# Holding a Mirror up to 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 generators in frameworks like QuickCheck [1] are designed to generate values that satisfy application-specific validity conditions, while optimising distributional properties of the generation. Their careful construction makes them good at their primary job of generating random data, but can they do more?

We propose a new generator abstraction and an accompanying combinator library. Built on the idea of *bigenerators* [21], these *reflective generators*—named for the bidirectional nature of reflection and for the way they reflect on their choices—facilitate improvements at multiple points in the testing process. First, they allow powerful introspection that helps programmers *validate* generators as they write them. Second, reflective generators 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, reflective generators provide infrastructure for *implementing and generalising* state-of-the-art dynamic testing strategies [8, 20]; in particular, they enable a novel approach to invariant-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 programs are mainly used to produce random values, functionality that is most commonly associated with property-based testing (PBT), but with a little additional effort, they can do so much more.

Prior work has already demonstrated how tweaking the generator abstraction can lead to richer functionality. *Bigenerators* [21], which are written in an annotated version of QuickCheck's monadic generator language, act as normal generators that produce random values when run "forward". However, they can also 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. This is a clever idea, but it is only the tip of the iceberg.

We present *reflective generators*, a further refinement of QuickCheck's domain-specific language that builds on 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 philosophical reflection: the choices that the generator makes to produce an input are available for analysis (and even external control). Reflective generator provide unified solutions to problems throughout the testing process. In particular, they provide tools to improve generator validation, test suite analysis, and strategies for dynamic testing.

*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 complicated enough to disobey this guidance. Such generators can produce invalid values or, more insidiously, can miss important parts of the valid space of inputs. Relatedly, random generators induce a distribution over test values, and it can be frustratingly difficult to achieve the right one. Our framework addresses these concerns and gives programmers new ways to understand the behaviour of their generators.

*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 constructor of a sum type 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 presented 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 often computed in an ad-hoc way for each data type, but our framework provides these analyses via the information already available in the generator.

*Dynamic Testing Strategies.* While vanilla QuickCheck works for many properties, it is sometimes necessary to use more guided or adaptive strategies to find bug-provoking inputs. This is where reflective generators shine brightest. By leveraging all of the power they bring to bear, reflective generators generalise core ideas behind two state-of-the-art testing approaches. In world of example-based tuning, reflective generators replicate and expand on results from *Inputs from Hell* [20], lifting a useful testing technique from grammar-based testing to much more powerful monadic generation. In the world of adaptive fuzzing (a la FuzzChick [8]), reflective generators provide a novel and generic way to mutate values while preserving tricky validity conditions.

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

- We enrich bigenerators by annotating them with semantic choice labels, creating *reflective generators* (§3).
- We present novel ways to use our generators for introspection: reflective generators validate their own correctness and distributional properties, giving insights that would otherwise require tedious manual calculations (§4).
- We show that our new reflective generators can perform black-box analyses on par with the state of the art [3] (§5).
- We use reflective generators backward and forward to generalise an existing approach to example guided testing [20] and establish a novel technique for validity-preserving mutation that enables better adaptive fuzzing [8] (§6).

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

## 2 BACKGROUND

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

### 2.1 QuickCheck Generators

When it comes to property-based testing, the binary search tree (BST) is a classic example. Consider this `Tree` datatype and validity function, `isBST`, that decides whether or not a `Tree` is a BST:

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

The following QuickCheck property can check that an `insert` function upholds BST invariants:

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

During testing, 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 enough valid BSTs to pass the `isBST t` check is low, making the property vacuously true, and the testing insincere. Luckily, libraries like QuickCheck provide means for creating better generators that can, for example, enforce the `isBST t` predicate 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 example QuickCheck generator is written in monadic style using QuickCheck combinators. Its definition revolves around the desired invariant, 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 recursive call for child trees, and the combinator `chooseInt` for selecting its value.

## 2.2 Bigenerators

Generators like `genBST`, which are carefully designed to only produce values that satisfy a given validity condition, bake in a lot of logic about exactly how that condition is (and is not) satisfied. Ideally, that information could be used for other purposes. This is exactly the idea behind one of the examples in *Composing Bidirectional Programs Monadically* [21]. Xia et al. make QuickCheck-style generators *bidirectional*: they can run forward as generators, and they can also run backward as a checker of the validity condition that the generator enforces.

To enable backward interpretation, the monad is combined with a profunctor as the `Profmonad` class, which just insists that something is both a `Monad` and a `Profunctor`. Profunctors are bifunctors 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 bidirectionally.

```
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 the generator, there are two things to verify: the component parts (`x`, `l` and `r`), and how they are composed. The former can be achieved by bidirectionalising the generators of the component parts. The latter is handled by the contravariant argument to `dimap`; it acts as a witness of how `b` was used to create `a`. For example, the following function represents the way an `Int` is used to create a `Node`:

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

But what happens when `nodeValue` is given a `Leaf`? What happens if `genBST` is given a tree that is not a BST? To accommodate these scenarios, bigenerators must also handle failure:

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

Wrapping the contravariant parameter means that the annotation function will have type `a -> Maybe b`, allowing functions like `nodeValue` to fail gracefully on inputs like `Leaf`. The output of `Nothing` represents that `b` was not used to make `a`. Injecting a `Maybe` this way broadens the applicability of monadic profunctors for bidirectional programming. Internalising a notion of failure means that they are no longer limited to perfect round trips: if something goes wrong, they have the option to fail gracefully by returning `Nothing`. Combining profunctors with a monad that can internalise failure allows for the bidirectionalisation of a generator through the insertion of `dimap` annotations:

```
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)
```

This is a bidirectionalised version of the part of `genBST` that generates nodes. The structure is the same, each composed generator is just replaced with a bigenerator and annotated with `dimap`. The annotation wraps around each bigenerator, adding information about what part of the overall structure it constitutes to facilitate backward interpretation. For example, `nodeValue` knows that integers are generated as the second argument to a `Node`, and when run backward checks that any occurrences of integers that `biChooseInt` says it could have made only occur there.

## 3 REFLECTIVE GENERATORS

Bidirectionalising a generator is useful on its own, but adding a few further tweaks to Xia et al.'s framework gives significantly more power. Our key insight is to extract the idea of "generator choices" into a type class:

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

This type class represents a structure with choice points and marks the way these choices can be made. Each choice in the list is paired with a semantic label or *tag*, `t`, and there is much to be gained by exposing this information.[1]

We combine this extra expressive power with bigenerators to create *reflective generators*:

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

The `Reflective` class packages the aforementioned classes together giving us monadic profunctors (`Profmonad`) that can internalise failure (`ProfunctorPartial`) and have a notion of choice (`Pick`). A unified interface like `Reflective` allows reflective generators to be written using a "classy" (or tagless final [6]) embedding. Reflective generators written in our framework do not have a concrete type like `Gen`, instead they have a universally quantified type like `forall g. Reflective g t b a`, meaning

[1] Generator APIs often treat `choose` as fundamental and implement `oneof` and `frequency` in terms of that. In this case, we felt that being able to semantically associate tags to sub-generators was more important than a theoretically minimal API.

that reflective generators can be interpreted in many ways, examples of which will be explored soon.

But first, what does it actually look like to write one of these programs? This reflective generator of BSTs will serve as a running example:

```
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 ),
      ( "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) )
    ]
```

The recipe for converting a QuickCheck generator (shown on the left) into a reflective generator (right) is fairly straightforward:

(1) Substitute `frequency` or `oneof` for `pick`, and `Gen` for the type class.
(2) Switch to tags. Hopefully, this is intuitive and can be as simple as selecting meaningful strings.
(3) Add the backward annotations. This process can be expedited by our observation that the function provided to `dimap` is uniquely defined by the structure of the data being generated[2]. The `at` combinator provides this function from automatically derived prisms (such as `_Node` in the example).

Once a generator has been converted, it can be interpreted in a variety of ways. As a coherence check, QuickCheck's `Gen` monad should be a member of the `Reflective` class where `genBST` is used to generate BSTs:

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

Since this type is a wrapper around `Gen`, it is a monad, and it is trivially a profunctor (since contravariant parameter is ignored). This means that the only nontrivial instance needed to get an `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 between the list of reflective 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 String Tree Tree

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

[2]For those interested in optics [2], the function is the `match` of the prism corresponding to the data type being generated.

```
Node Leaf (-4) (Node Leaf 10 Leaf)
```

Another interpretation of the `Reflective` class replicates Xia et al.'s checkers. This time `genBST` will be specialised to `UnGenCheck`:

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

This structure is a monad because it is the composition of two monads, `Maybe` and `((->) r)`, and it is a profunctor because it maps from its contravariant parameter. The `Pick` implementation is a bit more interesting, utilising an `Alternative` instance:

```
instance Pick UnGenCheck where
  pick xs = asum (map (filterEq <=< snd) xs)
    where
      filterEq a = UnGenCheck (\b -> if a == b then Just a else Nothing)

instance Alternative (UnGenCheck t b) where
  empty = UnGenCheck (const Nothing)
  ux <|> uy = UnGenCheck (\b -> run ux b <|> run uy b)
```

The `pick` function asks which (if any) of the listed generators could have generated the value in question. It does this by ignoring the tags using `snd`, adjusting the listed checkers to only check for the value in question, and taking the first that does using `asum` (from the `Alternative` instance, which is just a wrapper about the `Maybe Alternative` instance).

Interpreting `genBST (-10, 10)` using 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 behaviors 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, and, even better, there is rich information stored `pick` tags that neither of the above interpretations capitalize on. The next few sections explore the variety of powerful ways to use reflective generators.

## 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 often useful to be able to validate that a generator behaves as expected. This section describes two interpretations of reflective generators that allow for *introspection*: they give the tester the ability to query properties of its own distribution.

### 4.1 Testing Generators

There are two properties one will consider for a generator: soundness and completeness. Soundness promises that every value generated by the generator is valid with respect to certain properties. Completeness asserts that all valid values can be generated.

It is already folklore that QuickCheck generators should be checked 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 actually involve the program under test, but purely for the purpose of testing the generator. Assuming that there is a checker `isBST` (which is often a reasonable assumption as checking is generally simpler than generation), the property checks that any tree produced by `genBST` satisfies `isBST`.

Completeness is as important as soundness, if not more: one may tolerate spurious bug reports due to unsound test input, but missing out on some (categories of) test inputs unknowingly is definitely a bad omen. Despite this, as far as we are aware, there is no good way to check completeness as it requires an automatic way to know if a generator can produce a particular value. 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 correct. In fact basic generators are so obvious which can be produced by automated tools [13, 14]. In the BST example, this will be `genTree`, for values 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 ensures that for every value `t` of type `Tree`, if `isBST t` then `genBST` can generate `t`. (Note that this testing is not vacuous (§3) because the `forAll` combinator is used.)

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. But remember, this testing is done once-and-for-all. The real testing of the program under test will simply use `genBST` with the confidence that it does not miss important areas of the input space. Moreover, one may consider a more sophisticated basic generator that performs better than the likes of `genTree`. The space between the most basic generator and a practical implementation that considers invariants, 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.

These techniques are an important first-pass when evaluating the quality of a generator. They ensure that the generator can produce the expected range of values. But these testing schemes do not confirm that the generator has the right *distribution*. That is addressed 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 of 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" [4]. 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 invaluable for testers, but they are not a complete solution. Their solution provides statistics on the sample of data generated for a property as it has run; we propose 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 what `UnGenProb` does:

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

Each type class that `UnGenProb` is an instance of contributes a different element of probability. Firstly, `Profunctor` is key to getting any probability at all because it is what enables backward interpretations. The rest transfer what they abstractly represent into probability: `Monad` represents sequentiality, thus they contribute knowing the probability of an event occurring after another; `Semigroup` introduces uniform choice, corresponding to the probability that this or that happened; `Pick` collects tester specified choices, which will be tester specified probabilities.

This information is accessed using the 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
```

Alongside the generator, this function requires the weights that the tester wants to assign to each tag. It can then find the probability of the provided value being generated by specialising to the `UnGenProb` type. Akin to checking, this process could fail: the generator may not be able to make the value. In this case, a probability of `0` is returned, otherwise `UnGenProb` does its magic and `unGenProb` extracts the probability.

[Note: We might also have an ungenerator for finding the probability that a space of inputs is generated. Stay tuned!]

## 5 TEST SUITE ANALYSIS

The previous section shows how generators can validate properties internal to itself, but what about external ones? If a test suite is has already been generated, or is provided by some external source, how might a tester evaluate how it is to be useful? Reflective generators' choice annotations make them the perfect tool for answering questions like these. This section starts with some infrastructure, setting up abstractions and tools that are broadly useful, and then explores how that infrastructure can be used to analyse test suites.

### 5.1 Understanding Generator Choices

Similar to the previous section, this section uses the backward direction of a reflective generator to understand how the generator produces a value. However, unlike in the previous section the generator interpretations track annotations more carefully. This gives access to rich information about the generation process and enables interesting new techniques.

*5.1.1 Listing Choices.* The annotations in an 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`. Thus, when generating the value `Node Leaf 5 Leaf`, the generator makes choices:

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

This process of turning an unstructured string of choices into into structured data looks a lot like *parsing*, and it is well known bidirectionalising a parser yields a *pretty-printer* [2, 11, 17, 21]. What does this mean for reflective generators?

The following function takes an 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. 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.

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.

The choice list is built up in the `Monad` and `Pick` instances for `UnGenList`. The `Monad` instance behaves like the composition of the list and writer monads, with `return` producing an empty list of choices, and `(>>=)` concatenating all pairs of choice lists together. `Pick` adds a new choice to the front of each list, depending on exactly which branch is taken. For example, in the following line from `genBST`, the first choice adds `"leaf"` to the list, and the second choice adds `"node"`.

```
pick [("leaf", return Leaf), ("node", do ...)]
```

The end result works as desired, with:

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

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

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

```
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.[3]

With this new type class, it is possible 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 type `[t]` has been replaced by `c t` for any container `c` that is `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 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 analyze 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 its internals of the system under test (e.g., because it is compiled, closed-source, etc.), and 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.

*5.2.1 Choice Frequencies.* One simple, effective way to understand a test suite is measuring the frequencies with which each data constructor of a type appears in a test suit. 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 misses a large portion of the values available to it.

When data types are very simple, there are ways to automatically tune a generator to ensure a uniform distribution over constructors [14], but in general it not easy to know if a generator produces a reasonable distribution of constructor frequencies. In this area, we make two contributions: we refine constructor frequency to *choice* frequency, 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 simply-typed lambda calculus (STLC) terms, which needs to select between available free variables:

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

The labels on the variable indices are all the same, `""`. There is no reason to prefer one free variable over another, so it would be counter-productive to distinguish such granular choices. As a result, a choice frequency measurement correctly ignores such details (simply giving a frequency for the empty tag, which can be ignored), whereas a naïve constructor frequency measurement does not.

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

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 `Frequencies` represents a map from choices frequency of that choice, and is defined as:

```
newtype Frequencies a =
  Frequencies {getFrequencies :: Ord a => Map a Int}

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

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

These instances succinctly describe how the frequencies should be tracked and combined throughout the ungeneration process, and `unGenChoices` can be used to extract the `Frequencies` for a value without any extra code. Those frequencies can be aggregated across a test suite to get a more global picture.

*5.2.2 Combinatorial Coverage.* Moving beyond choice counting, reflective generators can also measure more nuanced test suite metrics like *combinatorial coverage*. Combinatorial coverage comes from the systems literature [7]. Classically, it is 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 it is tested in the 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
```

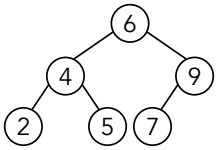
and it also shows behaviours that happen when:

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

etc. In this way, a single test exercises a combinatorial explosion of potentially buggy interactions between 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 work well for algebraic data types because it does not take structure into account, but advancements in the literature present a more general definition [3]. Instead of looking at relationships between configuration values, Goldstein et al.'s notion of coverage measures 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 descriptions count transitive constructor relationships because similar behaviours are often tested regardless of what happens between two constructors.)

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

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

This is conceptually simple to do, but it is somewhat 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` does the obvious thing: 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) :: [Term String]
["node"("5", "node"("4", "leaf", "leaf"), "leaf")]
```

(The tree is rendered via a `Show` instance, but under the hood these are just nested lists of choices.) 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]
```

Running this analysis for each test in the suite gives a wealth of information that can be used to rule out classes of bugs and alert the tester to potential gaps in test coverage.

## 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 wonderful ways. This section shows that the two directions of a reflective generator can be used together to replicate and generalise state-of-the-art strategies for dynamic random testing.

### 6.1 Example-Based Generation

Testers often have intuition about what regions their input space are most interesting; that is what unit tests are all about! However, in traditional unit testing, it is always possible that the next example is the one that will catch the bug. Example-guided testing solves this problem by

combining tester expertise with the power of random testing. Testers provide potent example inputs that are used to guide the generation of further random test inputs, enabling thorough exploration of interrogative input spaces. Consider testing the following expression language, a language that has previously been used to show-case a framework for generating test inputs from examples [20]:

```
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 what if common use-cases (represented by `exampleInputs`) only include a subset of the available operations? A tester might want to ensure that this restricted space of inputs is thoroughly tested. Similarly, a tester might want to explore the space of *uncommon* use-cases (very unlike those in `exampleInputs`). Example-based generator tuning can handle both of these testing strategies, generating common and uncommon inputs by following (or avoiding) the distribution of a set of examples:

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

*6.1.1 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) (Optionally) 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. A "weight map" biases values that are generated:

```
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 alwaysLeafs
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 the first set of weights extracted from a given value (the `listToMaybe` does this a lovely type safe manner, wrapping the result in a `Maybe`):

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

Using `Frequencies` here means that the weight map is now represented as a key-value map. This means that the weights can be adjusted programmatically, 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 functions like `const`. The key-value weight maps can be converted to function weight maps for generation via the following function:

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

Together, `weightedGen` and `unGenWeights` can perform example guided testing. They can be used to test types such as `Expr` if an 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 <$$> genFactor 0
genTerm n =
  pick
    [ ("factor", _Factor <$$> genFactor (n - 1)),
      ("times", _Times <$$> genTerm (n - 1) <**> genFactor (n - 1)),
      ("div", _Div <$$> genTerm (n - 1) <**> genFactor (n - 1))
    ]
```

To ensure termination, `genTerm` takes an `Int` argument that is decremented on subsequent calls, and triggers convergence on a terminal constructor when it hits zero. For simplicity, the semantic tags are just strings that represent the choices made e.g. `"times"` for when the `Times` constructor is selected.

This function makes use of our "bomber" combinators `<$$>` and `<**>`, which alongside the use of prisms expedites the process of writing reflective generators. They are used akin to the applicative combinators `<$>` and `<*>` to "apply" the prism to reflective generator arguments, allowing users to write in a style they are familiar with.

```
(<$$>) :: (ProfunctorPartial p, Profmonad p) => Prism' b a -> p a a -> p b b
(<**>) :: (ProfunctorPartial p, Profmonad p) => p a a -> p b b -> p (a, b) (a, b)
```

When termination is not triggered, `genTerm` makes a choice, using `pick`, between its three constructors. Prisms (`_Factor`, `_Times`, and `_Div`) that have been automatically derived, are used to annotate sub-generators for backward interpretation. When there is more than one sub-generator, such as for `Div`, the application of the prism is chained using `<**>` in applicative combinator style.

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

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

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

As expected, the weight map features representative occurrences of 1, 2, and 3, but no other numbers. 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] -> Frequencies t
mine g = foldr (split . fromJust . unGenWeights g) none
```

Thus targeted testing can be implemented using reflective generators. Moreover, because they are monadic, they can operate on a broader scope of inputs than originating examples of targeted testing [20].

*6.1.2 Targeted Testing with Preconditions.* Since reflective generators are monadic, they can tackle more complicated examples. For example, the `Expr` generators can be upgraded to omit divisions by zero. This new and improved version of `genTerm` does just that:

```
genTerm :: (Reflective g) => Int -> g String Term Term
genTerm 0 = _Factor <$$> genFactor 0
genTerm n =
    pick
      [ ("factor", _Factor <$$> genFactor (n - 1)),
        ("times", _Times <$$> genTerm (n - 1) <**> genFactor (n - 1)),
        ( "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
        )
      ]
```

In the `Div` case, the applicative style has been dropped in lieu of the more expressive monadic do notation. 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.

## 6.2 Adaptive Fuzzing

Another dynamic testing strategy that reflective generators can help with is adaptive fuzzing. Adaptive fuzzers find and exploit interesing inputs via random mutation. Rather than generate new test inputs over and over again, 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 "near-by".

Traditional fuzzers mutate values by flipping or swapping random bits—they generally operate on unstructured binary data—but FuzzChick brings adaptive fuzzing to functional programming with the help of structured mutation [8]. FuzzChick mutates its seed values at a much higher level than bit flips: it might 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 is fantastic for many use-cases, but what if the random values need to satisfy a precondition? Unsurprisingly, reflective generators can help.

*6.2.1 Mutating with Trees.* As with the previous example, doing precondition-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) Mutate the choices randomly.
(3) Run the choices forward through the generator to get a new value.

By construction, this process must produce a value that preserves the generator's precondition, because the mutated value comes from running the generator forward.

Unfortunately, the naïve implementation of this process does not quite work as intended. 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 choices:

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

And those choices might be mutated to:

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

Strangely, even though the change is ostensibly to the left side of the tree, the most natural interpretation of these choices sees

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

and removes *both* children to produce `Node Leaf 5 Leaf`. Ultimately, working with lists of choices is simply 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`. The type classes it implements reinforce this intuition, translating abstract operations to concrete syntax:

```
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 invariant-preserving mutations.

*6.2.2 Manipulating Choices.* In order to do step (2) above, we need some functions to mutate choice trees. 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 captures modifying a node value (`5` to `8`), chopping 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` and the rest are wrapped in `Keep`.

Another mutation is a form of shrinking: simply 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" _`; 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 compatible one from farther down in the tree. 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 actually never works for BSTs—any swaps break the invariant—but it is useful for other types and harmless for BSTs for reasons discussed soon. This again requires a compatibility relation to work well:

```
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`.

*6.2.3 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 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 -- That choice is 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 behavior of `pick` makes a strong assumption about the way the reflective generator is written: in particular, it assumes that the first choice in any given `pick` is not recursive. This is a bit of an unfortunate requirement, but it greatly improves the mutation behaviour. It means that changing a `Leaf` to a `Node` in a BST will add a single node instead of some arbitrary random tree. 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 an reflective generator, a value in its range, and a compatibility relation on labels to produce a random generator of mutated values:

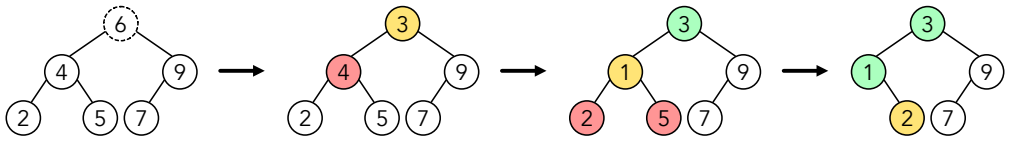
```
mutate :: Ord t => (forall g. Reflective g => g t a a) -> (t -> t -> Bool) -> a -> Gen a
mutate g eq x = QC.elements (unGenChoices g x) >>= (manipulate >=> mutGen g)
  where
    manipulate c =
      QC.oneof
        [ rerollMut c,
          fmap Keep <$> swapMut eq c,
          fmap Keep <$> shrinkMut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

That process is quite confusing! Invariant-preserving mutation is subtle, and it would be easy to do it incorrectly even with a bespoke mutation function. Amazingly, reflective generators do it all automatically! The entire procedure above was carried out by the same `genBST` that we have been working with since §3 and a bunch of entirely generic library code. To our knowledge, generic invariant-preserving mutation like this has never been achieved before.

## 7 LIMITATIONS

Briefly, we should note the gaps between the reflective generator abstraction and the one available in vanilla QuickCheck. First, reflective generators do not 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 parameterise 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. Of course, this is really just cosmetic limitation; size information can be passed around manually in our reflective generators, and sizes could be added back to the abstraction if needed.

Second, the API that we chose for `pick` makes some concessions around manually weighted generators. In §6, we present `WeightedGen`, which recovers most of the 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., by adding computed weight information to a generator's tag type), but the most robust solution would simply 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, the most fundamental limitation of reflective generators is bidirectionalisation: some generators simply cannot be made bidirectional for fundamental computational reasons. A generator might compute a hash that is computationally hard to reverse, generate data and throw it away entirely, or do anything else that would make bidirectional annotations infeasible. This

is an unavoidable limitation, but we do not believe it detracts from the results we present here. Many complex generators (including one for well-typed System F terms) are straightforward to bidirecionalise.

## 8 RELATED WORK

The following ideas intersect with our work in various ways:

*Bidirectional Programming.* Having focussed on generation, it is worth taking a step back and considering other domains of bidirectional programming, and what reflective generators might bring to the table.

Pickling is a bidirectional process where values are packed and unpacked for transmission across a network. In this domain, there exists a bidirectional interface [5], predating the unifying interface of monadic profunctors [21], that also encapsulates notions of compositionality and choice. Since they are applying their notion of choice to a necessarily deterministic process, their notion of choice builds in a mechanism for making the choice. The existence of this work suggests that reflective generator interpretations featuring an appropriate decider could perform pickling.

We are also not the first to take a "classy" approach to bidirectionalisation. Rendel and Ostermann use type classes to create a unified interface for describing syntax that can either be interpreted as a parser or a printer. Similarly to picklers, there is no reason to believe that reflective generators could not be used to represent parser/ printers.

Clearly, there are more horizons to be explored with our new generator abstraction.

*Exposing Generator Choices.* The Hypothesis [10] testing library in Python views generators explicitly as turning 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 that 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; reflective generators seem to be the only abstraction currently able to handle those kinds of cases and "ungenerate" the choices. Second, in §6 we highlight that 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 [16] guides a QuickCheck-style generator with reinforcement learning by externally biasing labelled choice points. Fundamentally, this choice labelling is quite similar to the way choices are highlighted in this work. In principle, a similar approach might work for reflective generators, but it would likely require a more restricted API for choices than the one described in this paper.

*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. [14] 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 constrained by a dense invariant and 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 [12] library is a good point of comparison for the mutation library provided in §6. While Mutagen does not handle invariant-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 lay the foundation for a new age in property-based testing. They provide a unified solution to challenges in generator validation, test suite analysis, and dynamic testing strategies, without requiring significant extra programmer effort. Using our 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 just random generation.

Here are some thoughts on where this work might go in the future.

*Automating Shrinking.* There are a couple of interesting applications in shrinking that have not yet been explored. First, it seems likely that tools like `SmallReGen` in §6 could be useful for automatically shrinking certain data types. It would go almost like mutation, extract choice, 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. This seems quite easily within reach.

Building on this idea, reflective generators might also improve shrinking outside of Haskell. The Hypothesis library [22] 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 (§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 that 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 an 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 expressivity. 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 langauge model could learn to generate values that really capture essence of the examples.

*What else can we get for free?* This section already suggests a few new reflective generator interpretations, on top of the seven presented in the rest of the paper and implemented in our library, but the more the merrier! Each new interpretation has the potential to improve the utility of *every* reflective generator, so spending more time looking for use-cases is certainly worthwhile. We plan to look for new applications in the areas that we already explored, but we also hope to look farther afield for potential ideas.

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

[5] 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/

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

[7] D Richard Kuhn, Raghu N Kacker, and Yu Lei. 2010. Practical combinatorial testing. *NIST special Publication* 800, 142 (2010), 142.

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

[9] Andreas Löscher and Konstantinos Sagonas. 2017. Targeted Property-Based Testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 46–56. https://doi.org/10.1145/3092703.3092711

[10] 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.

[11] 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

[12] 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.

[13] 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.

[14] 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

[15] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 329–340.

[16] 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

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

[18] 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

[19] 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

[20] 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).

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

[22] Experiment Zone. [n. d.]. Hypothesis. https://www.hypothesislibrary.com/