Reflecting on Random Generation

Anonymous Author(s)

Abstract

Property-based testing (PBT) frameworks like QuickCheck allow programmers to validate properties of their code, expressed as boolean functions, against inputs produced by random *generators*. These generators are often carefully handcrafted to ensure they only produce *valid* test cases for the program being tested. Concretely, this means that the random choices available to the generator as it executes are necessarily those that lead to valid test inputs.

We exploit this observation to design *reflective generators*, a new generator abstraction that enables a wider range of PBT strategies by leveraging the validity information latent in QuickCheck-style generators. Reflective generators build on ideas from bidirectional programming, including *monadic profunctors*, to "reflect" on the choices made when generating one value to guide the generation of other values.

We showcase the power of reflective generators by generalising two testing strategies from the literature, demonstrating (1) that reflective generators can be "tuned by example" to mimic the distribution of a suite of example inputs, and (2) that they can be used for validity-preserving mutation of test inputs.

1 Introduction

Haskell programmers love QuickCheck because property-based testing (PBT)—writing mathematical specifications for code and then using randomly generated examples to check those specifications—fits seamlessly with the ethos of the pure functional programming community. On top of that, the language that QuickCheck testers use to write their random data generators is built around a powerful abstraction, the `Gen` monad, that allows generators to express constraints on the values that they generate, ensuring that they always yield valid inputs for the program under test and speeding up testing significantly.

The testing literature is full of different techniques for producing random example inputs. For instance, *example-based tuning* [20] changes a generator's distribution to mimic a set of interesting examples, while *test-case mutation* [1, 10] is used in a variety of testing approaches to explore "the space around" a particular value. But these techniques can be expensive when applied to the complex domains that QuickCheck testers are used to, because they generally involve *rejection sampling*—producing both valid and invalid values and throwing away the invalid ones—which can be costly if valid inputs are rare compared to invalid ones.

Haskell'22, September 15–16, 2022, Ljublijana, Slovenia
2022.

This situation is a bit silly: We may already have put considerable effort into avoiding rejection sampling by handwriting a custom generator that produces only valid inputs, but this generator is only able to produce new values completely from scratch, while both example-based tuning and test-case mutation create new values based on old ones: in example-based tuning, a set of old values is used as a model for what new values should look like, and in test-case mutation a single old value is the starting point from which new values are derived.

But there is a better way. QuickCheck generators make a series of random *choices* during their execution, and those choices are constrained to only those that ensure valid results. If we could constrain the choices further, we could ensure that the new values the generator produces are also related to old values that we have in hand. We can essentially use choices as an intermediate: first we determine which choices produce our old value(s), then we ask the generator to use those choices to constrain future ones. Critically, the generator will make sure that any choice it makes still leads it to a valid value.

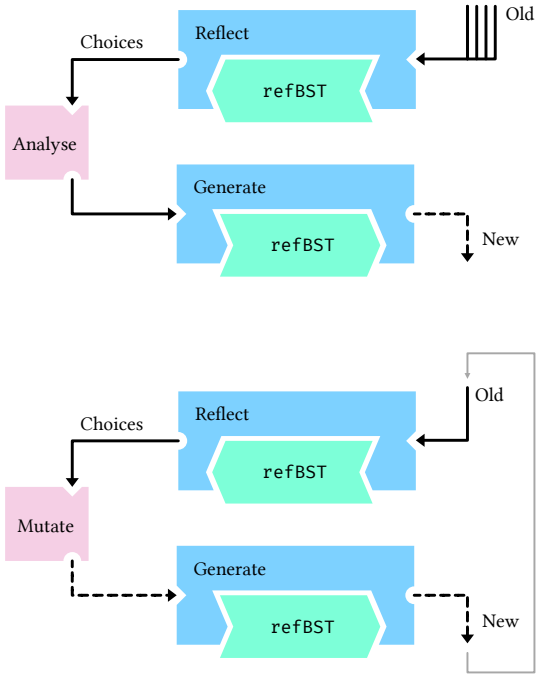

Figure 1. Using a reflective generator, `refBST`. Top: Example-based tuning. Bottom: Test-case mutation. Dashed lines represent random sampling.

With this intuition in mind, we present a new abstraction for writing generators that "reflect" on the choices made when generating one value to guide the generation of other values. Like *lenses* [3], our *reflective generators* can be run in two directions: forward, as normal generators that make choices and produce values, and also backward, taking values and analysing the choices required to produce those values. As shown in Figure 1, the backward and forward modes of reflective generators can be used in tandem to implement both of the generation techniques mentioned above. In both cases, the reflective generator is first run backward to reflect on the choices that produce some old values, then forward to produce new values based on those choices.

After a bit of background (§2), we offer the following contributions:

- We propose *reflective generators*, a variant of standard QuickCheck generators that can be run both forward, to generate random values, and backward, to reflect on the choices that lead to given values (§3).
- We apply reflective generators to *example-based tuning*, simplifying a recently published technique [20] and extending its domain from applicative to monadic generators (§4).
- We apply reflective generators to *test-case mutation*, presenting a novel algorithm for validity-preserving mutation that does not require type-specific heuristics (§5).

We conclude with some limitations of our approach (§6), related work (§7), and ideas for future work (§8).

2 Background

The reflective generator abstraction is built on ideas from the literature on testing and bidirectional programming. This section reviews key concepts.

2.1 QuickCheck Generators

Property-based testing was popularised by the QuickCheck library [2]. One classic demonstration of QuickCheck that we lean on throughout this paper is the binary search tree (BST). Consider the `Tree` datatype and a 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 preserves the BST invariant:

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

To test this property, QuickCheck will try to generate many integers and trees, checking that the property holds for all of them. The `(==>)` combinator does rejection sampling, ensuring that the property is only checked for trees that are valid BSTs. Unfortunately, the likelihood of the tree generator randomly stumbling on valid BSTs—to say nothing of interesting ones—is low. Random trees are likely to fail the `isBST t` check and make `prop_insertValid` succeed vacuously.

This is a typical situation where rejection sampling does not perform well. Luckily, QuickCheck provides tools for creating handwritten 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, do
          x <- chooseInt (lo, hi)
          l <- genBST (lo, x - 1)
          r <- genBST (x + 1, hi)
          return (Node l x r)) ]
```

Sampling from `genBST (lo, hi)` gives BSTs with values in the range `lo..hi`. If that range is empty, there is no choice but to `return` a `Leaf`. Otherwise, the generator may still choose to produce a `Leaf`, or it may produce a `Node` by generating a value in the appropriate range and two subtrees with values in appropriately narrowed ranges.

By taking care with the way ranges are passed through the generator, we guarantee that any value produced by `genBST` is a valid BST. This means that there is no need to check `isBST t` in the property any more: we can write a (much) more performant property that explicitly uses `genBST` to produce trees and drops the precondition check:

```
prop_insertValid' :: Int -> Property
prop_insertValid' x =
  forAll (genBST (-10, 10)) $ \t ->
    isBST (insert x t)
```

2.2 Bigenerators

Generators encode knowledge about what it means to satisfy a validity condition. Our goal with reflective generators is to exploit that knowledge to guide interesting testing strategies. We build on *Composing Bidirectional Programs Monadically* [21], 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. This is less powerful than the "reflecting on choices" paradigm that we will present in §3, but the underlying machinery is the same.

(Fair warning: the abstractions discussed in this section are fairly technical, but they need not be understood in depth to follow the rest of the paper. Readers not familiar with

profunctors may find it useful to refer to [21] for further examples.)

Generating and Checking. It turns out that, to enable backward "checking", a generator should be more than just a monad—it needs to be a *profunctor* as well. 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, while the first flips the arrows. The following Haskell type-class shows how to map over structures like this:

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

The `Profunctor` class has one operation, `dimap`, that knows how to map the covariant argument forward (from `c` to `d` like a normal `fmap` operation) and the contravariant argument backward (from `b` to `a`). These forward and backward maps allow profunctors to capture two different directions of computation at the same time. We build up a forward computation by operating on the covariant parameter, and we build a mirrored backward computation by operating on the contravariant parameter.

Since we do not want to give up the monadic composition used to build up QuickCheck generators, we require that the profunctors we work with be monads in their second argument. This is expressed via the `Profmonad` class, which has no operations of its own:

```
class
  ( forall a. Monad (p a),
    Profunctor p )
  => Profmonad p
```

Additionally, it is useful to be able to express what happens when the backward direction of a computation fails. For example, when checking if a value can be produced by a generator, we want to be able to stop if we know for sure that the value cannot be generated. For this we need one more type class:

```
class Profunctor p
  => PartialProfunctor p where
  internaliseMaybe :: p u v -> p (Maybe u) v
```

Remember, that a function from `p u v` to `p (Maybe u) v` actually gives us a function from `Maybe u` to `u` in the backward direction. We are able to drop the `Maybe` by internalising it into the profunctor type. At a high level, this function enables computations to be partial in the backward direction.

Xia et al. implement forward and backward computation using two different monadic profunctors:

```
type Gen   b a = QuickCheck.Gen a
type Check b a = b -> Maybe a
```

The `Gen` profunctor ignores its contravariant argument, so functionally it just behaves like QuickCheck's standard `Gen` monad. More interesting is `Check`, which takes a value of type `b` and tries to produce an `a`. If the function succeeds with `Just _`, then the value is "valid"; if not, it is "invalid".

This checking process makes most sense for an *aligned* profunctor—one whose contravariant and covariant arguments are the same. In that context, we can think of

```
Check a a = a -> Maybe a
```

as a function that takes an `a` and tries to use the generator to re-construct it. If the reconstruction succeeds, then the value must be in the range of the generator—i.e., there exists a sequence of generator choices that leads to this value. In this way, the checking function is able to determine if a value is valid with respect to the validity condition that the generator enforces.

Programming with Monadic Profunctors. Coding with monadic profunctors is like standard monadic programming, but we need to (1) add `dimap` annotations to make the "backward" direction of the computation explicit and (2) use `internaliseMaybe` when the backward direction might fail. Consider this example, which can be read forward as producing the pair `(4, 5)` or backward as rejecting all values except that pair:

```
1  profmEx :: (Profmonad p, PartialProfunctor p)
2          => p (Int, Int) (Int, Int)
3  profmEx = do
4      x <- dimap (exact 4) id .
5           internaliseMaybe .
6           dimap fst id $ return 4
7      y <- dimap (exact 5) id .
8           internaliseMaybe .
9           dimap fst id $ return 5
10     return (x, y)
11
12 exact :: Int -> Int -> Maybe Int
13 exact m n | m = n = Just m
14 exact _ _         = Nothing
```

Focusing on lines 4–6, we see the way that annotations are used to specify the backward direction of the computation. Since the value `x` will eventually be placed in the first element of the tuple `(x, y)`, we use `dimap fst id` to extract it in the backward direction. Furthermore, since we expect the value to be exactly 4, we need the backward direction to fail when given any other value; we do this with `internaliseMaybe` followed by `dimap (exact 4) id`. (The following section shows how to express annotations like this a bit more ergonomically.)

```
genBST ::
          (Int, Int) -> Gen Tree
genBST (lo, hi) | lo > hi =
  return 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) ) ]
```

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

Figure 2. Converting a QuickCheck generator to a reflective one.

3 Reflective Generators

The bigenerators presented by Xia et al. only say whether or not *there exist* generator choices that result in a given value. What we really want to know is *which* choices! In this section, we describe reflective generators and show how they are able to "reflect" on the choices that produce a particular value and expose them for external use.

3.1 The Reflective Abstraction

Until now, we have not talked explicitly about where generator choices happen, but moving forward we will need to be a bit more specific. The first step in building reflective generators is isolating choices and labelling them; we do this with a new type class:

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

This `pick` function takes a list of generators (of type `g a a`, an aligned profunctor), each of which is paired with a semantic label, or *tag*, that can be used to identify that choice.[1] To choose between a couple of integers, one could write:

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

A *reflective generator* satisfies the combination of `Pick` with the bigenerator classes from the previous section:

```
class
  ( Profmonad g,
    PartialProfunctor g,
    Pick g ) =>
  Reflective g
```

Every `Reflective` supports the operations from `Pick` (enabling labelled choice), `Profmonad` (enabling sequencing and backward computation), and `PartialProfunctor` (the backward direction is allowed to fail). Together, these operations enable rich, bidirectional, labelled generation.

Figure 2 shows a reflective generator for BSTs that we will use throughout the paper. The recipe for converting such a QuickCheck generator (shown on the left) into a reflective generator (shown on the right) is fairly straightforward (colors connect each step to the parts of the generator it modifies):

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

For `refBST`, the backward annotations ensure that only a `Leaf` can result from choosing `"leaf"` (or reaching an empty range), and that only a `Node` (with appropriate arguments) can result form choosing `"node"`. In general, an annotation is needed on all sub-generators, including base-cases, to ensure that the backward direction works properly.

This example eschews the verbose `dimap` annotations from the previous section and instead uses a much more convenient `at` combinator that we provide. The `at` combinator uses *lenses* and *prisms*[2] that can be conveniently derived using Template Haskell[3]. 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` lenses extract the first, second, and third argument respectively. We also provide combinators that let users write backward annotations with normal functions, rather than lenses, but we find the lens presentation quite natural.

This annotation scheme is not difficult to apply to real-world code. For example, we converted all of the generators

[1] For simplicity, we use strings here for choice tags; our implementation generalises this to an arbitrary `Ord` tag type.

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

[3] https://hackage.haskell.org/package/template-haskell

in xmonad[4]—an industrial-strength, dynamically tiling X11 window manager that is famed for being throughly tested in QuickCheck—to reflective generators. The whole exercise took just a few hours, much of it spent understanding the rather complex domain types and testing goals.

3.2 Interpretations

The type of `refBST` is abstracted over a type variable `g`:

```
refBST :: forall g. Reflective g
       => (Int, Int) -> g Tree Tree
```

I.e., for any type `G` that is an instance of `Reflective`, we can get:

```
refBST (-10, 10) :: G Tree Tree
```

We call the type `G` an *interpretation* of the reflective generator.

Reflective generators like `refBST` can be thought of as expressions in a "classily" embedded generator language consisting of constructs provided as type-class methods of the `Reflective` type class (`pick`, `(>>=)` etc.).

Using polymorphism for interpretation like this gives us what is often called a *tagless-final embedding* [7]; besides labelling choices, this is the main place that our reflective generators diverge from bigenerators. Tagless-final style takes advantage of Haskell type classes in a "classy" embedding that easily supports adding both new interpretations (new type-class instances) and new constructs (new type-class methods).

Forward. We find it useful to distinguish between "forward" and "backward" interpretations of reflective generators. Forward interpretations generate values, while backward interpretations reflect on them.

A simple forward interpretation is `Gen`, which simply generates values by making uniformly weighted choices:

```
newtype Gen b a = Gen
  { run :: QuickCheck.Gen a }
```

Note that this type ignores its contravariant parameter; this is always the case for forward interpretations.

Since this type is just a wrapper around `QuickCheck.Gen`, it is a monad, and it can trivially be made a profunctor by ignoring the contravariant parameter. This leaves `Pick` as the only non-trivial part of the `Reflective` class:

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

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

To use this interpretation, the `run` function of `Gen` is applied to `refBST`, specialising its type in the process. The result can be used like a normal QuickCheck generator:

```
gen :: (forall g. Reflective g => g a a)
    -> QuickCheck.Gen a
gen = run

> sample (gen (refBST (-10,10)))
Leaf
> sample (gen (refBST (-10,10)))
Node Leaf 4 (Node Leaf 10 Leaf)
```

Backward. Now we can look at a backward interpretation, again borrowing an example from Xia et al.. This function checks if its input is part of a value that can be produced by the generator; if aligned this is equivalent to checking if a value is in the range of the generator:

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

This structure is both a monad and a profunctor, and its `Pick` instance looks like this:

```
instance Pick Check where
  pick xs = Check $ \b ->
    (listToMaybe
    . mapMaybe (flip run b . snd)) xs
```

This goes through the listed generators (again ignoring tags with `snd`) and runs each on the `b` value that is 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. (Morally, the `Just` value simply means `True`, but we need a `Maybe` here for technical reasons.)

Interpreting `refBST (-10, 10)` using a wrapper function `check` (which simplifies the result of `pick` to a Boolean in addition to specialising with `run`) works as intended:

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

> check (refBST (-10,10)) Leaf
True
> check (refBST (-10,10)) (Node Leaf 4 Leaf)
True
> check (refBST (-10,10)) (Node Leaf 13 Leaf)
False
```

3.3 Ramping Up

Our abstraction recovers the behaviours of both standard QuickCheck generators and "classic" 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 away. In particular, there is rich information in the

[4] https://xmonad.org/

`pick` tags that neither of the above interpretations capitalise on. §4 and §5 present two relatively sophisticated interpretations; in the rest of this section, we explore some smaller examples to make sure that the mental model is clear.

From now on, we'll only show `Pick` instances. `Monad` instances should be clear from the type of the interpretation, while `Profunctor` instances simply apply the covariant and contravariant maps to the appropriate places in the structure and `PartialProfunctor` instances generally just internalise `Maybe` as a `Maybe` in the interpretation type.

Enumerating Values. Another very simple forward interpretation *enumerates* values in the range of the generator:

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

instance Pick EnumGen where
  pick = Enum . concatMap (run . snd)
```

Interpreting `refBST (-10, 10)` with this type produces a list of all possible BST with node values between `-10` and `10`. The implementation of `pick` is dead simple: map `run` over the subgenerators and then concatenate the results. We also have proof-of-concept interpretations that use better enumeration monads like `Logic` [9] and SmallCheck's `Serial` [18], which are more efficient with regard to depth and fairness, albeit with restricted control over enumeration order.

Analysing Probabilities. A more useful example is a backward interpretation that calculates the probability of producing a particular value, given a weighting function on choices. An example of such a weighting function is:

```
w "leaf" = 1
w "node" = 5
w "0"    = ...
...
```

This says that the `node` choice should be made five times more often than the `leaf` choice. The `pick` function looks roughly like this (we have removed a few `fromIntegral` coercions that made it harder to see what was going on):

```
newtype Prob b a = Prob
  { run :: (b, Choice -> Weight)
        -> (a, Rational) }

instance Pick Prob where
  pick xs = Prob $ \(b, w) ->
    let totalWeight = sum [w t | (t, _) <- xs]
        prob = sum
          [ p * q | (t, g) <- xs
                  , let p = w t / totalWeight
                  , (_, q) <- run g (b, w) ]
    in (b, prob)
```

First, we compute the sum of the weights of all possible choices; this will be the denominator of the probabilities. Next, for each pair of a tag and a generator, we compute `p`, the probability that we make that particular choice, and `q`, the probabiltiy that `g` produces the value `b`. The probability of picking a choice and generating `b` is `p * q`, and the total probability of generating `b` is the sum of those individual probabilities.

We can use this interpretation to inspect a reflective generator and understand its distribution. If we are not sure that a particular weighting function produces values with the frequencies we want, we can simply pick some examples and see what their probabilities are.

4 Example-Based Tuning

Random testing is only as good as the distribution that the random values are drawn from. QuickCheck includes combinators like `frequency` for exactly this reason. But the task of "hand-tuning" a generator built from these combinators often requires significant expertise.

Luckily, at least in certain domains, there are more automatic tuning methods that are conceptually simple and work quite well—for example, "example-based" tuning. In this paradigm, testers write down a list of examples that they use as a template for generating more inputs to the program under test. One instance of this approach, *Inputs from Hell* [20], recommends writing down a list of "typical" inputs that the application might commonly encounter and using those to teach a generator how to produce further inputs that are either rather similar to or rather different from the given ones.

In this section, we show that reflective generators can implement this paradigm. Additionally, since reflective generators are more expressive than the grammar-based generators used by Soremekun et al., they can express a wider variety of data structures, including—critically—those constrained by validity conditions.

4.1 Inputs from Hell, Briefly

The running example from *Inputs from Hell* is a generator for arithmetic expressions. Starting from an example expression like `1 * (2 + 3)`, the authors derive a generator that produces inputs like:

```
(2 * 3)
2 + 2 + 1 * (1) + 2
((3 * 3))
...
```

These generated inputs look similar to the example, in the sense that they make the same choices with the same frequencies. The expressions use numbers (`1`, `2`, and `3`), operations (`*` and `+`), and parentheses with weights proportional

to their frequency in the original example. The full expression language has many more constructors, but the example constrains the generator to a subset of those options (e.g., subtraction and division never appear). Additionally, by *inverting* the constructor frequencies, the *Inputs from Hell* method can tune the generator to produce inputs that are very different from the original example:

```
+5 / -5 / 7 - +0 / 6 / 6 - 6 / 8 - 5 - 4
-4 / +7 / 5 - 4 / 7 / 4 - 6 / 0 - 5 - 0
+5 / ++4 / 4 - 8 / 8 - 4 / 8 / 7 - 8 - 9
```

Soremekun et al. show that, in certain domains, this style of tuning is very effective at finding bugs.

Naturally, there are limitations to this approach. Most notably, the tuning above requires that we have access to a context-free grammar (CFG) that describes the set of possible inputs. In our running example of BSTs this is impossible, since the value ordering in a BST are context-sensitive. Luckily, reflective generators can help.

4.2 Tuning Reflective Generators

For consistency with the rest of this paper, we demonstrate the example-based tuning process for reflective generators using binary search trees instead of expressions; a full implementation of the *Inputs from Hell* example can be found in Appendix A.

As promised in §1, the key to implementing example-based tuning is reflection. Specifically, we want to take an example like `Node Leaf 5 Leaf`, understand what choices would lead the BST generator to produce that tree, and then tune the generator to make further choices with weights derived from those past choices. This process is shown pictorially in Figure 1.

Reflecting on `[Choice]`**.** We want a function that takes a `Tree` and returns a `[Choice]` that represents the choices made when producing a given tree. We can get one by instantiating `refBST` with an interpretation called `ReflectList`:

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

This type makes most sense when aligned; then it is a function that takes a value of type `a` and attempts to re-create it using the generator. The generator follows the structure of the input and it records its choices in the `[Choice]`. The result is a list of pairs, since a given value might be produced by the generator in many different (including possibly zero) ways.

We also implement a helper function to call `run` and discard the re-created values:

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

Calling

```
reflectList (refBST (-10, 10))
            (Node Leaf 5 Leaf)
```

gives:

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

That is: to build this tree, the generator first chooses to construct a node, then chooses 5 for the value, and then chooses leaves for both the left and right subtrees.

This is all well and good, but what is actually happening when we call `run`? The `Pick` instance looks like:

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

This runs every available choice and, for those that might lead to the given value, records the choice label in the output list.

The same machinery works for any reflective generator `g`. Given a value `x` of appropriate type, we can reflect on the choices `g` makes to produce `x` by calling `reflectList g x`.

Weighted Choices. Next, suppose we have obtained a big bag of choices by running `reflectList` on a suite of examples. How do we go about generating new values with a similar distribution? Following the lead of *Inputs from Hell*, we start by aggregating the bag of choices into a map of type `Map Choice Int` that tracks the count of each choice.[5] Then we use another `Reflective` interpretation to randomly generate new values, given a weighting function:

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

weightedGen :: Ord t =>
  (forall g. Reflective g => g a a) ->
  Map Choice Int -> QuickCheck.Gen a
weightedGen g = run g . (!)
```

The `weightedGen` function takes the aggregated bag of choices and uses it as a weighting function, ensuring that the new generator makes choices with weights equal to the aggregated frequencies of the choices made to produce a suite of examples.

The `Pick` instance describes exactly how the weights are applied during generation:

[5] One could imagine remembering the *order* of choices that lead to the given outputs and trying to replicate that too; for simplicity, we follow the lead of Soremekun et al. and pay attention only to the distribution of individual choices.

```
instance Pick WeightedGen where
  pick xs = WeightedGen $ \w ->
    frequency' [(w t, run x w) | (t, x) <- xs]
    where
      frequency' ys
        | all ((== 0) . fst) ys =
          oneof (map snd ys)
        | otherwise =
          frequency ys
```

Each choice is made with frequency `w t`, where `w` associates a natural number weight to each tag `t`. This form of weighted choice is similar to QuickCheck's `frequency` combinator, except that the weights are provided externally rather than being built into the generator. (The `frequency'` combinator acts like `frequency`, except when all weights are zero; `frequency` fails in that case, while `frequency'` simply picks uniformly.)

Putting It Together. Gluing everything together, we can build a function `tunedLike` that takes a reflective generator and a suite of examples and produces a plain QuickCheck generator that produces values similar to the examples:

```
tunedLike ::
  (forall g. Reflective g => g a a) ->
  [a] -> Gen a
tunedLike g =
  weightedGen g .
  Map.fromListWith (+) .
  map (,1) .
  concatMap (head . reflectList g)
```

We can call it like this:

```
refBST (-10, 10) `tunedLike`
  [Node Leaf 5 Leaf, Leaf, ...]
```

We can also define an analogous function `tunedUnlike` that does almost the same thing but inverts the weights to get values that are different from the given examples.

This setup does more than just replicate the results of Soremekun et al.: it generalises them. The generators in *Inputs from Hell* are written in a quite constrained format—probabilistic context-free grammars—that is much less expressive than our monadic generator language. This works fine for some classes of values, but it cannot express the kinds of dependencies that monadic generators allow. In the case of the expression example, reflective generators make it possible to encode constraints (e.g., expressions must contain division by zero) that are impossible to encode in a context-free way. And, of course, to generate BSTs, we need the ranges for the subtrees of each node to depend on its label.

In summary, every reflective generator comes with an example-based tuning method for free, and we can build reflective generators for a much broader class of structures than were supported by earlier work on example-based tuning.

5 Test-Case Mutation

The standard QuickCheck approach to testing is a bit wasteful. Suppose a generator goes to the trouble of producing a particularly interesting value. Maybe that value exercises code paths that most values do not, or maybe it has a particular shape that is interesting. Regardless, QuickCheck will use that value once, then discard it and generate the next input from whole cloth. But what if, instead, we could *mutate* values that seem interesting? Then we could "squeeze more juice out of them" by exploring other similar values.

This is superfinally similar to example-based tuning, in the sense that old values are used to produce new ones, but it is operationally quite different in a couple of ways. First, new mutated values are expected to be *structurally similar* to the old values; the *Inputs from Hell* approach does not attempt to do this. Second, whereas examples in example-based tuning are provided by the tester, interesting values that warrant mutation here would typically be discovered with a feedback-driven process—e.g., choosing examples that exercise parts of the system under test that have not been explored by previous tests.

Test-case mutation is used by *coverage guided fuzzing (CGF)* and related techniques [1, 5, 10], and it can be extremely effective at exercising a system under test. Unfortunately, the mutation strategies in existing tools do not respect validity conditions; rather, they simply discard mutants that turn out to be invalid. This can be costly if mutations frequently produce invalid values.

A better approach is to use reflective generators to create random mutations while attempting to preserve validity.

5.1 Value Preserving Mutation

In CGF, 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 "similar". While fuzzers like FuzzChick [10] have been designed with PBT in mind, incorporating mutations for complex structured data.

To see why this is a problem, consider trying to mutate a BST like the one shown in Figure 3. If, for example, we change the root node's value from 6 to 3, we leave the tree in an invalid state, with a 4 to the left of the 3. If we stopped here, we'd need to throw away this mutant tree and try again.

But we do not have to stop: we can go a bit farther and try to fix up the tree to again satisfy the validity condition. This is the idea of *validity-preserving mutation*: take a value that satisfies a validity condition and turn it into a "nearby" value that also satisfies the condition. For BSTs, the process is illustrated in Figure 3: it starts the same way, mutating 6

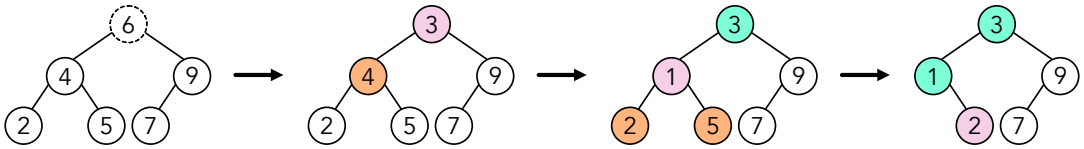

Figure 3. A validity-preserving mutation on a BST.

to 3, then propagates the validity condition down the tree, updating more values until the invariant is restored.

It might seem like such mutators would be a pain to implement, requiring bespoke code for each type of value and validity condition; surprisingly, they can be derived available automatically for any reflective generator!

5.2 Mutating with Reflective Generators

The idea behind mutating with reflective generators is to mutate the *choices* that are made when a value is generated. As with example-based tuning, this requires two coordinated interpretations of the same reflective generator:

1. The backward interpretation gives us the choices that led to a value.
2. We randomly mutate these choices.
3. We then use the forward interpretation to construct a new value using the mutated choices, being careful to recover gracefully if the mutated choices do not actually correspond to choices the generator can make.

This process is shown pictorially in Figure 1.

Note that step 2 is type agnostic, mutating the "raw" choices with abandon, and step 3 uses information baked into the generator to intelligently recover from mutated choices that do not make sense. This combination is extremely powerful: it means that validity-preserving mutation can be performed without hand-writing complex mutators. All that is required is a reflective generator.

Refining the Choice Structure. In §4 we captured choices in a list. This representation turns out to be inadequate here, since it loses some information about where the choices were made. Consider this list of choices and the term it produces:

```
["node", "5", "node", "2", "leaf",
 "leaf", "node", "7", "leaf", "leaf"]
==> Node   (Node Leaf 2 Leaf)
         5 (Node Leaf 7 Leaf)
```

It is hard to see which choices correspond to which subtree. It also makes it hard to preserve the tree's structure once we start mutating the choices. For example, suppose we randomly change the third element from `"node"` to `"leaf"`. The left subtree of the root obviously changes from `(Node Leaf 2 Leaf)` to `Leaf`. When we come to generating the right subtree, the next available choice is `"2"`, which makes no sense in this context, so we discard it and use the next choice, which is `"leaf"`, to complete the tree:

```
["node", "5", "leaf", "2", "leaf",
 "leaf", "node", "7", "leaf", "leaf"]
==> Node Leaf 5 Leaf
```

What's happened is that a small, local change to the list of choices has led to a large, global change in the generated tree. This is not what we want.

We can do better by storing choices in a data type that encodes the branching structure of the generator:

```
data Choices c
  = None
  | Mark c (Choices c)
  | Split (Choices c) (Choices c)
```

(Note that the type is polymorphic over the choice representation `c`, for reasons we will see in a moment.) The choices leading to the tree above would be represented like this:

```
Mark "node"
  (Split (Mark "5" None)
  (Split (Mark "node"
           (Split (Mark "7" None)
           (Split (Mark "leaf" None)
                  (Mark "leaf" None))))
         (Mark "node"
           (Split (Mark "2" None)
           (Split (Mark "leaf" None)
                  (Mark "leaf" None))))))
```

We can change the backward interpretation of the reflective generator from §4 to use this representation instead of lists of choices:

```
newtype Reflect b a = Reflect
  { run :: b -> [(a, Choices Choice)] }

reflect :: (forall g. Reflective g => g a a)
        -> a -> [Choices Choice]
reflect x a = snd <$> run x a
```

The class instances are almost identical to the ones above for `ReflectList`; we just modify the way the choice structure is built. We can run `reflect` on our example tree to get the tree structure above.

Mutating Choices. Now that we can extract `Choices` from values, we need some ways to mutate a tree of choices.

The most obvious mutation is to simply change a single choice to a different one. In the BST example, this might replace one number with another or change a node to a leaf. We can do that with a mutation called `rerollMut`:

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

This function randomly picks a single choice in the tree and wraps that choice in a `Reroll` constructor; all other choice are wrapped in `Keep`. The forward interpretation of the generator (which we will see momentarily), takes these instructions into account: encountering `Keep c` tells it to make choice `c`; seeing `Reroll c` means make any other choice. Deferring the mutation in this way, rather than trying to pick a new choice during mutation, prevents generating certain kinds of nonsense. For example, if our forward direction encounters `Reroll "4"` when generating a node label, it can pick a compatible choice like `"6"` rather than a meaningless one like `"leaf"`.

Here are two more useful mutators:

```
swapMut   :: Choices a -> Gen (Choices a)
shrinkMut :: Choices a -> Gen (Choices a)
```

They behave as one might expect from the names: `swapMut` chooses two subtrees in the choice tree and swaps them, and `shrinkMut` chooses a single subtree to replace the whole choice tree. These mutations will be more or less useful depending on the type of the values being mutated (e.g., `swapMut` is generally a poor choice for BST), but in the case that a mutation produces an invalid tree of choices, the forward interpretation does its best to recover.

These mutations are designed to mimic FuzzChick mutations, which themselves mimic the industry standard fuzzing tool AFL [1]. Together, they form a comprehensive picture of the ways one might want to mutate a structured value. Still, we do not claim that these mutations are the best possible ones, and we expect careful thought will need to be put into exactly which mutations are applied when. For now, our goal is simply to demonstrate that a wide variety of mutations are possible, and that those mutations can be designed without specialising to a particular type of value. Any mutator written over trees of choices can be used to mutate any value using a reflective generator.

Reconstructing a Value. Finally, with a (potentially broken) tree of choices in hand, we can build a new, mutated value. To do this, we need one more interpretation:

```
newtype MutGen b a = MutGen
  { run :: Choices MayReroll
        -> QuickCheck.Gen a }
```

This takes a tree of choices (some of which may have been re-rolled) and produces a distribution over mutated values. The definition of `pick` needs to be fairly clever to deal with the fact that some choices may not be valid:

```
1  instance Pick MutGen where
2    pick gs = MutGen $ \case
3      Mark (Keep c) rest ->
4        case find ((== c) . fst) gs of
5          Nothing -> arb gs rest
6          Just (_, g) -> run g rest
7      Mark (Reroll c) rest ->
8        case filter ((/= c) . fst) gs of
9          [] -> arb gs rest
10         gs' -> arb gs' rest
11     _ -> run (snd (head gs)) None
12     where
13       arb gens rest = do
14         g <- elements (snd <$> gens)
15         run g rest
```

We take our 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 (line 4). If the choice is invalid for some reason, a random valid choice is taken instead (line 5). In the second case, the generator has explicitly been instructed not to choose a particular choice (line 7), so it randomly chooses a different one if it can (line 10). Finally, if the top-level constructor is not a `Mark` at all, it means that this part of the mutated tree of choices has "gotten out of sync" with the structure we are now generating, so the generator greedily makes the first choice available (line 11).

This greedy choice made by `pick` rests on 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 behaviour of mutation. To see why, consider a mutation that changes a `Leaf` to a `Node` in a BST. If we want to keep the mutation "small", we want new subtree to be as small as possible; specifically, we would like to have `Leaf` on both sides of the new `Node`. The greedy choice assumption makes this happen without a fuss. Happily, if the generator terminates at all, it can be rewritten to satisfy this assumption, so this does not pose a significant problem.

To complete the picture, we can write a function that takes a reflective generator and a value and yields a mutated value:

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

Throughout this process, we have given no type-specific mutation instructions at all—the mutation strategy is totally type-agnostic. Coverage-guided fuzzing and other adaptive fuzzing strategies can be made available to anyone with a reflective generator.

6 Limitations

The reflective generator abstraction we have presented has 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
```

that allows testers to parametrise their generator by a size. Building this into the framework allows the testing framework to vary the target size dynamically during testing. Adding size information to reflective generators should be unproblematic.

Second, the API that we chose for `pick` does not directly support manually weighted generators. The `WeightedGen` defined in §4 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:

```
let n = ... in
frequency [(1, foo), (n, bar)]
```

There is currently no way to choose `n` within the weighting function provided to `weightedGen`, because the information needed exists only dynamically during generation. We believe there are workarounds within the framework as we present it, but a more 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. 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 invert or generate data and throw it away entirely (making it impossible to replicate those choices backward). This is an unavoidable limitation.

However, many functions that are commonly thought of as "non-invertible" work perfectly well as reflective generators. Functions with partial inverses are handled automatically by `PartialProfunctor`, 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, we have found that even fairly complex generators (e.g., for well-typed System F terms) are usually straightforward to bidirectionalise.

7 Related Work

Several threads of prior work intersect with ours.

Test Data Producers. Our work depends most directly on QuickCheck [2], 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 [18]. Reflective generators can be instantiated as enumerators, as discussed in §3. Separately, reflective generators are related to *free generators* [4], in that both derive value from labelling generator choices.

Bidirectional Programming. Reflective generators are not the only useful example of incorporating bidirectional programming [3] into a monadic computation.

Pickling is a bidirectional process whereby values are packed and unpacked for transmission across a network. Kennedy [6] presents a bidirectional interface (predating the monadic profunctors interface we build on [21]) that also encapsulates notions of compositionality and choice.

We are also not the first to take a "classy" approach to bidirectionalisation. Rendel and Ostermann [17] use type classes to create a unified interface for describing syntax that can be interpreted as either a parser or a printer. Both of these examples are compatible with reflective generators, in the sense that 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 tree of random choices 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`. That said, Hypothesis cannot shrink a value once it has discarded the choices that produce that value (e.g., if the user wanted to manipulate the value outside of the Hypothesis test-runner); reflective generators are

(to the best of our knowledge) the only abstraction able to recover the choices in such cases.

The RLCheck [16] tool uses reinforcement learning to guide a QuickCheck-style generator by externally biasing labelled choice points. This is related to approaches like *Inputs from Hell* and could potentially be used with reflective generators to provide a more powerful backend for example-based tuning.

Test-Case Mutation. The Mutagen [13] library is a good point of comparison for the mutation library provided in §5. 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 `Choices`). The heuristics applied by our `mutate` function could likely be improved by following Mutagen's lead.

8 Conclusions and Future Work

Reflective generators are a powerful variant of QuickCheck generators enabling input generation strategies that go far beyond standard QuickCheck. They leverage bidirectional execution to provide example-based tuning and test-case mutation, advanced generation techniques that were not previously available for classes of values with complex validity conditions.

We see a number of potential directions for future work.

Automatic Translation. It is fairly straightforward to turn a QuickCheck generator into a reflective one—so straightforward that, in many cases, it might be possible to automate the process, either within Haskell's type system (e.g., with fancier lens combinators) or outside it (e.g., with Template Haskell).

Algebraic Effects. Particularly devious readers may have noticed that interpretations of reflective generators could be provided as monad transformers [11]. We have eschewed this presentation for simplicity, but it should be investigated. Incorporating monad transformers, or alternatively algebraic effects [8, 14, 15, 19], 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 §5 could be useful for automatically shrinking certain data types. This would go almost like mutation—extract choices, shrink those choices, and then regenerate—but extra care would be needed to ensure that the resulting values are always smaller than the original in an appropriate sense.

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 and then *manipulated* in some way before being 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 it is from the original process that generated it.

Learning from Choices. The guided testing example in §4 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. Could one combine reflective generators with more sophisticated learning algorithms?

One way forward might be to 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 for testing.

The main advantage this has over the approach in §4 is expressiveness. Choice weights cannot express relationships between different parts of a data structure, and thus may not capture certain aspects of the examples. By contrast, an appropriate language model could learn to generate new values in a way that better captures the essence of the examples.

References

[1] AFL. 2022. American Fuzzing Lop (AFL). https://lcamtuf.coredump.cx/afl/

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

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

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

[5] Honggfuzz. 2022. Honggfuzz. https://honggfuzz.dev/

[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] Oleg Kiselyov, Chung-chieh Shan, Daniel P Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). *ACM SIGPLAN Notices* 40, 9 (2005), 192–203.

[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] 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.

[14] 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

[15] 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.

[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. 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

[18] 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

[19] 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

[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.

Appendix

A Expression Example

```
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

refExpr :: Reflective g => Int -> g Expr Expr
refExpr 0 = (Term <$> refTerm 0) `at` _Term
refExpr n = pick [ ("term", (Term <$> refTerm (n - 1)) `at` _Term),
                   ("plus", do
                       x <- refExpr (n - 1) `at` (_Plus . _1)
                       y <- refTerm (n - 1) `at` (_Plus . _2)
                       pure (Plus x y)),
                   ("minus", do
                       x <- refExpr (n - 1) `at` (_Minus . _1)
                       y <- refTerm (n - 1) `at` (_Minus . _2)
                       pure (Minus x y)) ]

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

refFactor :: Reflective g => Int -> g Factor Factor
refFactor 0 = (Digits <$> refDigits 0) `at` _Digits
refFactor n = pick [ ("digits", (Digits <$> refDigits (n - 1)) `at` _Digits),
                     ("pos", (Pos <$> refFactor (n - 1)) `at` _Pos),
                     ("neg", (Neg <$> refFactor (n - 1)) `at` _Neg),
                     ("parens", (Parens <$> refExpr (n - 1)) `at` _Parens) ]

refDigits :: Reflective g => Int -> g Digits Digits
refDigits 0 = (Digit <$> int) `at` _Digit
refDigits n = pick [ ("digit", (_Digit <$> int) `at` _Digit),
                     ("more", do
                         x <- refInt `at` (_More . _1)
                         y <- refDigits (n - 1) `at` (_More . _2)
                         return (More x y) ]
  where refInt = pick ([([x], return x `at` like x) | x <- ['0' .. '9']])
```