Reflecting on Random Generation

SAMANTHA FROHLICH, University of Bristol, UK
HARRISON GOLDSTEIN, University of Pennsylvania, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA
MENG WANG, University of Bristol, UK

Hand-written *custom generators* in frameworks like QuickCheck [1] are crafted to generate values satisfying application-specific validity conditions, while optimising distributional properties. If they are written well, custom generators can do a great job at their primary task of sampling random data. But can they do anything more?

Yes, much more! We show that an enriched generator abstraction, *reflective generators*, can facilitate a diverse collection of improvements at multiple points in the testing process. First, reflective generators allow powerful introspection that helps programmers *validate* generators as they write them. Second, they can be used to *analyse* testing efficacy via black-box metrics like combinatorial coverage [3], even when test cases are not produced by the generator itself. Finally, they provide common infrastructure for *implementing* and indeed *generalising* state-of-the-art "dynamic" testing strategies [10, 24]; in particular, they enable a novel approach to validity-preserving mutation for adaptive fuzzing.

1 INTRODUCTION

Users of testing frameworks like QuickCheck [1] often write "custom generators" to sample random data from finely tuned distributions or to satisfy application-specific validity conditions. These generators are mainly used to produce random values, e.g., for property-based testing (PBT), but with a little extra effort, they can do much more.

Prior work has already explored how tweaking the generator abstraction can lead to richer functionality. In particular, *bigenerators* [26], written in an annotated version of QuickCheck's monadic generator language, act like normal generators of random values when run "forward". Moreover, they can be run "backward" to analyse a value and determine whether it satisfies the generator's validity condition—i.e., whether the forward direction can produce this value with non-zero probability. But this clever idea is only the tip of the iceberg.

We present *reflective generators*, a further refinement of the bigenerator framework. These generators are "reflective" in the sense of bidirectionality—like a beam of light bouncing off a mirror—but also in the sense of mindful reflection: the internal choices that the generator makes to produce a value are made available for external analysis and direct manipulation. Reflective generators provide a unified approach to problems throughout the testing process: in particular, they provide tools for improving generator validation, test suite analysis, and strategies for dynamic testing. We discuss these in turn.

Generator Validation. Conventional wisdom says that code for testing should be as simple as possible, to reduce the likelihood of bugs in the testing framework itself. Unfortunately, generators for values with non-trivial validity conditions are often quite complicated. Such generators can produce invalid values or, more insidiously, can miss important parts of the valid space of inputs. Tuning a generator to obtain a desired probability distribution can also be frustratingly difficult [9]. Our framework gives programmers new ways to understand the behaviour of their generators; all they need to do is to add some straightforward "bidirectionalisation annotations" to unlock the ability to extract distributional information.

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

Test Suite Analysis. There are many metrics for measuring testing efficacy, and no single one gives a complete picture. Thus, it is best to use a variety of metrics to analyse a test suite. One compelling class of metrics is black-box *combinatorial* analyses. The simplest metric in this class is constructor frequency (how frequently each constructor of a given datatype appears in a test suite), which gives a high-level signal of potential gaps in the test distribution. More involved metrics like the combinatorial coverage metric in *Do Judge a Test by its Cover* [3] give deeper ways to understand how well a test suite covers the space of possible inputs. These metrics are typically computed in an ad-hoc way for each data type, but our framework, through the introduction of choice labels, can support them generically via information already available in the generator.

Dynamic Testing Strategies. While vanilla QuickCheck works fine for many properties, more guided or adaptive strategies can sometimes do a better job at finding bug-provoking inputs. This is where reflective generators shine brightest. By utilising both forward and backward interpretations, reflective generators generalise core ideas behind two state-of-the-art testing approaches. In the world of example-based tuning, reflective generators can replicate and generalise results from *Inputs from Hell* [24], lifting a useful testing technique from grammar-based testing to more powerful monadic generation. In the world of adaptive fuzzing (*á la* FuzzChick [10]), reflective generators provide a novel, generic way to mutate values while preserving tricky validity conditions.

After some technical background (§2), we offer the following contributions:

- We enrich bigenerators by annotating them with semantic choice labels, creating *reflective generators* (§3).
- We show how to use reflective generators for introspection, validating their own correctness and distributional properties and giving insights that would otherwise require tedious manual measurements (§4).
- We show that reflective generators can streamline two test suite analyses that are used to understand real-world testing efficacy [3] (§5).
- We use reflective generators to generalise an existing approach to example-based testing [24] and develop a novel technique for validity-preserving mutation that enables better adaptive fuzzing [10] (§6).

In all, we give eight interpretations of reflective generators, each providing testing value for a low cost that is paid only once. We conclude with a discussion of limitations (§7), references to related work (§8), and ideas for future research (§9).

2 BACKGROUND

Our reflective generator abstraction is built on ideas from the worlds of testing and bidirectional programming. This section reviews the background that informs the rest of our contributions.

2.1 QuickCheck Generators

Binary search trees (BSTs) are a classic example for property-based testing. Consider this `Tree` datatype and validity function, `isBST`, that decides whether or not a `Tree` is a binary search tree (BST):

```
data Tree = Leaf | Node Tree Int Tree
isBST :: Tree -> Bool
```

The following QuickCheck property can check that an `insert` function preserves BST validity:

```
prop_insertValid :: Int -> Tree -> Property
prop_insertValid x t = isBST t ==> isBST (insert x t)
```

To test this property, QuickCheck generates hundreds or thousands of integers and trees, checking that `prop_insertValid x t` is true for all of them. However, the likelihood of the tree generator randomly stumbling on valid BSTs—especially large ones—is relatively low, so random trees are likely to fail the `isBST t` check, thus making the test vacuously succeed. This is a typical situation where rejection sampling (sampling values from a larger space and filtering out the ones that are invalid) does not perform well. Luckily, libraries like QuickCheck provide means for creating better "constructive" generators that can, for example, enforce the `isBST` validity condition by construction:

```
genBST :: (Int, Int) -> Gen Tree
genBST (lo, hi) | lo > hi = return Leaf
genBST (lo, hi) = frequency
                    [ ( 1, return Leaf      ),
                      ( 5, genNode (lo, hi) ) ]

genNode (lo, hi) = do
  x <- chooseInt (lo, hi)
  l <- genBST (lo, x - 1)
  r <- genBST (x + 1, hi)
  return (Node l x r)

prop_insertValid' :: Int -> Property
prop_insertValid' x = forAll genBST $ \t -> isBST (insert x t)
```

This generator is written in monadic style using QuickCheck combinators. Its definition revolves around the desired validity condition, using an argument to ensure that only integers within the correct range are used. The base case uses the monadic `return` to construct a degenerate generator that always produces a `Leaf`. Then, the inductive case uses the `frequency` combinator to make a weighted choice between the `Leaf` and `Node` constructors, where the nodes are generated using do notation to naturally compose together the call to `chooseInt` for selecting its value the recursive calls for child trees.

2.2 Bigenerators

Constructive generators like `genBST`, which are carefully designed to only produce values that satisfy a given validity condition, bake in a lot of knowledge about how that condition is satisfied. Ideally, this information could be used for other purposes. This is exactly the idea behind one of the examples in *Composing Bidirectional Programs Monadically* [26], which makes QuickCheck-style generators *bidirectional*: they can run forward as generators, and they can also run backward as checkers for the validity condition that the generator direction enforces.

To enable backward interpretation, the generator monad is combined with a profunctor using a `Profmonad` class, which just insists that something is both a `Monad` and a `Profunctor`. Profunctors are two-argument functors that are contravariant in their first parameter, and covariant in the second. In other words, the second parameter acts like a normal functor, but the first flips the arrows, applying functions "backward", making them intuitively the perfect structure for encapsulating bidirectionality.

```
class Profunctor p where
  dimap :: (a -> b) -> (c -> d) -> p b c -> p a d
```

More concretely, consider genNode. In order to check that a value can be produced by this generator, there are two things to verify: the component parts (x, l, and r), and how they are composed. The former can be achieved recursively by bidirectionalising the generators of the component parts. The latter is handled by the contravariant argument to dimap, which acts as a witness of how b was used to create a. For example, the following function "inverts" the construction of a Node to extracts its Int component:

```
nodeValue :: Tree -> Int
nodeValue (Node _ x _) = x
```

But what happens when nodeValue is given a Leaf? More generally, what happens if the backwards direction of a genBST bigenerator is given a tree that is not a BST? To accommodate such scenarios, bigenerators must also handle failure—i.e., they must be instances not just of Profunctor but of ProfunctorPartial:

```
class Profunctor p => ProfunctorPartial p where
  toFailureP :: p u v -> p (Maybe u) v
```

Wrapping the contravariant parameter in a Maybe allows functions like nodeValue to fail gracefully on inputs like Leaf. A Nothing result signals that the b argument *cannot* be used make an a. This broadens the applicability of monadic profunctors for bidirectional programming, since they are no longer limited to perfect round trips.

Combining a PartialProfunctor with a Monad gives everything needed to build a bigenerator. For example, here is a bidirectionalised version of the part of genBST that generates nodes:

```
biGenNode (lo, hi) = do
  x <- (dimap nodeValue id  . toFailureP) (biChooseInt (lo, hi))
  l <- (dimap nodeLeft  id  . toFailureP) (biGenBST (lo, x - 1))
  r <- (dimap nodeRight id  . toFailureP) (biGenBST (x + 1, hi))
  return (Node l x r)
```

The structure is the same as before; each subgenerator is replaced with a bigenerator and annotated with a call to dimap that adds information about what part of the overall structure it constitutes. This is the key that enables backward interpretation. For example, when the value Node Leaf 5 Leaf is passed to the backward direction of biGenNode (-10, 10), the call to nodeValue extracts 5 and ensures that it can be generated by biChooseInt (-10, 10). In this way, every sub-structure is checked.

3 REFLECTIVE GENERATORS

The bigenerators presented by Xia et al. are quite useful, but with a few adjustments they can be given significantly more power.

3.1 The Reflective Abstraction

Our insight is that the choices that a generator makes while producing a value can be discovered *post-hoc* by running the generator backward. Reflecting on choices like this is the key to implementing the myriad of applications in the remainder of the paper.

But what is a generator choice, specifically? The standard bigenerator framework gives simple combinators like choose to sample from a range of values, but reflective generators are built using a more semantically meaningful notion of choices, represented by the type class:

```
class Pick g where
```

```
pick :: (Eq t, Eq a) => [(t, g t a a)] -> g t a a
```

This `pick` function takes a list of generators (represented by the type `g t a a`), each of which is paired with a semantic label or *tag*, `t`, that can be used to identify that choice when executing in the backward direction.

To choose between a few integers, one could write:

```
pick [("one", pure 1), ("two", pure 2), ("three", pure 3)]
```

In this case we instantiate the type parameter `t` with `String` (and we will continue to do so in the remainder of the paper), but in principle `t` could have rich, domain-specific structure.

Now to actually define *reflective generators*. We combine `Pick` with the bigenerator classes from the previous section to create:

```
class
  ( Pick g,
    forall t. Profmonad (g t),
    forall t. ProfunctorPartial (g t)
  ) =>
  Reflective g
```

Every `Reflective` supports the operations from `Pick` (enabling labelled choice), `Profmonad` (all operations from both `Monad` and `Profunctor` that enabling sequencing and backward computation), and `ProfunctorPartial` (the backward direction is allowed to fail). Together, these operations are exactly what is needed to do complex, bidirectional, labelled generation.

But first, what does a reflective generator look like? Figure 1 shows one for BSTs that will as a running example in this paper. The recipe for converting a QuickCheck generator (shown on the left) into a reflective generator (on the right) is fairly straightforward:

(1) Replace `Gen` with `Reflective` and `frequency` and `oneof` with `pick`.
(2) Add a descriptive tag at each choice point.
(3) Add backward annotations to guide the behaviour of the backward direction.

For `genBST`, the backward annotations show the backward direction of the generator when the value should be a `Leaf` and, if the value is a `Node`, how to extract its arguments. In general, an annotation is needed on all sub-generators, including base-cases, to ensure that the backward direction works properly.

```
genBST ::
         (Int, Int) -> Gen Tree
genBST (lo, hi) | lo > hi = pure Leaf
genBST (lo, hi) =
  frequency
    [ ( 1, return Leaf ),
      ( 5, do
          x <- genInt (lo, hi)
          l <- genBST (lo, x - 1)
          r <- genBST (x + 1, hi)
          return (Node l x r) ) ]
```

```
genBST :: forall g. Reflective g =>
         (Int, Int) -> g String Tree Tree
genBST (lo, hi) | lo > hi = pure Leaf `at` _Leaf
genBST (lo, hi) =
  pick
    [ ( "leaf", return Leaf `at` _Leaf),
      ( "node", do
          x <- genInt (lo, hi) `at` (_Node . _2)
          l <- genBST (lo, x - 1) `at` (_Node . _1)
          r <- genBST (x + 1, hi) `at` (_Node . _3)
          return (Node l x r) ) ]
```

Fig. 1. Converting a QuickCheck generator to a reflective one.

All of these annotations use the convenient `at` combinator that we provide, which does much the same thing as `dimap` and `toFailureP` above. The main difference is that rather than normal (partial) functions, the `at` combinator uses *prisms* provided by the lens [1] library that can be conveniently derived via metaprogramming. The `_Leaf` prism simply checks that the value is, in fact, a `Leaf`, and the `_Node` prism extracts arguments from the `Node` constructor. The `_1`, `_2`, and `_3` prisms extract the first, second, and third argument respectively. If this is all too Haskelly for you, it is also totally fine to annotate using a simple function instead, but we think that this presentation is quite natural once you get used to it.

In fact, this annotation scheme is so natural that we have had little trouble applying it to real-world code. We converted all of the generators in xmonad[2] to reflective generators! The whole exercise took only a few hours, and much of that time was spent understanding the domain types and testing goals. Of course, there are a few limitations to conversion (which did not come up in xmonad), but to avoid getting bogged down here we explore them in §7.

3.2 Interpretations

Once a generator has been converted to reflective style, it can be interpreted in a variety of ways. Rather than pick a particular type that fits the `Reflective` interface, reflective generators like `genBST` are written with an abstract type like `forall g. Reflective g t b a`. This means that the way a particular reflective generator behaves is totally dependent on how the type variable `g` is instantiated. Critically, there is no need to pick just one instantiation for `g`, so these structures are exceedingly flexible. This kind of unified "classy" interface is usually referred to as a tagless final embedding [7] and is a staple of embedded domain specific languages.

As a coherence check, let's make QuickCheck's `Gen` monad a member of the `Reflective` class:

```
newtype Gen t b a =
  Gen {run :: QC.Gen a}
```

Since this type is a wrapper around `Gen`, it is a monad, and it can trivially be made a profunctor by ignoring the contravariant parameter. This means that the only non-trivial instance needed to get a `Reflective` is `Pick`:

```
instance Pick Gen where
  pick = Gen . QC.oneof . map (run . snd)
```

The implementation of `pick` ignores the tags using `snd`, then picks from the resulting list of ordinary generators using `oneof`.

With these instances, the universally quantified type of `genBST` can be specialised to `QC.Gen` using `run` and used as a generator:

```
      genBST (-10,10)  :: forall g. Reflective g => g      String Tree Tree
run (genBST (-10,10)) ::                            QC.Gen        Tree

> QC.sample (run (genBST (-10,10)))
Leaf
> QC.sample (run (genBST (-10,10)))
Node Leaf (-4) (Node Leaf 10 Leaf)
```

A different interpretation of the `Reflective` class replicates Xia et al.'s generators-and-checkers. This time, `genBST` will be specialised to `UnGenCheck`:

[1] https://hackage.haskell.org/package/lens
[2] https://xmonad.org/

```
newtype UnGenCheck t b a =
  UnGenCheck {run :: b -> Maybe a}
```

This structure is both a monad and a profunctor, as demonstrated in the original bigenerator work, and its `Pick` instance looks like this:

```
instance Pick UnGenCheck where
  pick xs = UnGenCheck (\b -> head <$> mapM (\x -> run (snd x) b) xs)
```

The `pick` function goes through the listed generators (ignoring tags with `snd`), running each on the `b` value that is currently being analysed. As long as one of the generators in `xs` can produce `b`, the result of `pick` will be a `Just`, signalling that the generator can in fact produce the desired value.

Interpreting `genBST (-10, 10)` using a wrapper function `unGenCheck` works as intended:

```
unGenCheck :: (forall g. Reflective g => g t a a) -> a -> Bool
unGenCheck x a = isJust (run x a)

> unGenCheck (genBST (-10,10)) Leaf
True
> unGenCheck (genBST (-10,10)) (Node Leaf (-4) (Node Leaf 10 Leaf))
True
> unGenCheck (genBST (-10,10)) (Node Leaf 13 Leaf)
False
```

These examples show that our abstraction recovers the behaviours of both standard QuickCheck generators and bigenerators, but this is just the first step. The beauty of reflective generators is that they can go *far beyond* simple test generation and validity checking. Other interpretations are just a few type class instances and annotations away; moreover, there is rich information in the `pick` tags that neither of the above interpretations capitalise on. The next few sections explore more ways that reflective generators can be used.

4 GENERATOR VALIDATION

While conventional testing wisdom recommends that testing code be as simple as possible, it is hard to avoid the fact that generators are complex programs. Accordingly, it is useful to be able to validate that a generator behaves as expected. This section describes two interpretations of reflective generators that allow for this sort of validation, giving the tester the ability to query properties of the generator's distribution.

4.1 Testing Generators

There are two key properties for a generator that aims to respect some validity condition: soundness and completeness. Soundness asserts that every value generated by the generator is valid. Completeness asserts that all valid values can be generated.

Careful QuickCheck users are already in the habit of checking their generators for soundness. For the BST example, this means testing the following property:

```
prop_genBSTSound = forAll (gen genBST) (\t -> isBST t)
```

Note that this property does not involve the program under test; it is purely for the purpose of testing the generator. Assuming that there is a checker `isBST` (a reasonable assumption, since checking is often simpler than generation), the property checks that any tree produced by `genBST` satisfies `isBST`.

Completeness is as important as soundness, if not more: we might tolerate spurious bug reports due to unsound test inputs, but missing out on whole categories of test inputs unknowingly may result in missing bugs entirely. Despite this, as far as we are aware, there is no good way to check completeness. Reflective generators are the key that unlock this! The previous section shows that `unGenCheck` is able to tell whether a given value can be generated. A bit more development may scale this up. In order to check that `isBST` is complete, a tester can do the following:

(1) Create a basic generator that is obviously complete (but not necessarily sound, and not necessarily as efficient as the one whose correctness we are testing). There are automated tools for creating such naive generators [15, 16]. In the BST example, this could be `genTree`, a generator for any value of type `Tree`. (Remember that `genTree` would not be a good choice for the actual testing, since many of the values it produces will not satisfy `isBST`.)
(2) Check the property:

```
prop_genBSTComplete = forAll genTree
  (\t -> isBST t ==> unGenCheck genBST t)
```

This tries to ensure that for every value `t` of type `Tree`, if `isBST t` then `genBST` can generate `t`. (Not terribly efficient, but a valid test for completeness!)

This works for the same reason that `unGenCheck` works above: running the generator as a checker is really just asking if the value is in the generator's range.

Of course, step (2) is quite slow: it requires that `genTree` stumble upon valid values without any programmer guidance. Fortunately, this completeness testing only needs to be performed when the *generator* is written or changed. Also, we may sometimes be able to use a more sophisticated reference generator that performs better than `genTree`. The space between the most basic generator and a practical implementation that considers validity conditions, distribution, and efficiency is large, and the above approach can be used in a spectrum of scenarios, including regression testing where a newer version of a generator can be tested against an older one that it replaces.

Testing soundness and completeness is an important first step when evaluating the quality of a generator, but it does not imply that the generator has a good *distribution*. We address this next.

4.2 Probabilistic Analysis

Having soundness and completeness is not enough for random testing. As the input space is typically large, often infinite, the theoretical possibility that a certain test input is generated (completeness) is not sufficient for effective testing. Quoting Hughes (one of the creators of QuickCheck): "it is meaningless to talk about random testing without discussing the distribution of test data" [5]. Indeed, without distributional knowledge, a tester might happily rest on their laurels, thinking that their program is thoroughly tested, when in truth some important part of the input space is so unlikely to be covered that it is essentially untested.

To this end, QuickCheck provides tools for tracking and reporting the kinds of inputs that are used to check a given property. These features are valuable for testers, but they are not a complete solution. They provide statistics on the sample of data generated for a property as it has run. Reflective generators offer a more principled approach where the generator calculates properties of its own distribution.

When reflective generators are run backward, they can do more than just check whether a value could have been generated: they can also report the *probability* that a value was generated. This is done by interpreting a reflective generator using the type:

```
newtype UnGenProb t b a =
  UnGenProb {run :: (b, t -> Weight) -> Maybe (a, Probability)}
```

This looks a bit like `UnGenCheck`, but with a few extra bits and pieces. First, there is a new function in the argument: a mapping from choice tags to *weights*. Weights (integer values representing relative probability) are more common in QuickCheck generators than probabilities (see the `frequency` combinator), so the lookup function passed to `UnGenProb` returns values of type `Weight`. Second, the return value includes a `Probability` that tracks the likelihood that the generator produces a given value.

All of this works with the following `Pick` instance:

```
instance Pick UnGenProb where
  pick xs = UnGenProb (\(b, w) ->
    let totalWeight = sum (map (w . fst) xs)
        probLabeled = map (\(t, g) -> (w t / totalWeight, g)) xs
        filtered = mapMaybe (\(t, g) -> fmap (t,) (run g (b, w))) probLabeled
    in return (b, sum (map (\(p, (_, q)) -> p * q) filtered)))
```

The `pick` function does a bit of math to convert weights to probabilities, filters out sub-generators that cannot produce `b`, and then aggregates the nonzero probabilities together. That last step is really the key: the generator computes the joint probability of making each choice *and* generating `b`, which inductively computes the desired probability.

Happily, this complexity is hidden, and the user simply calls using the following function:

```
unGenProb
  :: (forall g. Reflective g => g t a a)
  -> (t -> Weight) -> a -> Probability
unGenProb x w a =
  case run x (a, w) of
    Nothing     -> 0
    Just (_, p) -> p
```

Like checking, this process fails if the generator cannot produce this value. In this case, a probability of `0` is returned; otherwise `UnGenProb` does its magic and `unGenProb` extracts the probability. Running this function with the uniform weighting on an example tree looks like:

```
> unGenProb (genBST (-10, 10)) (const 1) (Node Leaf 5 Leaf)
1 / 168
```

This particular tree has probability $\frac{1}{168}$ of being generated.

5 TEST SUITE ANALYSIS

The previous section shows how generators can validate properties internal to themselves, but what about external ones? If a test suite has already been generated or is provided by some external source, how might a tester evaluate its usefulness? Reflective generators can also help answering questions like these.

This section starts with some infrastructure, setting up abstractions and tools that are broadly useful, and then explores how this infrastructure can be used to analyse test suites.

5.1 Understanding Generator Choices

Like the previous section, this one uses the backward direction of a reflective generator to understand how that generator produces a value. But here the generator interpretations track annotations more

carefully, giving access to rich information about the generation process and enabling interesting new techniques.

Listing Choices. The annotations in a reflective generator like `genBST` mark the different choices that the generator might make. For example, `"node"` marks the choice to make a new `Node`, and `"5"` marks the choice of the value `5` in a `Node`. So, when generating the value `Node Leaf 5 Leaf`, the generator might make these choices:

```
["node", "5", "leaf", "leaf"]
```

This process of turning an unstructured string of choices into structured data looks a lot like *parsing* [4, 12], and it is well known that bidirectionalising a parser yields a pretty-printer [2, 13, 20, 26]. What does this mean for reflective generators?

The following function takes a reflective generator and a value and "pretty-prints" the list of choices that the generator makes when producing the value:

```
unGenList :: (forall g. Reflective g => g t a a) -> a -> [[t]]
unGenList x a = snd <$> run x a
```

Note that the return type is `[[t]]`, because there might be multiple ways of getting to a particular value.[3]

As with previous examples, `unGenList` is implemented using a data structure that is an instance of `Reflective`:

```
newtype UnGenList t b a = UnGenList {run :: b -> [(a, [t])]}
```

The `b` parameter represents the value that is run backward through the generator, and the `[(a, [t])]` is a list of potential outputs and the choices that lead to them. We implement `Pick` for this type like this:

```
instance Pick UnGenList where
  pick xs = UnGenList (\b -> do
    (t, g) <- xs
    (x, ts) <- run g b
    return (x, t : ts))
```

This implementation of `pick` runs each of the given sub-generators to get a list of values and the choices that produce them. Then the different possibilities are aggregated together with the most recent choice added to the front. All of this uses Haskell's list monad (also thought of as a nondeterminism monad) to keep track of the different possibilities at each step.

The implementation works as expected:

```
> unGenList (genBST (-10, 10)) (Node Leaf 5 Leaf)
[["node", "5", "leaf", "leaf"]]
```

[3]This is a design choice. It is also possible make `unGenList` return `Maybe [t]`, taking the "leftmost" choice sequence, or even `Maybe ([t], Probability)`, taking the most likely choice sequence. We choose the list-of-lists presentation for simplicity.

Generalizing the Choice Structure. Before looking at applications of `unGenList`, it will be useful to generalise. The choices that a generator makes have slightly more structure than a list, and we capture that structure with the help of a typeclass.

The `Monad` and `Pick` instances for `UnGenList` use three different operations to assemble choice lists: `[]`, `(++)`, and `(:)`. The first two are required for `Monad`, and the last for `Pick`. A suitable generalisation needs to capture all three of these operations. To that end, we introduce the following typeclass:

```
class Choices f where
  none  :: f a
  split :: Ord a => f a -> f a -> f a
  mark  :: Ord a =>   a -> f a -> f a

instance Choices [] where
  none  = []
  split = (++)
  mark  = (:)
```

Intuitively, `none` represents the case where no choices are made, `split` represents a branch in the tree with independent choices on each side, and `mark` actually marks a specific choice as being made. [4] The rest of the paper has plenty of examples of `Choice` structures, but for now it's enough to know that we can use this setup to generalise `UnGenList` to `UnGenChoices`:

```
newtype UnGenChoices c t b a =
  UnGenChoices {run :: (Ord t, Choices c) => b -> [(a, c t)]}

unGenChoices :: (Ord t, Choices c)
  => (forall g. Reflective g => g t b a) -> b -> [c t]
unGenChoices x b = fmap snd . ($ b) . run $ x
```

The inner list has been replaced by `c t` for any container `c` that is an instance of `Choices`. The instances are updated accordingly, replacing `[]` with `none`, `(++)` with `split`, and `(:)` with `mark`—everything else stays the same.

This more general interface is just as easy to use, and it specialises to the type of `unGenList` from before:

```
unGenList' :: Ord t => (forall g. Reflective g => g t b a) -> b -> [[t]]
unGenList' = unGenChoices
```

Next, we show how to use `unGenChoices` to analyse a suite of tests.

5.2 Evaluating Test Suite Coverage

Among the many ways that testers estimate how well a given set of test examples might catch bugs, *black-box* methods are some of the most versatile: they can be applied even if there is no way to access or instrument the internals of the system under test (e.g., because it is closed-source, etc.), since they rely only on properties of the test suite itself. This section describes two different black-box metrics and shows how reflective generators can help make those metrics easier to compute.

[4] The eagle-eyed reader might be tempted, as we were, to view this structure in the context of monoids. This is possible up to a point, but not all reasonable instances make `split` associative, so we found the connection less than helpful.

Choice Frequencies. One simple way to understand a test suite is measuring the frequencies with which the constructors of a datatype appears in a test suite. For BSTs, this means counting the relative frequencies of nodes and leaves, as well as measuring the distribution of values across a set of trees. These kinds of measurements are great coherence checks to make sure that there are no big gaps in a test suite. For example, it would be a problem to find that a suite of BSTs never tests a tree containing `0`.

For simple data types, there are ways to automatically tune a generator to ensure, for example, a uniform distribution over constructors [16], but in general it is not easy to know if a generator produces a reasonable distribution of constructor frequencies. In this area, we make two contributions: we refine the usual notion of constructor frequency to *choice* frequency, instead counting how often particular choices are made to produce a test suite, and we give an automatic way to compute choice frequencies for test suites, including those that were obtained from somewhere other than the reflective generator at hand.

Moving from constructor frequencies to choice frequencies is a subtle change, but a useful one. Consider a generator for de Bruijn-indexed terms of the simply typed lambda-calculus, which needs to select between available free variables:

```
pick [("", Var v) | v <- freeVars]
```

The labels on the variable indices are all the same, "", because the tester has chosen not to prefer one free variable over another and thus it would be counterproductive to make such granular distinctions. A choice frequency measurement ignores such details (simply giving a frequency for the empty tag, which can be ignored), whereas a naïve constructor frequency measurement does not. Critically, this is how the programmer would likely have written their reflective generator anyway, so these considerations are handled without extra effort.

Automatically computing choice frequencies is easy, given the machinery from the previous section. The type `Freqs` represents a map from choices to frequencies:

```
newtype Freqs a =
  Freqs {getFreqs :: Ord a => Map a Int}

instance Choices Freqs where
  none = Freqs Map.empty
  split (Freqs x) (Freqs y) = Freqs (Map.unionWith (+) x y)
  mark x (Freqs xs) = Freqs (Map.insertWith (+) x 1 xs)

unGenFreqs :: Ord t => (forall g. Reflective g => g t a a)
                    -> a -> [Freqs t]
unGenFreqs = unGenChoices
```

These instances succinctly describe how the frequencies should be tracked and combined through as the generator runs backward, and `unGenChoices` can be used to extract the `Freqs` for a value without any extra code. Those frequencies can be aggregated across a test suite to get a more global picture. Here it is in action:

```
> unGenFreqs (genBST (-10, 10)) (Node Leaf 5 Leaf)
[Freqs {getFreqs = fromList [("leaf", 2), ("node", 1), ("5", 1)]}]
```

Combinatorial Coverage. Moving beyond choice counting, reflective generators can also measure more nuanced test suite metrics like *combinatorial coverage*. Combinatorial coverage is classically used to improve testing for systems with large numbers of simple configuration parameters, asking the question "how well does a particular set of configurations exercise the system"?

Suppose a system has four configuration parameters, `a` through `d`, and consider a particular configuration:

```
config = { a = True, b = False, c = True, d = False }
```

The key insight behind combinatorial testing is that this single configuration exercises interactions between all pairs of parameters. For example, testing with `config` shows how the system behaves when

```
a == True && b == False
```

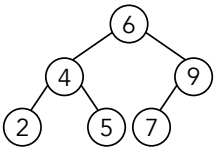
and it also shows behaviours that happen when

```
c == True && d == False
```

etc. If any of these specific interactions causes a bug on its own, testing under `config` will make that issue clear. In this way, a single test exercises a combinatorial explosion of potentially buggy interactions between subsets of parameters. We say that `config` covers the 2-way description `a == True && b == False`.

Measuring combinatorial coverage gives concrete guarantees about testing efficacy. Knowing a test suite covers all t-way descriptions means that no bug in the system can depend on fewer than $t + 1$ parameters simultaneously—the tester can categorically rule out certain shapes of bugs.

This simple notion of t-way description does not apply directly to algebraic data types because it does not take structure into account. But a suitable generalization has been proposed [3]. Instead of looking at relationships between configuration values, this notion of coverage counts spatial relationships between the data constructors in the type. For example, a BST like the one below covers the 2-way description to "4 to the left of 6" and also "2 to the left of 6".



The QuickCover library computes this generalised coverage metric by converting values into a tree of constructors:

```
data Term a = Term a (Cons a)
newtype Cons a = Cons [Term a]
```

The `Term` type is a rose tree of constructors that captures the rough structure of the value. For example, the tree for `Node Leaf 5 Leaf` should look like:

```
Term "node" ( Cons [ Term "5" (Cons []),
                     Term "leaf" (Cons []),
                     Term "leaf" (Cons [])] )
```

Computing an appropriate `Term` from a value is conceptually simple to do, but it is tedious and requires some non-trivial choices about which constructors should "count" towards coverage. This hassle can be avoided with the help of reflective generators and a few lines of code:

```
instance Choices Cons where
  none = Cons []
  split (Cons xs) (Cons ys) = Cons (xs ++ ys)
  mark x xs = Cons [Term x xs]
```

The `Choices` instance for `Term` builds the desired tree: every `pick` creates a new root of the constructor tree, and every `(>>=)` concatenates constructors at the same level of the tree. As a simple example:

```
> unGenChoices (genBST (-10, 10)) (Node (Node Leaf 4 Leaf) 5 Leaf)
["node"("5", "node"("4", "leaf", "leaf"), "leaf")]
```

This all *just works*. The type of `unGenChoices` specialises to the desired type, and the tree is computed automatically. Even better, this is a tree of choices, not a tree of constructors, so it has the same advantages that choice frequencies have over constructor frequencies. In particular, the natural tagging scheme for reflective generators appropriately ignores the impact of uninteresting constructors (like variable indices) in the tree.

To wrap up the story, a `Term` can be passed to a function that computes the list of descriptions covered by that `Term`:

```
covers :: Ord a => Int -> Term a -> [Description a]
```

The `Description` type here represents the "X to the left of Y" descriptions discussed above. Running this analysis for a suite of BSTs might highlight that trees tend to lean too far to the left or right, that consecutive values appear together infrequently, and other potentially concerning systematic issues. It might also give the tester confidence that bugs do not appear in trees with particular assortments of values (for example, small ranges of values appearing together). All of these are valuable measurements that come included with every reflective generator.

6 DYNAMIC TESTING STRATEGIES

The applications so far have mostly focused on the backward direction of reflective generators, but there are novel interpretations for the forward direction as well. Even better, these synergise with the information gained by the backward direction in helpful ways. This section shows that the two directions of a reflective generator can be used together to replicate and generalise state-of-the-art "dynamic" strategies for random testing—strategies that dynamically update their testing strategy based on feedback from the testing process. Here we see the true power of reflection: knowledge about past choices can influence the way we make future ones.

6.1 Example-Based Generation

Testers often have good intuitions about what regions of their input space are most interesting—this is why unit testing is so effective! However, in traditional unit testing, it is always possible that the next example is the one that will catch the bug, and unlike in random testing it takes programmer effort to get more. Example-based testing addresses this problem by combining tester expertise with the power of random testing. Testers provide example inputs that are used to guide the generation of further random test inputs, enabling more thorough exploration of "interesting" parts of a system's input space. A fantastic example of this pattern, which we riff on here, is *Inputs from Hell* [24].

Consider testing the following expression language from that paper:

```
data Expr = Term Term | Plus Expr Term | Minus Expr Term
data Term = Factor Factor | Times Term Factor | Div Term Factor
data Factor = Digits Digits | Pos Factor | Neg Factor | Parens Expr
```

```
data Digits = Digit Char | More Char Digits

exampleInputs = ["(2*3)", "1*(5)", "((3+4))" ...
```

The language supports a wide variety of arithmetic expressions, but perhaps a tester wants to thoroughly test the system's behaviour on just a few forms of expression (represented by `exampleInputs`). Or, a tester might want to explore inputs that are very *different* from some set of given use-cases. Example-based generator tuning can handle both of these testing strategies, generating similar (or very different) inputs by drawing from a probability distribution describing a given set of examples (or the complement of this distribution):

```
> commonInputs exampleInputs
["4*5", "4", "(3)", "2*(((5))+4)" ...
> uncommonInputs exampleInputs
["7-9-(8)-56/9+87", "4-8-2*7", "(7)" ...
```

Exampled-Based Generation with Reflective generators. The process of generating common and uncommon inputs from examples works like this:

(1) Extract choice weights from a set of examples.
(2) If "uncommon" examples are desired, invert the weights.
(3) Use the weights to guide the generation of new test values.

Implementing this process concretely requires a generator that can generate based on an assignments of weights to choice tags (similar to the weights we saw in §4 with `UnGenProb`):

```
newtype WeightedGen t b a =
  WeightedGen {run :: (t -> Int) -> Gen a}

weightedGen :: (forall g. Reflective g => g t a a) -> (t -> Int) -> Gen a
weightedGen = run
```

The weight map is represented by a function from the semantic tags of the generator to weights. For example, `genBST` can be biased to always make leaves:

```
alwaysLeaves = weightedGen (genBST (-10,10))
                           (\t -> if t == "leaf" then 1 else 0)

> QC.sample alwaysLeaves
Leaf
Leaf
...
```

The backward counterpart of `WeightGen` has already come up: it is `UnGenFrequencies` from §5. The `unGenWeights` function is just like `UnGenFrequencies`, but it only cares about a single set of weights extracted from a given value (the `listToMaybe` does this in a type-safe manner, wrapping the result in a `Maybe`):

```
unGenWeights :: Ord t => (forall g. Reflective g => g t a a)
                      -> a -> Maybe (Freqs t)
unGenWeights x b = listToMaybe (unGenChoices x b)
```

Using `Freqs` here means that the weight map is now represented as a key-value map. This means that the weights can be adjusted programmatically, e.g., making inversion possible when generating uncommon examples. Using functions in the other direction was a design choice that improves user experience because, with functions, users have access to wildcard pattern matches and utilities like `const`. The key-value weight maps can be converted to function weight maps for generation via the following function:

```
weightMap :: Ord t => Freqs t -> (t -> Int)
weightMap m t = fromMaybe 0 . Map.lookup t . getFreqs $ m
```

Together, `weightedGen` and `unGenWeights` can perform example-based testing. They can be used to test types such as `Expr` if a reflective generator is made for each of these types. As an example, here is the one for terms:

```
genTerm :: (Reflective g) => Int -> g String Term Term
genTerm 0 = (Factor <$> genFactor 0) `at` _Factor
genTerm n =
  pick
    [ ("factor", (Factor <$> genFactor (n - 1)) `at` _Factor),
      ("times", do
          x <- genTerm (n - 1) `at` (_Times . _1)
          y <- genFactor (n - 1) `at` (_Times . _2)
          pure (Times x y)),
      ("div", do
          x <- genTerm (n - 1) `at` (_Div . _1)
          y <- genFactor (n - 1) `at` (_Div . _2)
          pure (Div x y)) ]
```

To ensure termination, `genTerm` takes an `Int` argument that is decremented on subsequent calls and used to stop generation early. For simplicity, the semantic tags are just strings that represent the choices made e.g., `"times"` for when the `Times` constructor is selected.

When the size argument is nonzero, `genTerm` makes a choice, using `pick`, between its three constructors. Automatically derived prisms (`_Factor`, `_Times`, and `_Div`), are used to annotate sub-generators for backward interpretation.

To demonstrate that these reflective generators can be used to generate common and uncommon inputs, consider this example input: `i = parse "1*(2+3)"`, where `parse` is a parser for the `Expr` type and `print` is the corresponding printer. Then `unGenWeights` can run backward to get the weights that lead to it:

```
iWeights :: Maybe (Freqs String)
iWeights = unGenWeights (genExpr 4) i


> fmap (M.toList . getFrequencies) iWeights
Just [("1",1),("2",1),("3",1),("digits",3)...
```

As expected, the weight map features representative occurrences of 1, 2, and 3, but no other numbers. (There are three instances of `"digits"` because that appears three times in the corresponding parse tree.) Values generated from this weight map using `weightedGen` can be expected to only use these numbers, following the lead of `i`:

```
iReplicate :: Gen Expr
iReplicate = weightedGen (genExpr 4) (weightMap . fromJust $ iWeights)

> QC.sample (fmap print iReplicate)
"1+3"
"1"
"3*1+(1)*(2+1)"
```

These common inputs are clearly derived from `i`. Success! Uncommon inputs are also achievable by inverting the weights in `iWeights`, and the weights of multiple examples can be amalgamated using the following function, which uses the semigroup operation to sum together the weights of tags from the different examples:

```
mine :: Ord t => (forall g. Reflective g => g t a a) -> [a] -> Freqs t
mine g = foldr (split . fromJust . unGenWeights g) none
```

Thus example-based testing can be implemented using reflective generators. Moreover, because they are monadic, they can operate on a broader scope of inputs than prior work on example-based testing [24], as we show next.

Example-Based Testing with Validity Conditions. Now this is the exciting part. Up until now reflective generators have simply replicated the results of Soremekun et al., but they can actually tackle more complicated examples. For example, what if the generated expressions should be restricted to avoid divide-by-zero errors? The `genTerm` reflective generator from above can be modified in to do just that!

```
genTerm :: (Reflective g) => Int -> g String Term Term
genTerm 0 = (Factor <$> genFactor 0) `at` _Factor
genTerm n =
    pick
      [ ("factor", (Factor <$> genFactor (n - 1)) `at` _Factor),
        ("times", do
            x <- genTerm (n - 1) `at` (_Times . _1)
            y <- genFactor (n - 1) `at` (_Times . _2)
            pure (Times x y)),
        ( "div", do
            x <- genTerm (n - 1) `at` (_Div . _1)
            y <- genFactor (n - 1) `at` (_Div . _2)
            pure $ case evalFactor y of
              Just v | v /= 0 -> Div x y
              _ -> x ) ]
```

Now the generation of `Div` terms deals with each argument to the division separately, and uses an evaluator of `Expr` terms to ensure that the divisor is never zero.

Critically, this *would not* have been possible with the grammar-based generators in *Inputs from Hell*. Grammar-based generators are powerful, but they cannot execute arbitrary computation during generation. The monadic approach of reflective generators has allowed us to expand the horizons of what example-based testing can be applied by inviting monadic validity conditions to the party!

6.2 Adaptive Fuzzing

Another dynamic testing strategy that reflective generators can help with is adaptive fuzzing. Adaptive fuzzers find and exploit interesting inputs via random mutation. Rather than generate fresh test inputs over and over, the fuzzer keeps a pool of "seed" values that it deems interesting and mutates those values to try to find other interesting values that are conceptually "close".

Traditional fuzzers mutate values by flipping or swapping random bits—they generally operate on unstructured binary data—but more modern fuzzers such as FuzzChick [10] bring adaptive fuzzing to functional programming with the help of structured mutation. FuzzChick mutates seed values at a much higher level of abstraction than bit flips: it might, for example, replace one value or constructor with another, shrink a tree to one of its subtrees, or swap the children of a constructor. These mutations can be synthesized from type information, which works well for many use-cases, but what if the random values need to satisfy some validity condition? FuzzChick can't do it, but reflective generators can!

Mutating with Trees. As with the previous example, doing validity-preserving mutation requires two interpretations of the same reflective generator, one forward and one backward. At a high level, the process goes like this:

(1) Take a seed value and run it backward through the generator to get the choices that led to it.
(2) Randomly mutate the *choices*.
(3) Run the new choices forward through the generator to get a new value.

By construction, this process is guaranteed to produce a value that preserves the generator's validity condition, because the mutated value comes from running the generator forward.

The most naïve implementation of this process does not actually quite work. For example, suppose choices are extracted using `unGenList`, and a mutation consists of simply replacing one choice in the list for another. The term `Node (Node Leaf 2 Leaf) 5 (Node Leaf 7 Leaf)` run backward through `genBST` yields these choices:

```
["node", "5", "node", "2", "leaf", "leaf", "node", "7", "leaf", "leaf"]
```

Those might be mutated to this:

```
["node", "5", "leaf", "2", "leaf", "leaf", "node", "7", "leaf", "leaf"]
```

Strangely, even though the change is ostensibly to the left side of the tree, a conservative generator (that discards choices it cannot use) interprets these choices as

```
["node", "5", "leaf", ..., "leaf", ...]
```

and removes *both* children to produce `Node Leaf 5 Leaf`. The problem here is that working with lists of choices is too unstructured.

The solution is one final instance of the `Choices` class:

```
data FChoices a = None
                | Split (FChoices a) (FChoices a)
                | Mark a (FChoices a)
```

This structure is a bit different from previous instances: instead of implementing some semantically interesting version of each required operation, it simply acts as a syntax tree of `Choices` operations. As the name suggests, it is essentially the "free" instance of `Choices`.

```
instance Choices FChoices where
  none  = None
```

```
  split = Split
  mark  = Mark
```

Calling `unGenChoices` with the `Choices` container specialised to `FChoices` yields `FChoices` trees that mimic the structure of the generator that produced them. These are perfect for mutation, because the tree-like structure enables targeted mutations that only affect the part of the tree that they intend to. This infrastructure can be used to implement a rich library of validity-preserving mutations, as we will see next.

Manipulating Choices. In order to do step (2) above, functions that can mutate choice trees are needed. These functions are inspired by the kinds of mutations that are done in libraries like FuzzChick.

One way to mutate a value is to simply change some constructor in its syntax tree. In the case of a BST, this includes modifying a node value (`5` to `8`), lopping off a branch of the tree (`Node` to `Leaf`), and growing the tree by one node (`Leaf` to `Node`). These kinds of mutations can be accomplished by re-making one of the choices in the choice tree, leaving the rest of the choices intact. This is done by a function called `rerollMut`:

```
data MayReroll a = Keep a | Reroll a
rerollMut :: FChoices a -> Gen (FChoices (MayReroll a))
```

The `MayReroll` type represents two possibilities: keep the choice that was made, or avoid a choice and "re-roll" it. The `rerollMut` function randomly picks one choice in the tree to wrap in `Reroll`; the rest are wrapped in `Keep`.

Another mutation is a form of shrinking: replace a value with some sub-value. For BSTs this might mean replacing `Node (Node Leaf 4 Leaf) 5 Leaf` with `Node Leaf 4 Leaf`. But trying to make this kind of mutation on the tree of choices leaves us with a question: which choices "count" as reasonable sub-values? It would be silly to replace a tree starting with `Mark "node" _` with one that looks like `Mark "5" _` as there would be no way to use that new tree to reconstruct an appropriate value. It might be possible to solve this problem with some heuristics, but it is simple enough to just ask the programmer for help in the form of a compatibility relation on tags:

```
type Compat a = a -> a -> Bool
```

There are a couple of restrictions that make a good compatibility relation. First, it should be an equivalence relation; the induced equivalence classes are the sets of tags that are mutually interchangeable. Second, the relation must respect types. If two choices are used to produce values of different types, then their tags must not be compatible. Note that the converse is not true: the compatibility relation is free to distinguish tags that produce the same type. For example, tag equality is a perfectly good compatibility relation.

Given an appropriate compatibility relation, a mutator can be defined that shrinks a value to one of its sub-values:

```
shrinkMut :: Compat a -> FChoices a -> Gen (FChoices a)
```

This function walks down the tree to the first `Mark` choice and then replaces that choice with a subtree of choices whose root is compatible with the original. For example, if compatibility is `(==)`, `shrinkMut` would mutate a BST by replacing the whole choice tree with a subtree with `"node"` at the top. As long as the compatibility relation is chosen appropriately, this will result in a nicely formed tree of choices that will reconstruct part of the original value.

Finally, a value can also be mutated by swapping two of its sub-values, for example turning `Node (Node Leaf 4 Leaf) 5 Leaf` into `Node Leaf 5 (Node Leaf 4 Leaf)` (by swapping the

left Node with the right Leaf). This is not a desirable mutation for BSTs, as any swaps break the validity condition, but is useful for other types, and the validity preserving power that reflective generators bring to the table (and that will be discussed soon) ensures that such mutations are rendered harmless for types like BSTs.

```
swapMut :: Compat a -> FChoices a -> Gen (FChoices a)
```

Like shrinkMut, this function walks the choice tree, but rather select one sub-tree compatible with the root, it selects two that are compatible with each other and interchanges them. The same compatibility relation can be used for both shrinkMut and swapMut.

Regenerating Values. Once the choices have been mutated, they can be turned back into a value (step (3) above). Often this process is a straightforward walk down the choice tree, making the choices as instructed, but when choices need remade, there is more work to be done. The MutGen type is the forward interpretation that takes care of this. It takes a choice trees created by the backward interpretation and uses it to create a mutated value:

```
newtype MutGen t b a =
  MutGen {run :: FChoices (MayReroll t) -> Gen a}

mutGen :: Eq t => MutGen t a a -> FChoices (MayReroll t) -> Gen a
mutGen = run
```

The mutGen function gives results in the Gen monad because of the choices it may still need to make; recall that rerollMut does not actually make a new choice, it simply marks the old choice as invalid. The Pick instance for MutGen is worth exploring, as it makes a few assumptions:

```
instance Pick MutGen where
  pick gs = MutGen $ \case
    Mark (Keep c) rest -> -- Normal case: make the given choice
      case find ((== c) . fst) gs of
        Nothing -> arb gs rest -- Choice invalid: pick a random valid one
        Just (_, g) -> run g rest
    Mark (Reroll c) rest -> -- Dropped choice: avoid that choice
      case filter ((/= c) . fst) gs of
        [] -> arb gs rest -- The discarded choice was the only valid one: pick it
        gs' -> arb gs' rest
    _ -> run (snd (head gs)) None -- The tree is out of sync: generate smallest
    where
      arb gens rest = QC.elements (snd <$> gens) >>= \g -> run g rest
```

The pick function takes it cue from the top-level constructor of the choice tree. In the first case, the generator simply makes the choice it is told to, as long as that choice is valid. If the choice is invalid for some reason, a random valid choice is chosen instead. In the second case, the generator has explicitly been instructed not to choose a particular choice, so it randomly chooses a different one (if it can). Finally, if the top-level constructor is not an Mark at all, something must have gone wrong so the generator greedily makes the first choice available to it.

This final behaviour of pick makes one strong assumption about the way the reflective generator is written: it assumes that the first choice in any given pick is not recursive. This is a somewhat unfortunate requirement, but it greatly improves the mutation behaviour. It means that changing a Leaf to a Node in a BST will add just that one node instead of some arbitrary random subtree. The

non-recursive requirement is also consistent with many QuickCheck generators in the wild, so in practice we do not expect this to be a large problem. Regardless, this assumption can be relaxed with a slightly modified version of `MutGen`, if needed.

Putting everything together, a single function, `mutate`, can be defined that uses a reflective generator, a value in its range, and a compatibility relation on labels to produce a random generator of mutated values:

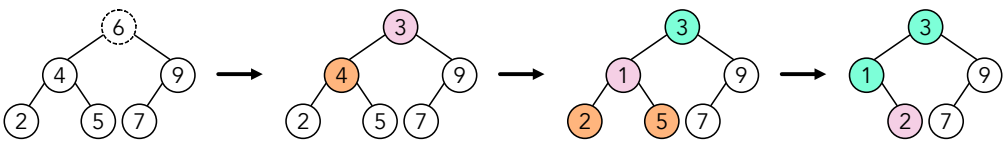
```
mutate :: Ord t => (forall g. Reflective g => g t a a)
                -> (t -> t -> Bool) -> a -> Gen a
mutate g eq x = QC.elements (unGenChoices g x)
                >>= (manipulate >=> mutGen g)
  where
    manipulate c =
      QC.oneof
        [ rerollMut c,
          fmap Keep <$> swapMut eq c,
          fmap Keep <$> shrinkMut eq c ]
```

This is exactly the algorithm from the beginning of the section.

(1) Extract the choices by running the generator backward with the `UnGenChoices` interpretation.
(2) Mutate the choices with `manipulate`, which combines the mutation strategies discussed above.
(3) Re-generate a value by running the generator forward with the `MutGen` interpretation.

Importantly, the compatibility relation can be set to `(==)` as long as the tester does not have specific mutation requirements, so this all works essentially "for free" when an appropriate reflective generator is available.

Despite its simplicity, this setup is quite powerful. Consider the following mutation of a BST:



The root node of the tree is chosen for mutation and marked with `Reroll`, so the `MutGen` interpretation of `genBST (1, 9)` starts by changing the root node's value to another one that is not 6, in this case 3. But there is a problem: 4 is no longer an appropriate value for the left child. To fix this, the mutator makes a random valid choice to replace the 4; the new value of 1 is valid child, but now its children are invalid. Since 1 is the smallest valid in the tree, there is no choice but to remove the left node entirely. Finally, the only value available for the right node is 2. Phew! That was complicated…

Amazingly, reflective generators can do this all automatically! Validity-preserving mutation is subtle—it's easy to do incorrectly even with a bespoke mutation function—but the entire procedure above was carried out by the same `genBST` generator that we have been working with since §3, together with a bunch of entirely generic library code. To our knowledge, generic validity-preserving mutation like this has never been achieved before.

7 LIMITATIONS

The reflective generator abstraction does have a few shortcomings compared to the one available in vanilla QuickCheck.

First, reflective generators do not currently handle the hidden size parameter that is baked into the normal generator abstraction. QuickCheck has a function:

```
sized :: (Int -> Gen a) -> Gen a
```

which allows testers to parametrise their generator by a size. The main advantage, is that building this into the framework allows the test runner to vary the size dynamically during testing. Adding size information to reflective generators should be unproblematic.

Second, the API that we chose for `pick` makes some concessions around manually weighted generators. The `WeightedGen` defined in §6 covers most of the functionality of QuickCheck's `frequency` by assigning weights to choice labels, but there is one pattern that `WeightedGen` fails to capture:

```
frequency [(1, foo), (n, bar)]
```

If `n` is not constant, assigning an external weight is insufficient: there is no way to replicate the computation that produced `n` within the weighting function provided to `weightedGen`. We believe there are workarounds within the framework as we present it (e.g., adding computed weight information to a generator's tag type), but the most robust solution would be to provide a slightly less elegant version of `pick` that takes weight information alongside tags. Again, this can be done fairly easily if needed.

Finally, a more fundamental limitation of reflective generators is the dependency on bidirectionalisation: some generators simply cannot be made bidirectional for fundamental computational reasons. For example, a generator might do things like compute a hash that is computationally hard to reverse or generate data and throw it away entirely (making it impossible to replicate those choices backward). This is, of course, an unavoidable limitation.

However, many traditionally "non-invertible" programs are perfectly fine in reflective generators. Functions with partial inverses are handled automatically by `ProfunctorPartial`, and non-injective functions can be made to work by choosing a canonical inverse. In general, we expect that only generators with very complex control flow (e.g., ones that require higher-order functions) and pathological generators like the ones mentioned above will pose problems in practice. Indeed, many complex generators (including one for well-typed System F terms) are straightforward to bidirectionalise.

8 RELATED WORK

Several threads of prior work intersect with ours.

Test Data Producers. Our work depends most directly on QuickCheck [1], but it also has connections to other domain specific languages that produce test data. For example, SmallCheck uses enumeration, rather than random generation, to obtain test inputs for property-based testing [22]. Reflective generators can be made into enumerators quite simply, by using

```
newtype Enum t b a = Enum {run :: [a]}
```

as the interpretation; this is one more way that reflective generators help to unify existing ideas in the testing space. Separately, reflective generators are related to *free generators* [4], in that both derive value from labelling generator choices.

Bidirectional Programming. Our discussion has been focussed on random generation, but it is worth taking a step back and considering other domains where bidirectional programming has been applied and what reflective generators might bring to the table.

Pickling is a bidirectional process whereby values are packed and unpacked for transmission across a network. Kennedy present a bidirectional interface predating the monadic profunctors interface we build on [26], that also encapsulates notions of compositionality and choice. We are also not the first to take a "classy" approach to bidirectionalisation. [21] uses type classes to create a unified interface for describing syntax that can either be interpreted as a parser or a printer. Both of these examples are quite compatible with reflective generators—either could be implemented by a reflective generator if there was a compelling testing use-case.

Exposing Generator Choices. The Hypothesis [12] testing library in Python views generators as processes that turn a stream of random bit-flips into a data structure. Operations like shrinking are done by shrinking the underlying choices and then regenerating, in much the same way as we do with `shrinkMut`. There are two main differences to highlight. First, Hypothesis cannot shrink a value once it has discarded the choices that produce that value (e.g., if the user wants to manipulate the value outside of the Hypothesis test-runner); reflective generators are (to the best of our knowledge) the only abstraction currently able to handle those kinds of cases and recover the choices. Second, as we highlight in §6, accurate mutation requires structured choices, which are not something they use. This is not a fundamental limitation of Hypothesis, but we do not know of anyone who has explored structured choices in that context.

RLCheck [19] uses reinforcement learning to guide a QuickCheck-style generator by externally biasing labelled choice points. Fundamentally, this choice labelling is similar to the way choices are highlighted in this work. Thus, in principle, a similar approach might work for reflective generators, but it would likely require a more restricted API for choices than the one we use.

Generator Validation. In addition to its domain-specific languages for generators and properties, QuickCheck [1] provides useful tools for understanding a generator's distribution. Users can annotate their properties to track the proportion of tests of a certain shape or size, giving important feedback about testing efficacy. Reflective generators, particularly those proposed in §4, complement these tools with directly computed distributional measurements and an automatic technique for understanding generator completeness.

Test Suite Analysis. Sometimes test-coverage goals can be ensured by construction. Mista et al. [16] provide automated methods for deriving generators that are guaranteed to have a good constructor distribution. This is certainly the right approach if the required test data is either unconstrained or if the constraints are not particularly strict, and if there is no need to analyse previously run tests. Outside of that situation, techniques for producing the right distribution automatically are harder to find, so the analysis provided by reflective generators would be helpful.

In a similar vein, QuickCover [3] is uses a novel combinatorial coverage metric to automatically "thin" a generator's distribution and pick out the most useful tests. Reflective generators are a nice addition to this workflow: they can be used to automatically compute the coverage information used by the algorithm, reducing glue code.

Dynamic Testing Strategies. The Mutagen [14] library is a good point of comparison for the mutation library provided in §6. While Mutagen does not handle validity-preserving mutation, it does have carefully tuned heuristics for mutating trees generically (accomplished via metaprogramming rather than a generic data structure like `FChoices`). The heuristics applied by our `mutate` function could likely be improved by following Mutagen's lead.

9 CONCLUSIONS AND FUTURE WORK

Reflective generators They provide a unified solution to challenges in generator validation, test suite analysis, and dynamic testing strategies. Using this framework, testers can annotate their QuickCheck-style generators with the appropriate backward maps and semantic tags to gain access to a wealth of powerful testing tools that go far beyond plain random generation.

There are a number of potential directions for future work.

Algebraic Effects. An astute reader may have noticed that interpretations of reflective generators could be provided as monad transformers [11]. This is a presentation that we have avoided for simplicity, but a connection that should be investigated. Incorporating monad transformers, or alternatively algebraic effects [8, 17, 18, 23], unlocks a wealth of literature, which could enable new interpretations.

Automating Shrinking. There are a couple of interesting applications in shrinking that have not yet been explored. First, it seems likely that tools like `shrinkMut` and `MutGen` in §6 could be useful for automatically shrinking certain data types. It would go almost like mutation—extract choices, shrink those choices, and then regenerate—but extra care would need to be taken to ensure that the resulting values are really shrinks and not unrelated values.

Building on this idea, reflective generators might also improve shrinking outside of Haskell. The Hypothesis library in Python already does shrinking automatically by remembering the generator choices that produced a value and shrinking those choices. But what happens if a value is generated, *manipulated* in some way, and then used as a test? The choices get lost and there is no good way to shrink. Reflective generators would give a way to retrieve choices from any value in the range of the generator, regardless of how disconnected that value is from the original generation process.

Learning from Choices. The guided testing example in §6 illustrates one way that reflective generators can be used for a kind of "learning". In that case, example inputs are treated as training data to learn a simple model of choice weights. But this form of learning is quite simple. Is it possible to combine reflective generators with more sophisticated learning algorithms?

One potential path forward is use reflective generators to train and interpret *language models*. The process would go something like this:

(1) Run a reflective generator backward to extract choice information from a dataset of desirable inputs.
(2) Train a language model to produce similar choices.
(3) Sample new choices from the model.
(4) Run the reflective generator forward to turn those choices into new data structures that can be used for testing.

The main advantage this has over the approach in §6 is expressiveness. Choice weights cannot express relationships between different parts of a data structure, and thus may not capture certain aspects of the examples. In contrast, an appropriate language model could learn to generate values that really capture essence of the examples.

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] John Nathan Foster. 2009. *Bidirectional programming languages.* Ph. D. Dissertation. University of Pennsylvania.

[3] Harrison Goldstein, John Hughes, Leonidas Lampropoulos, and Benjamin C. Pierce. 2021. Do Judge a Test by its Cover. In *Programming Languages and Systems: 30th European Symposium on Programming, ESOP 2021, Part of the European*

Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021 (Lecture Notes in Computer Science, Vol. 12648). 264–291. https://link.springer.com/chapter/10.1007%2F978-3-030-72019-3_10

[4] Harrison Goldstein and Benjamin C. Pierce. 2022. Parsing Randomness: Unifying and Differentiating Parsers and Random Generators. arXiv:2203.00652 [cs.PL]

[5] John Hughes. 2007. QuickCheck testing for fun and profit. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 1–32.

[6] Andrew Kennedy. 2004. Functional Pearl: Pickler Combinators. *Journal of Functional Programming* 14 (January 2004), 727–739. https://www.microsoft.com/en-us/research/publication/functional-pearl-pickler-combinators/

[7] Oleg Kiselyov. 2012. Typed tagless final interpreters. In *Generic and Indexed Programming*. Springer, 130–174.

[8] Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible Effects: An Alternative to Monad Transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell* (Boston, Massachusetts, USA) *(Haskell '13)*. Association for Computing Machinery, New York, NY, USA, 59–70. https://doi.org/10.1145/2503778.2503791

[9] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. 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*. 114–129. http://dl.acm.org/citation.cfm?id=3009868

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

[11] Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 333–343. https://doi.org/10.1145/199448.199528

[12] David R MacIver, Zac Hatfield-Dodds, et al. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891.

[13] Kazutaka Matsuda and Meng Wang. 2018. FliPpr: A System for Deriving Parsers from Pretty-Printers. *New Generation Computing* 36, 3 (2018), 173–202. https://doi.org/10.1007/s00354-018-0033-7

[14] Agustín Mista. 2021. MUTAGEN: Faster Mutation-Based Random Testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 120–122.

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

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

[17] Gordon Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11 (02 2003), 69–94. https://doi.org/10.1023/A:1023064908962

[18] Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–94.

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

[20] Tillmann Rendel and Klaus Ostermann. 2010. Invertible syntax descriptions: unifying parsing and pretty printing. *ACM Sigplan Notices* 45, 11 (2010), 1–12.

[21] Tillmann Rendel and Klaus Ostermann. 2010. Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing. In *Proceedings of the Third ACM Haskell Symposium on Haskell* (Baltimore, Maryland, USA) *(Haskell '10)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1863523.1863525

[22] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, Andy Gill (Ed.). ACM, 37–48. https://doi.org/10.1145/1411286.1411292

[23] Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. 2019. Monad Transformers and Modular Algebraic Effects: What Binds Them Together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell* (Berlin, Germany) *(Haskell 2019)*. Association for Computing Machinery, New York, NY, USA, 98–113. https://doi.org/10.1145/3331545.3342595

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

[25] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. *SIGPLAN Not.* 49, 12 (sep 2014), 1–12. https://doi.org/10.1145/2775050.2633358

[26] Li-yao Xia, Dominic Orchard, and Meng Wang. 2019. Composing bidirectional programs monadically. In *European Symposium on Programming*. Springer, 147–175.