn,xn)]
```

This representation is considered in column 2 of Fig. 12. In this regime, our example urn becomes `[(2,R), (6,G), (9,B)]`. Now,

[9]The `frequency` function actually has type `[(Int, Gen a)] -> Gen a`, as discussed in §4.1, so it deals with a "distribution over distributions"; however, all the representations function equally well holding `a`s and `Gen a`s, so we elide this extra detail.

the total weight is stored in the last position in the vector – accessible in constant time – and we can use binary search to find which bucket an element of [0, `total`) belongs to in logarithmic time. This works because each position in the vector stores the upper bound on its bucket, which is what the index needs to be compared against.

As it is not a design goal of the library, `random-fu` itself does not expose any operations for dynamic updates. Nevertheless, we can consider how this representation would work if we were to extend it to support them. As it happens, this representation requires linear time for all updates. For delete, update, and replace, this is independent of the runtime of the array operations; since the array stores *cumulative* weights, modifying any weight in the middle of the array requires modifying all subsequent weights as well. To insert a value, we can add a new value to the end of the array without the need to update any other weights; however, because our arrays are immutable, this still requires copying the entire array and thus takes linear time. If our distribution were mutable (in `ST` or `IO`), we could get amortized constant-time append, and thus improve the efficiency of insertion (but, due to the cumulative weights, not remove, update, or replace).

5.2 Alternative Tree-Based Structures

Urns are reminiscent of other data structures based on complete binary trees: certain variants of heaps store a tree linearized into an array, with the children of node i at indices $2i + 1$ and $2i + 2$ (using 0-indexing). If we always fill the array from left to right – taking care to convert leaves into nodes when necessary – then we will always have a balanced tree. However, as with all array-based representations, updates in a purely functional setting require copying the entire array, and so cost $O(n)$; we get no sharing at all. Urns provide an elegant, purely functional alternative, filling a very real need in the community. Additionally, since urns are immutable trees with no important laziness properties, they may be used in a persistent context with the same performance as when used ephemerally. That is to say, reverting to any previous state of an urn costs nothing.

In the functional setting, there already exist self-balancing data structures like red-black trees and AVL trees. However, these data structures maintain complicated invariants, and are notoriously difficult to get (and prove) correct [1]. Moreover, QuickCheck generators, one of the main applications of urns, cannot be given an `Ord` structure: they are implemented as Haskell functions. Since urns do not maintain a specific arrangement of their values, they can contain generators, as well as arbitrary functions, `IO` actions, or other objects without imposing any constraints.

Finally, Okasaki's binary random-access lists (BRALs) [10] – and similar structures based on numeric representations – have an indexing scheme that is very similar to the one urns use when inserting new elements. Both BRALs and urns take an index and repeatedly use its least-significant bit to traverse the tree. The difference is that BRALs are designed to allow efficient access by index; urns only use indexing when inserting and uninserting elements. The sampling process is instead based on weights, which have no analog in BRALs. We could emulate this sampling behavior by using the same trick that Pałka et al. use in `permuteWeighted` [13] (see §4.2): when inserting an element with weight w, insert w copies of it into a BRAL. However, this takes pseudo-polynomial time and space; it also doesn't support any of the urn operations, such as `remove`, which look at all those copies at once.

6 Conclusion and Future Work

In this paper we presented the *urn*, a simple tree-based data structure that allows for sampling from and updating discrete distributions with logarithmic performance in the worst case, and demonstrated its usefulness in existing random testing applications from the literature.

Looking to the future, a natural question is to optimize instead for *expected case* performance. For example, suppose we had the distribution $\{(1\,000\,000, R), (2, G), (2, B)\}$. In this case, we will pick R 99.9996 % of the time, so it would be convenient if R were stored as high up as possible in the tree, even if this pushes G and B deeper; however, our implementation of `Urns` will make sure all values are equally deep. If we were to locate values in the internal tree of an urn according to the length of their Huffman code [8], the most heavily weighted values would take the fewest steps to reach, and thus the least time to generate. Investigating the possibility of an imbalanced urn-like data structure that allows for better expected case performance could yield an even greater efficiency boost in many random testing applications.

Another possible direction would be to try *wider* tree representations, similar to those used in Clojure [6]. If we index into 2^n-ary trees using a base-2^n representation, we could decrease the constant factor associated with the number of steps required to reach an value in the urn. This could further enhance the efficiency of the structure.

Acknowledgments

We are grateful to Benjamin Pierce, Cătălin Hriţcu, George Karachalias, Christine Rizkallah, Alex Roederer, Nikos Vasilakis, Jennifer Paykin, Robert Rand, the entirety of the Penn PL Club, and the anonymous reviewers for their helpful comments. This work was supported by NSF award #1421243, *Random Testing for Language Design*, NSF award #1521523, *Expeditions in Computing: The Science of Deep Specification*, NSF award #1319880, *Rich Type Inference for Functional Programming*, and NSF award #1218002, *Beyond Algebraic Data Types: Combinatorial Species and Mathematically-Structured Programming*.

References

[1] A. Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.
[2] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2000.
[3] R. Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420–, 1964.
[4] C. Gibbard, B. Yorgey, et al. `MonadRandom`: Random-number generation monad. http://hackage.haskell.org/package/MonadRandom-0.4.2.3, 2016.
[5] Haskell Libraries Team and R. Leshchinskiy. `vector`: Efficient arrays. http://hackage.haskell.org/package/vector-0.11.0.0, 2015.
[6] R. Hickey. The Clojure programming language. http://clojure.org, 2012.
[7] C. Hriţcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos. Testing noninterference, quickly. In *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2013.
[8] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
[9] L. Lampropoulos, D. Gallois-Wong, C. Hritcu, J. Hughes, B. C. Pierce, and L. Xia. Beginner's Luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, 2017.

[10] C. Okasaki. *Purely Functional Data Structures.* Cambridge University Press, New York, NY, USA, 1998.
[11] B. O'Sullivan. Criterion: a Haskell microbenchmarking library. http://www.serpentine.com/criterion/, 2014.
[12] M. H. Pałka. Testing an optimising compiler by generating random lambda terms. http://www.cse.chalmers.se/~palka/testingcompiler/.
[13] M. H. Pałka, K. Claessen, A. Russo, and J. Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test.* 2011.
[14] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. In C. Urban and X. Zhang, editors, *6th International Conference on Interactive Theorem Proving (ITP).* 2015.
[15] QuickCheck developers, N. Smallbone, B. Bringert, and K. Claessen. QuickCheck: Automatic testing of Haskell programs. http://hackage.haskell.org/package/QuickCheck-2.8.2, 2016.
[16] D. Steinitz and J. Cook. `random-fu`: Random number generation. http://hackage.haskell.org/package/random-fu-0.2.6.2, 2015.