Reflecting on Random Generation

Anonymous Author(s)

Abstract

In property-based testing (PBT) frameworks like QuickCheck, users validate their properties with inputs produced by random data *generators*. These generators are often carefully written; for the best testing performance, they should only produce inputs that are *valid* relative to the property's precondition. Operationally, this means that the random choices available to a generator during its execution are exactly those that will lead it to a valid input; this can be exploited.

We present *reflective generators*, a new class of generators 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 this reflective process by generalising two testing strategies from the literature. A reflective generator can be tuned by example, mimicking the distribution of a suite of example inputs, and it can also enable validity-preserving mutation of test inputs.

1 Introduction

Haskell users love QuickCheck. Property-based testing (PBT)—writing mathematical specifications for code and then using randomly generated examples to validate 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 a powerful monadic abstraction. The `Gen` monad allows generators express constraints on the values that they generate, ensuring that their results are "valid" inputs to the program under test and speeding up testing significantly. This is a quintessential success story of pure functional programming.

Of course, there is more than one way to test a program; the testing literature is full of intriguing techniques for producing random example inputs. For instance, *example-based tuning* [18] changes a generator's distribution to mimic a distribution of interesting examples, and *test-case mutation* [1, 8] is used in a variety of testing approaches to explore "the space around" a particular value. But there is an annoying mismatch between these techniques and the QuickCheck mindset: these strategies usually require *rejection sampling*, producing both valid and invalid values and throwing away the invalid ones, which can be extremely expensive if valid inputs are hard to generate. If we have already gone to the trouble of writing a custom QuickCheck generator that knows how to produce valid inputs by construction, this feels like a waste!

Both example-based tuning and test-case mutation are tricky because they require us to generate new values based on old values. In the case of 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. Naïve approaches to these processes successfully turn old values into new ones, but they fail to produce valid; they simply do not have enough information. But what if we could use a generator to provide that extra information?

We observe that QuickCheck generators make a series of random *choices* during their execution, and they know exactly which choices lead to valid inputs. This means that we can use choices as an intermediate: first we determine which choices produce our old value(s), and then we ask the generator to use those choices as a guide for future choices. Critically, if the choices we tell the generator to make are

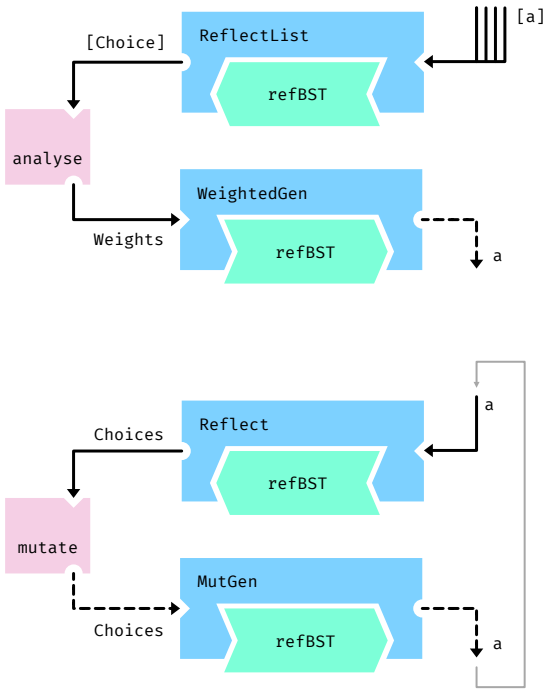

Figure 1. Top: Example-based tuning. Bottom: Test-case mutation. Dashed lines represent random sampling.

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

invalid, it knows how to recover; it can just make an available valid choice and continue generating.

With this intuition in mind, we present a new language for writing generators that operates on a simple mantra: "reflect on the choices made when generating one value to guide the generation of other values". Our *reflective generators* can be used like normal generators, making choices and producing values, but they can also run *backward*, taking values and reflecting on 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 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, and then it is run forward to produce new values based on those choices.

After some useful background (§2), we present the following contributions:

- We propose *reflective generators*, a variant of QuickCheck generators that can 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 [18] and extending its domain from applicatively to monadically described examples (§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 a discussion of some limitations of our approach (§6), related work (§7), and ideas for the future (§8).

2 Background

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

2.1 QuickCheck Generators

Property-based testing was popularised in Haskell (and much of functional programming) 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 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 generates hundreds or thousands of integers and trees, checking that the property holds for all of them. However, the likelihood of the tree generator randomly stumbling on valid BSTs—especially 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 (sampling values from a larger space and filtering out the ones that are invalid) does not perform well. Luckily, libraries like QuickCheck provide tools for creating better "constructive" generators that can, for example, enforce the `isBST` validity condition by construction:

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

This generator is written in monadic style using QuickCheck combinators. 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 produce a `Leaf`, or it may produce a `Node` by generating a value in the appropriate range and two subtrees with values in ranges that are truncated based on the side of the tree they are on.

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 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

When a generator enforces a validity conditions, it encodes information about what it means to satisfy that condition. Our goal with reflective generators is ultimately to use that knowledge to guide interesting testing strategies. The first step along that path is based on one of the examples in *Composing Bidirectional Programs Monadically* [19], 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 discussed in the introduction, but the underlying machinery is the same.

In order to enable this backward "checking", our generator needs to 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, but the first flips the arrows, applying functions "backward". This makes them the perfect structure for encapsulating bidirectionality. The Haskell classes `Profunctor` and `Profmonad` express these constraints:

```
class Profunctor p where
  dimap :: (a -> b) -> (c -> d)
        -> p b c -> p a d
class
  ( forall a. Monad (p a),
    Profunctor p )
  => Profmonad p
```

(Note that `Profmonad` has no operations of its own, it simply ensures that a particular profunctor is a monad in its covariant argument.)

Xia et al. achieve 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 entirely, so functionally it just behaves as QuickCheck's standard `Gen` monad. More interestingly, we have `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", and 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. In other words, there exists a sequence of generator choices that leads to the value in question. In this way, the check function is able to determine if a value is valid with respect to the validity condition that the generator enforces.

Working with monadic profunctors is very similar to monadic programming, but we also need to add `dimap` annotations to make the "backward" direction of the computation explicit. Concretely, this means we need to manage the contravariant type parameters of our sub-expressions; consider this example:

```
profmEx :: Profmonad p
        => p (Int, Int) (Int, Int)
profmEx = do
  x <- dimap fst id (return 4 :: p Int Int)
  y <- dimap snd id (return 5)
  return (x, y)
```

The sub-expression `return 4` has type `p Int Int`, so we can't just write:

```
x <- return 4
```

The problem is that the type of `(>>=)` in this context is:

```
(>>=) :: p (Int, Int) a
      -> (a -> p (Int, Int) b)
      -> p (Int, Int) b
```

This means we need to convert a value of type `p Int Int` to one of type `p (Int, Int) Int`. Luckily, `dimap fst id` does just that. Additionally, the `fst` and `snd` annotations specify exactly where each of the values "end up" in the final result of the program. We see that `x` ends up as the first element of the returned pair and `y` ends up as the second, so when running backward, with a value `p`, we need to ensure that `x` gets `fst p` and `y` gets `snd p`.

Unfortunately, this program still won't behave correctly as a checker. The intended behavior is the following:

```
profmEx (4, 5) => Just (4, 5)
profmEx (_, _) => Nothing
```

But, in fact, for any pair of integers `x` and `y`, the check function will return `Just (x, y)`. The problem is that the `dimap` annotations have no way to reject an invalid value; we really want to be able to annotate `return 4` with a function that rejects any values other than 4! The solution is one more type-class:

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

This type class (which is compatible with both `Gen` and `Check` from earlier), means that covariant `dimap` annotations can produce `Maybe` values, and then we can use `internaliseMaybe` to fix the types.

Using `internaliseMaybe` we can rewrite `profmEx` to do what we expect:

```
profmEx :: (Profmonad p, ProfunctorPartial p)
        => p (Int, Int) (Int, Int)
profmEx = do
  x <- dimap (eq 4 . fst) id .
       internaliseMaybe .
       return $ 4
  y <- dimap (eq 5 . snd) id .
       internaliseMaybe .
       return $ 5
  return (x, y)
  where
```

```
eq m n
  | m == n = Just m
  | otherwise = Nothing
```

Now, instead of just using `fst` in the backward direction, we use `eq 4 . fst` which extracts the first component of a pair and ensures that the value of that component is exactly `4`. Of course, this introduces a `Maybe` in the type, so we need to call `interaliseMaybe` to fix things up. Now this program behaves as expected: using it as a checker only returns `Just _` on exactly the pair `(4, 5)`.

3 Reflective Generators

The bigenerators presented by Xia et al. are a great start, but, as we discuss in the previous section, they can only say whether or not *there exist* generator choices that result in a given value. We want to know *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 those choices for external use.

3.1 The Reflective Abstraction

Until now, we have been relatively informal 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 (represented by the 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* is 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 complex, bidirectional, labelled generation.

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

Figure 2 shows a reflective generator for BSTs that we will continue to return to for illustrative purposes. The recipe for converting a QuickCheck generator (shown on the left) into a reflective generator (shown on the right) is fairly straightforward:

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.

These annotations eschew the verbose `dimap` annotations from the previous section and use the 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 provide combinators for using normal functions, rather than lenses, but , but we think that this presentation is quite natural once you get used to it.

In fact, this annotation scheme is so natural that we have had little trouble applying it to real-world code. We converted all of the generators in xmonad[4] to reflective generators! The whole exercise took only a few hours, and much of that time was spent understanding the domain types and testing goals. Of course, there are a few limitations to conversion (which did not come up in xmonad), but to avoid getting bogged down here we explore them in §6.

3.2 Interpretations

When we wrote `refBST`, we did not give it a concrete type. Instead, its type has a universally quantified type variable `g`:

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

This means that 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.

Keeping the type of our generator polymorphic like this has some amazing advantages; in particular, it allows us to use a generator like `refBST` at multiple different types, each of which provides its own implementation of the functions

[2] https://hackage.haskell.org/package/lens
[3] https://hackage.haskell.org/package/template-haskell
[4] https://xmonad.org/

```
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.

in the program. Reflective generators, such as `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 is often called a *tagless-final embedding* [6], and besides labelling choices this is the biggest place that our reflective generators diverge from bigenerators. Tagless final style takes advantage of Haskell type classes in a "classy" embedding that dually supports the ease of 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. This is not a techncial distinction, just one that helps with understanding. Morally, forward interpretations generate values and backward interpretations reflect on values.

An useful 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 the contravariant parameter; in general this will be the case for forward interpretations.

Since this type is just a wrapper around `QC.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 the type in the process. It can then 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..

Again, this type is a function that checks if its input can be produced by the generator (assuming the profunctor is aligned):

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

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

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

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

Interpreting `refBST (-10, 10)` using a wrapper function `check` (which does some post-processing 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
```

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

3.3 Ramping Up

The rest of our contributions focus heavily on interesting interpretations of reflective generators, so it is worth making sure that the mental model is clear. Here, we present a couple of smaller things that reflective generators can do that are interesting, but not contributions in their own right. Feel free to skip this section and get right to the exciting stuff, but we think these examples are fun and provide a bit more clarity.

For these and future interpretations, we only show the `Pick` instance. `Monad` instances are usually clear from the type of the interpretation, `Profunctor` instances simply apply the covariant and contravariant maps to the appropriate places in the structure, and `ProfunctorPartial` generally just internalises `Maybe` as a `Maybe` in the interpretation type.

Enumerating Values. Another extremely simple forward interpretation *enumerates* all 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 sub-generators and then concatenate the results. This interpretation would be a bit silly in practice—enumeration is best done with a monad that is more efficient with regard to depth and fairness—but as a proof of concept it is a nice simple example.

Analysing Probabilities. A more useful example is this backward interpretation that determines 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
...
```

This says that the node choice should be made five times more often than the leaf choice. The `pick` function looks like:[5]

```
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 our 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 total probability of generating `b` is the sum of those individual probabilities.

Interestingly, this is the only backward direction where `internaliseMaybe` does not simply translate to a `Maybe` in the interpretation type. Instead, internalising a `Nothing` results in a value with zero probability.

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 it can be surprisingly difficult to decide exactly what distribution values *should* be drawn from—the task of "tuning" a generator often requires significant expertise.

Luckily, in certain domains, there are compelling options for tuning that are conceptually simple and work quite well. One such option is to tune a generator based on a set of input examples, an approach we call "example-based tuning." In this paradigm, testers write down a list of examples that is

[5] For presentation purposes, we have removed a few `fromIntegral` coercions that made it harder to see what was going on.

used as a template for generating more inputs to the program under test. One instance of this approach, *Inputs from Hell* [18], recommends writing down a list of common inputs and using those to teach a generator how to produce both common and uncommon inputs for testing.

In this section, we show that reflective generators can build this paradigm. Specifically, a reflective generator can be tuned, using a set of examples, to produce both random test inputs that are either very similar to or very different from those examples. 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 those constrained by validity conditions.

4.1 Inputs from Hell, Briefly

The running example used in *Inputs from Hell* is a generator for expression parse trees. 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 (which are significant, since we care about parse trees) with weights proportional to their appearance in the original example. The full expression language might many more constructors, but the example constrains the generator to a subset of those options. Additionally, by inverting the constructor frequencies, the *Inputs from Hell* method can tune the generator to produce inputs that are 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
```

Admittedly, these expressions are a bit wild, but they certainly would put the system under test through its paces. Soremekun et al. show that, in certain domains, this style of tuning is quite effective at finding bugs.

Of course, there are limitations to this approach. Most notably, the tuning above relies heavily on the fact that the inputs we care about are parse trees. In particular, it requires that we have access to a context-free grammar (CFG) that describes the set of possible inputs. In our running example of binary search trees this is impossible. Luckily, reflective generators can help.

4.2 Tuning Reflective Generators

For the sake of clarity, we will demonstrate the example-based tuning process using binary search trees instead of expression parse trees, but the 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 our 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]`.** Our ultimate goal is to write a function that takes a `Tree` and returns a `[Choice]` that represents the choices made when producing a given tree. Of course, we do not want to implement that function ourselves. Instead, it will be implemented by interpreting `refBST` using a particular instance of `Reflective`. The instance in question is 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, it records its choices in the `[Choice]`. The output is a list of pairs, corresponding to the fact that a value might be produced in many different (and possibly zero) ways.

We can 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"]]
```

This makes sense: when making that 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 input value, records the choice label in the output list.

With this machinery written once, it works for every reflective generator. Given a reflective generator `g` and a value `x` of appropriate type, we can reflect on the choices that `g` makes to produce `x` by calling `reflectList g x`.

Weighted Choices. Now suppose we have aggregated a big bag of choices from a suite of examples. How do we go about generating new values with a similar distribution? Following the lead of *Inputs from Hell*, we can start by aggregating the bag of choices into a map of type `Map Choice Int` that tracks the count of each choice.[6] Then, we can use another `Reflective` interpretation to randomly generate new values, given a weighting function:

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

The `weightedGen` function takes our 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 our suite of examples.

The only thing left to do is see how it all works, and the `Pick` instance contains most of the explanation:

```
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` is the weighting function and `t` is the appropriate choice label. This lets externally weight choices in a similar way to how `frequency` works in classic QuickCheck.

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 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)
```

[6] One could certainly imagine remembering the order of choices and trying to replicate that too, but for simplicity we follow the lead of Soremekun et al..

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 the same thing but inverts the weights to get values that are different from the given examples. In this way we can replicate the common and uncommon input generation from *Inputs from Hell*.

And this setup does more than just replicate the results of Soremekun et al., it generalises them. The generators in *Inputs from Hell* are written using probabilistic context-free grammars, rather than a programmatic generator language. This works well for many classes of values, but it cannot express the kinds of dependencies that monadic generators allow. In the case of the expression parse tree example, reflective generators make it possible encode constraints (e.g., the expression does not contain division by zero) that are impossible to encode in a context-free way. And, of course, there is no way to represent BSTs without some dependence.

Ultimately, every reflective generator comes with example-based tuning strategies for free, and those strategies work in a broader range of domains than prior techniques.

5 Test-Case Mutation

Now, let us look at another testing strategy that can be improved on by reflective generators: test-case mutation.

In some ways, the standard QuickCheck approach to testing can be 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 it has a particular shape that is interesting. QuickCheck would use that value once and then discard it, generating a new input from whole cloth the next time. What if we could *mutate* the interesting value? It may be possible to "get more out of it" by exploring other similar values.

This is the approach used by coverage guided fuzzing (CGF) and related techniques [1, 4, 8], and it can be extremely effective at exercising a system under test. Unfortunately, mutators for these tools do not respect validity conditions, instead simply discarding any mutants that are made invalid. This can be costly if mutations fail frequently. Instead, we would like to use reflective generators to implement mutations that are much better at preserving validity conditions.

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 "close" by. While fuzzers like FuzzChick [8] have been designed with PBT in mind, incorporating mutations for complex structured data, to our knowledge there are no fuzzing techniques explicitly deal with validity conditions.

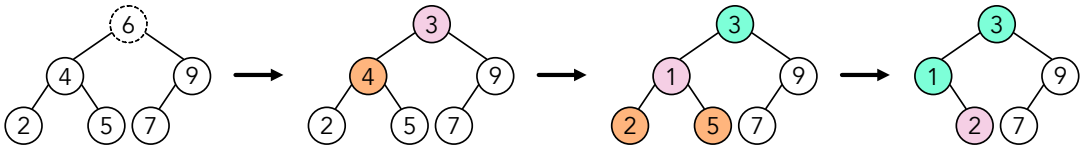

Figure 3. A validity-preserving mutation on a BST.

To see why this is a problem, consider trying to mutate a BST like the one shown in Figure 3. After a mutation, changing the root node's value from 6 to 3, the tree is in an invalid state: there is 4 to the left of the 3. If we stopped here, we'd need to throw away this mutant tree and try again. What we really want is a *validity-preserving mutator*. Such a mutator would take a value that satisfies a validity condition and turn it into a "nearby" value that also satisfies the condition. For BSTs, the process would look something like the full process from Figure 3: it would start the same way, mutating 6 to 3, but then it would propagate the validity condition down the tree, updating more values until the invariant is restored.

While this mutation process looks like it would be a pain to implement, requiring bespoke code for each type of value and validity condition, it is actually available automatically for any reflective generator! The mutation process in Figure 3 is the real mutation process carried out by `refBST`, interpreted as a mutator. In the rest of this section, we explain how that works.

5.2 Mutating with Reflective Generators

Mutating with reflective generators, boils down to mutating the choices that are made when a value is generated. As with example-based tuning, this requires two reflective generator interpretations:

1. Use a backward interpretation to get the choices that led a value.
2. Randomly mutate the choices.
3. Use a forward interpretation to construct a new value using the mutated choices, being careful to recover gracefully if the choices do not match choices the generator can actually make.

This process is shown pictorially in Figure 1.

Note that step 2 is type agnostic, mutating 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, but lists forget information about how the choices were actually 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)
```

Even knowing how the generator works, it is hard to see which choices correspond to which subtree. This can make it hard to preserve the tree's structure under mutation.

For example, if we try to mutate this tree in a way that ostensibly changes just one node to a leaf, we actually end up changing both children! Look what happens:

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

The most logical way to deal with choices that cannot be used, such as the number `"2"`, is to ignore them and await a usable choice, leading the generator to simply choose two `Leaf` children and discard the remaining choices.

A representation with more structure, such as the following, would be far easier to understand:

```
"node" "5"
    "node" "2"
        "leaf"
        "leaf"
    "node" "7"
        "leaf"
        "leaf"
```

Represented as a tree it is unambiguous who is whose parent, and the different children are clearly separated.

We can achieve this by storing choices in a data type capable of encoding dependency information:

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

This data type encodes exactly lists enhanced with dependency information: `None` corresponds directly to `[]`, `Mark` corresponds to `(:)`, and `Split` introduces another dimension that can encode dependency. Our example tree would be represented by this structure:

```
Mark "node"
  (Split (Mark "5" None)
  (Split (Mark "node"
```

```
      -- can be changed with  dependencies
        (Split (Mark "7" None)
        (Split (Mark "leaf" None)
               (Mark "leaf" None))))
      (Mark "node"
        (Split (Mark "2" None)
        (Split (Mark "leaf" None)
               (Mark "leaf" None))))))
```

A representation that binds the number and subtree choice to the node choice, preventing the previous mishaps.

We can update the backward interpretation of the reflective generator from §4 to output choices with this structure:

```
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 for `ReflectList`, we just modify the way the choices structure is actually 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 canonical ways to mutate a tree of choices.

The most obvious mutation is to simply change a single choice 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`. When we design the forward direction of our generator, can take these instructions into account: encountering `Keep c` means to make choice `c`, whereas seeing `Reroll c` means to make any other choice. Deferring the mutation in this way (rather than trying to pick a new choice in the mutator) prevents certain kinds of mistakes. When our forward direction encounters `Reroll "4"` it can be smart enough to pick another compatible choice like `"6"` rather than a nonsensical one like `"leaf"` that will always result in a broken choice tree.

Here are two more useful mutators:

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

They behave as one might expect: `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 direction of our reflective generator will do its best to recover.

These mutations are designed to mimic FuzzChick mutations, which themselves mimic AFL mutations. 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 choices. Instead, our goal is 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 mutators written over trees of choices can be used to mutate any value using a reflective generator.

Reconstructing a Value. With a (potentially broken) tree of choices in hand, we can finally build a new value and complete the mutation process. To do this, we need a final reflective generator interpretation:

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

The type should be fairly unsurprising: we take a reflective generator and a tree of choices (some of which have been rerolled) and produce a distribution over mutated values. The potentially surprising part comes in the way we interpret `pick`:

```
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 <- QC.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 3). If the choice is invalid for some reason, a random valid choice is chosen 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

9). 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 (line 11).

This final behaviour of `pick` makes one strong assumption about the way the reflective generator is written: it assumes that the first choice in any given `pick` is not recursive. This is a somewhat unfortunate requirement, but it greatly improves the mutation behaviour. It means that changing a `Leaf` to a `Node` in a BST will add just that one node instead of some arbitrary random subtree. The non-recursive requirement is also consistent with many QuickCheck generators in the wild, so in practice we do not expect this to be a large problem. Regardless, this assumption can be relaxed with a slightly modified version of `MutGen`, if needed.

Putting it all together we can write a function that takes a reflective generator and a value and mutates it:

```
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 ]
```

We want to stress that all of the type-specific mutation instructions from the reflective generator itself—the mutation strategy is totally type-agnostic. With this approach, coverage-guided and other adaptive fuzzing strategies can be made available to anyone with a reflective generator in hand.

6 Limitations

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

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

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

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

Second, the API that we chose for `pick` makes some concessions around manually weighted generators. The `WeightedGen` defined in §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:

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

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

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

However, many traditionally "non-invertible" programs are perfectly fine in reflective generators. Functions with partial inverses are handled automatically by `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, many complex generators (including one for well-typed System F terms) are 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 [16]. Reflective generators can be made into much simpler enumerators, as discussed in §3. Separately, reflective generators are related to *free generators* [3], in that both derive value from labelling generator choices.

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

Pickling is a bidirectional process whereby values are packed and unpacked for transmission across a network. Kennedy present a bidirectional interface predating the monadic profunctors interface we build on [19], that also encapsulates notions of compositionality and choice. We are also not the

first to take a "classy" approach to bidirectionalisation. [15] uses type classes to create a unified interface for describing syntax that can either be interpreted as a parser or a printer. Both of these examples are quite compatible with reflective generators—either could be implemented by a reflective generator if there was a compelling testing use-case.

Exposing Generator Choices. The Hypothesis [10] testing library in Python views generators as processes that turn a stream of random bit-flips into a data structure. Operations like shrinking are done by shrinking the underlying choices and then regenerating, in much the same way as we do with `shrinkMut`. There are two main differences to highlight. First, Hypothesis cannot shrink a value once it has discarded the choices that produce that value (e.g., if the user wants to manipulate the value outside of the Hypothesis test-runner); reflective generators are (to the best of our knowledge) the only abstraction currently able to handle those kinds of cases and recover the choices. Second, as we highlight in §5, 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 [14] uses reinforcement learning to guide a QuickCheck-style generator by externally biasing labelled choice points. Fundamentally, this choice labelling is similar to the way choices are highlighted in this work. Thus, in principle, a similar approach might work for reflective generators, but it would likely require a more restricted API for choices than the one we use.

Test-Case Mutation. The Mutagen [11] 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 that enable input generation strategies go far beyond the standard QuickCheck approach to example generation. They leverage bidirectional execution to provide example-based tuning and test-case mutation, techniques that were not previously available for classes of values with complex validity conditions.

There are a number of potential directions for future work.

Algebraic Effects. An astute reader may have noticed that interpretations of reflective generators could be provided as monad transformers [9]. This is a presentation that we have avoided for simplicity, but a connection that should be investigated. Incorporating monad transformers, or alternatively algebraic effects [7, 12, 13, 17], 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. It would go almost like mutation—extract choices, shrink those choices, and then regenerate—but extra care would need to be taken to ensure that the resulting values are really shrinks and not unrelated values.

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

Learning from Choices. The guided testing example in §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. But this form of learning is quite simple. Is it possible to combine reflective generators with more sophisticated learning algorithms?

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

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

The main advantage this has over the approach in §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. In contrast, an appropriate language model could learn to generate values that really capture 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] Harrison Goldstein and Benjamin C. Pierce. 2022. Parsing Randomness: Unifying and Differentiating Parsers and Random Generators. arXiv:2203.00652 [cs.PL]

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

[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] 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

[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] 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

[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] 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.

[12] 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

[13] 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.

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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).

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

Appendix

A Expression Parse Tree 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']])
```