Reflecting on Random Generation

ANONYMOUS AUTHOR(S)

The random data generators used in property-based testing are carefully crafted programs that encode what it means for a test input to be valid and interesting. This has led some researchers to repurpose generator programs, using them for test-case shrinking and mutation. But these techniques make a strong assumption: they assume that the value being shrunk or mutated was recently generated, and that the random choices used to produce the value are available.

We develop *reflective generators*, a framework for writing random data generators that can "reflect" on the choices made when producing a given value. Reflective generators combine ideas from two existing abstractions—free generators and partial monadic profunctors—to generalize the aforementioned shrinking and mutation algorithms to work with any input that *could* have been produced by the generator. Reflective generators generalize a published algorithm for example-based generation; they can also be used as enumerators, validity checkers, and more. For each of these cases, we replicate relevant empirical results.

1 INTRODUCTION

Property-based testing, popularized by Haskell's QuickCheck [Claessen and Hughes 2000], draws much of its bug-finding power from random data *generators*. These programs are carefully constructed, and encode important information about the system under test. In particular, QuickCheck generators like the one in Figure 1a capture what it means for a test input to be *valid*—in this case, ensuring that a tree satisfies the binary search tree (BST) invariant by keeping track of the minimum and maximum allowable values in each sub-tree. This generator is not just a program for generating binary search trees (BSTs), it defines BSTs.

This observation has led frameworks like Hypothesis [MacIver et al. 2019], arguably the most popular PBT framework with 6,500 stars on GitHub and an estimated 500,000 users [Dodds 2022], to repurpose generators as part of algorithms for data manipulation, including test-case *shrinkers* and *mutators*. These algorithms do not operate directly on data; instead, they operate on the generator's source of randomness: shrinking or mutating a value is accomplished by shrinking or mutating the *random choices* that produced that value, and then running the generator again on the modified choices [MacIver and Donaldson 2020]. Techniques like these treat generators as *parsers*, capitalizing on a perspective formalized by Goldstein and Pierce [2022] with their *free generators*. Viewing generators as parsers has two distinct advantages over other approaches: (1) shrinking (and mutation) code can be written once-and-for-all, since the modifications are applied to sequences of choices instead of the data itself, and (2) the modified test cases will necessarily satisfy any preconditions that the generator was designed to enforce (e.g., the BST invariant), since they are ultimately still produced by the generator.

Ideally, the type-agnostic, validity-preserving approaches that Hypothesis implements should subsume other more manual approaches. Why use any other kind of shrinker? Unfortunately, the current Hypothesis approach assumes that shrinker has the original random choices that the generator made when producing the value it plans to shrink, resulting in it not working when shrinking is separated (in time or space) from generation. The most important time Hypothesis shrinking fails is when the value was not generated in the first place—perhaps it was provided by an author of the code as a pathological example or perhaps it came from a real-world crash. More subtly, Hypothesis shrinking also breaks down if the value was modified between generation and shrinking or saved without a record of the choices. In all of these cases, shrinking would make huge difference to debugging, if it were available.

ICFP'23, September 4–9, 2023, Seattle, WA, USA
2023.

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

(a) QuickCheck generator.

```haskell
bst :: (Int, Int) -> Reflective Tree Tree
bst (lo, hi) | lo > hi = exact Leaf
bst (lo, hi) =
  frequency
    [ ( 1, exact Leaf),
      ( 5, do
        x <- focus (_Node._2) (choose (lo, hi))
        l <- focus (_Node._1) (bst (lo, x - 1))
        r <- focus (_Node._3) (bst (x + 1, hi))
        return (Node l x r) ) ]
```

(b) Reflective generator.

Fig. 1. Generators for binary search trees.

To use Hypothesis-style shrinking on an arbitrary value, the shrinker needs some way of retrieving a set of random choices that produce that value. Luckily, inspiration can be drawn from the grammar-based testing literature, specifically *Inputs from Hell* [Soremekun et al. 2020]. Soremekun et al. describe a way to produce test inputs that are similar to an existing one. Starting with a grammar-based generator, they first use the grammar to parse the input, determining which *productions* must be expanded to produce that input. Then, they bias a generator to expand those productions more often, thus resulting in more inputs that are similar to the original. In essence, this approach determines which generator choices lead to a desired value by *going backward*, and parsing the value with the same grammar that generated it.

The final step, then, is finding a way to reverse the computation of an arbitrary monadic generator. The *Inputs from Hell* approach does not suffice: it only works for generators based on context free grammars. A more powerful approach can be found in the to the bidirectional programming literature. Xia et al. [2019] describe *partial monadic profunctors*, that build on standard monads with extra operations that can be used to describe bidirectional computation. This infrastructure, along with the parsing-as-generation perspective of free generators, enables exactly the kind of bidirectional generation needed to extract random choices from a value.

Our contribution, *reflective generators*, is a language for writing bidirectional generators (demonstrated in Figure 1b) that can "reflect" on the choices made when producing a particular value. They subsume the grammar-based generators from *Inputs from Hell*, and, critically, they enable Hypothesis-style shrinking and mutation for arbitrary values in the range of the generator. Furthermore, reflective generators are built on *freer monads*, meaning that they can be interpreted in any number of ways besides generation and reflection. We have implemented three more interpretations that demonstrate the versatility of reflective generators as testing utilities.

Following a brief tour through some background (§2), we offer the following contributions:

- We present *reflective generators*, a framework that fuses *free generators* and *partial monadic profunctors* into a flexible domain-specific language for PBT generators that can reflect on the choices that produce a given value. (§3)
- We demonstrate the core behavior of reflective generators by generalizing prior work on example-based generation. Our implementation subsumes the *Inputs from Hell* generation algorithm, replicating empirical results from the paper, and extending it to work with monadic generators. (§4)

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b

do { x <- m; f x } = m >>= f

type Gen a = Int -> a
```

(a) Definitions for monadic generators.

```
data Freer f a where
  Return :: a -> Freer f a
  Bind :: f a
       -> (a -> Freer f b)
       -> Freer f b

data Pick a where
  Pick :: [(Weight, Choice, Freer Pick a)]
       -> Pick a
```

(b) Definitions for free generators.

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

  lmap :: (d -> c) -> p c a -> p d a
  lmap f = dimap f id

class Profunctor p => PartialProf p where
  prune :: p b a -> p (Maybe b) a
```

(c) Definitions for partial monadic profunctors.

Fig. 2. Background definitions.

- We apply reflective generators to manipulate user-provided inputs. Reflective generators enable generator-based shrinking and mutation algorithms, even when the original generator randomness is not available. Again, we replicate empirical results, showing that our shrinkers are at least as effective as other automated shrinking techniques. (§5)
- We leverage the flexibility of the reflective generator abstraction to implement other testing tools. Reflective generators can be re-cast as checkers of generator predicates, enumerators of test-cases, and "completers" that can randomly complete a partially-defined value. (§6)

We conclude with related work (§7) and discussion of future directions (§9).

2 BACKGROUND

The abstractions we present in this paper rely on a significant amount of prior work. In this section, we describe the ideas that make reflective generators possible.

2.1 Monadic Random Generators

Property-based testing was popularized by QuickCheck [Claessen and Hughes 2000], and the basic idea of testing executable properties using monadic generators has endured for more than two decades. The core structures QuickCheck relies on are relatively simple, and shown briefly in Figure 2a. The (simplified) `Gen` type represents the type of random generators; it treats the input `Int` as a random seed, and uses it to produce a value of the appropriate type. Generators, like the one in Figure 1a, are easy to write because they are *monads* [Moggi 1991]; the details of the `Monad` interface are out of scope for this paper, but at a high level monads provide a neat interface for chaining effectful computations.

2.2 Free Generators

Interfaces like the one provided by the `Monad` type class can be reified into a "free" data structure. These structures represent each operation of the type class as a data constructor; then those data constructors can be interpreted, potentially in multiple different ways. There are multiple such free structures for the monad interface, but we focus on the freer monad [Kiselyov and Ishii 2015],

shown in Figure 2b. The structure reifies `return` as `Return` and `(>>=)` as `Bind`, and it uses an extra type constructor `f` to capture any operations that are specific to a given monad.

Free generators [Goldstein and Pierce 2022] use this scheme, instantiating `f` as the type constructor `Pick`, which represents a choice between sub-generators. The `Pick` constructor is used to implement familiar QuickCheck-style combinators like `choose` which generates an integer in a given range and `oneof` which randomly selects between a list of generators. Goldstein and Pierce use free generators to connect random generation and parsing, interpreting the same free generator as both a generator and a parser and proving theorems that relate their behavior. A similar structure is used in the internals of the Crowbar [Dolan and Preston 2017] library in OCaml.

2.3 Partial Monadic Profunctors

The standard category theoretic construction of a profunctor allows for both "forward" and "backward" mapping operations. This appeals due to our desire to "go backward" and recover random choices. Profunctors are realized as the `Profunctor` class (Figure 2c) (popularized in Haskell by Pickering et al. [2017]), where the operations are called `rmap` and `lmap` respectively, and combine together into `dimap`.

Roughly speaking, profunctors operate in two "modes": when running forward, they use the covariant mappings to build values up; and when running backward, they use the contravariant mappings to break values down. The type `p b a` should be understood as a program that can "examine an instance of `b` and produce an `a`." Profunctors that are *aligned*, that is of type `p a a` for some `a`, re-trace the computation that produces their input. This process of reproduction is how a profunctor can simulate running both forward and backward. However, programming with profunctors is tricky when the profunctor is also a monad: alignment breaks down around monadic binds. Consider the type:

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

The contravariant type parameter does not change between the left and right arguments to a bind, but the covariant one does. The `lmap` operation makes it possible to take an aligned profunctor, `p a a` and turn it into a `p b a` by providing an *annotation* of type `b -> a`.

Xia et al. [2019] introduce *partial monadic profunctors* (PMPs) that use profunctors to encode bidirectional computations monadically by adding one more operation, `prune`, and ensuring that the profunctor is also a monad. (We have renamed `prune` from the original "`internaliseMaybe`", as it is more intuitive for our purposes where it prunes invalid values from generation.) Unlike with profunctors, which can only be annotated with total functions, PMPs can be annotated with partial functions. The combinator `comap` demonstrates this generalized annotation:

```
comap :: (c -> Maybe b) -> Reflective b a -> Reflective c a
comap f = lmap f . prune
```

Partial annotations are necessary to capture branching control flow, and they mean PMPs can capture a broad range of bidirectional computations.

The duality of PMPs enables all-in-one programs that are both parsers and pretty printers, getters and setters, and, critically, generators and *checkers* for the preconditions that the generators maintain. We return to checkers in §6; for now, we consider a more general viewpoint.

3 REFLECTIVE GENERATORS

Reflective generators combine free generators with PMPs, enabling a host of generalized testing algorithms. In this section, we explain the intuition behind reflective generators, describe their

structure, discuss their various interpretations, explore their theory, and finally examine practical examples.

The basic structure of a reflective generator comes from adding the partial monadic profunctor operations, `dimap` and `prune`, to the `Pick` datatype. We call this extended type `R`, for reflective generators, and implement it in the following way:

```
type Weight = Int
type Choice = Maybe String

data R b a where
  Pick :: [(Weight, Choice, Freer (R b) a)] -> R b a
  Lmap :: (c -> d) -> R d a -> R c a
  Prune :: R b a -> R (Maybe b) a
```

The `Pick` constructor has two small changes from the free generator presentation: we add an extra contravariant type variable `b`, and we modify the `choice` type to optionally elide choice labels.[1] Then we add `Lmap` which captures the backward direction of `dimap`—there is no need to explicitly represent `dimap` in its entirety, we will be able to encode it using the monad structure. And we also add `Prune`, with the expected type.

A reflective generator is a freer monad over `R b`:

```
type Reflective b a = Freer (R b) a
```

3.1 Intuition: How do reflective generators reflect?

Specializing the intuition from §2, the type `Reflective b a` of reflective generators should be understood to mean a program that can "reflect on choices that result in a `b` while generating an `a`."

As with PMPs, reflective generators use annotations to fix up the types around monadic binds, but what should these annotations do? Following the intuitive interpretation of the types, the goal is to take a generator that reflects on choices in an `a` and turn it into one that reflects on choices in a `b`. Here's the key: it suffices to show how to *focus on part of the* `b` *that contains an* `a`, because that focusing turns `a` choices into `b` choices. Put another way, the mapping that fixes the bind type should embed a mapping of type `b -> a` (or `b -> Maybe a`) that focuses on the `a` part of the `b`.

To see this in action, consider the example in Figure 1b, paying attention to the first bind in the `Node` branch, which looks like

```
do
  x <- focus (_Node._2) (choose (lo, hi))
  ...
```

where `...` continues on to produce the rest of the tree. The call to `choose` results in a `Reflective Int Int`, but the type of the enclosing monad is `Reflective Tree` so the type, we add an annotation bind that focuses on an `Int` in a `Tree` to get a `Reflective Tree Int`. In the example, we annotate with `focus (_Node._2)` (this syntax is introduced in the next section) but the following is equivalent:

```
comap (\ t -> case t of { Leaf -> Nothing; Node _ x _ -> Just x })
```

Stepping back, the process of reflecting on choices is all about the interaction between binds and `comaps`. A value flows through the program, and at each bind, the `comap` focuses on the part of the value that the left side of the bind should reflect on. If the focusing fails, it means that this whole branch of the generator is incorrect—there is no way to produce the desired value—but if it

[1] As in *Parsing Randomness* we represent weights in `Pick` as integers for simplicity, but formally they are required to be strictly positive; this will be necessary to prove Theorem 3.4.

succeeds then the left side can reflect on that part of the value, extracting some choices, and then reflection can continue on the right side.

With this intuition in mind, we can move onto the technical details of reflective generators.

3.2 Structure: Combining Free Generators and Partial Monadic Profunctors

We designed reflective generators to stay as close as possible to standard QuickCheck syntax while incorporating the benefits of both free generators and PMPs. We need quite a few definitions to get there, but at the end we will arrive at a language, which we believe is nice to work with, due to its grounding in existing syntax. (In §9, we discuss plans to validate this belief with a proper user study.)

Building a Domain-Specific Language. We implement a variety of combinators that make reflective generators easier to read and write. Here, we limit our discussion to only those that are informative or necessary later in the paper; the full suite can be found in our Haskell artifact.

Reflective generators are, as expected, partial profunctors:[2]

```
instance Profunctor Reflective where
  dimap _ g (Return a) = Return (g a)
  dimap f g (Bind x h) = Bind (Lmap f x) (dimap f g . h)

instance PartialProfunctor Reflective where
  prune (Return a) = Return a
  prune (Bind x f) = Bind (Prune x) (prune . f)
```

The way `dimap` interacts with `Bind` explains why only the `Lmap` constructor is necessary: the forward mapping can just be passed down the computation and applied at the leaves. Behind the scenes, the `Functor`, `Applicative`, and `Monad` operations are also implemented for free from the freer monad.

We found it convenient to use *lenses* [Foster 2009], to focus on parts of the structure in the backward direction, so we implement a version of `comap` that uses a (potentially partial) *getter*:

```
focus :: Getting (First b) c b -> Reflective b a -> Reflective c a
focus = comap . preview
```

Type is not terribly instructive; this just makes it possible to avoid annoying pattern matches in simple `comaps`:

```
focus (_Node._2) = comap (\ t -> case t of { Node _ x _ -> Just x; Leaf -> Nothing })
```

One more convenient helper for implementing backward annotations is `exact`, which operates like `return` but ensures that the returned value is exactly the expected one:

```
exact :: Eq a => a -> Reflective a a
exact a =
  comap (\ a' -> if a == a' then Just a else Nothing) (Pick [(1, Nothing, return x)])
```

Finally, to bring reflective generator syntax closer to QuickCheck's, we implement a few choice combinators that are easier to work with than the `Pick` constructor:

```
pick      :: [(Int, String, Reflective b a)] -> Reflective b a
labelled  :: [(String, Reflective b a)] -> Reflective b a
frequency :: [(Int,    Reflective b a)] -> Reflective b a
```

[2]Unfortunately these definitions do not quite work in Haskell, since type constructors cannot be partially applied. We simply implement the operations as functions and shadow the type-class definitions.

```
oneof     :: [Reflective b a] -> Reflective b   a
choose    :: (Int, Int)       -> Reflective Int Int
```

The implementations of these functions are mostly uninteresting, but it is worth noting that `frequency` and `oneof` forego labels—their choices are unlabeled, which may or may not be desirable depending on the intended interpretation.

Swept Under the Rug. The real type that we use for `R` is a bit more complicated than the one above. The actual implementation looks like:

```
data R b a where
  -- R as before (Pick, Lmap, Prune)
  ChooseInteger :: (Integer, Integer) -> R Integer Integer
  GetSize :: R b Int
  Resize :: Int -> R b a -> R b a
```

First, we add a constructor `ChooseInteger` for picking integers from an arbitrary range. Technically, this is implementable via `Pick` by simply enumerating all of the integers in the desired range, but doing so is inefficient if the range is large. Adding a separate structure for choosing between integers makes it easy to implement much more efficient interpretation functions later down the line.

Second, we add two constructors, `GetSize` and `Resize` that are analogous to similar operations implemented by QuickCheck. Maintaining size control is critical for ensuring generator termination, but explicit size operators like these are not actually necessary: sized generation can just as easily be handled by passing sizes manually as parameters to the generators. Still, internal size control cleans up the API of the combinator library and makes generators more readable, so we decided it was worthwhile to bake those operations into the type.

In future sections, we often elide definitions pertaining to these operations, but they do present a couple of theoretical complications that we note in §3.4.

3.3 Interpretation: Generation Forward, Focusing Backward

As with free generators, reflective generators do not do anything interesting until they are *interpreted*. Interpretation functions describe how the inert syntax of the generator program should be executed, either "forward" as some kind of generator, "backward" to reflect on choices, or even some combination of the two. Unless otherwise noted, the interpretations we present in this paper work for any reflective generator, so each new interpretation offsets the cost of upgrading.

The simplest (and most obvious) interpretation is as a standard QuickCheck generator:

```
generate :: Reflective b a -> Gen a
generate = interp
  where
    interpR :: R b a -> Gen a
    interpR (Pick  gs) = QC.frequency [(w, interp g) | (w, _, g) <- rs]
    interpR (Lmap _ r) = interpR r
    interpR (Prune  r) = interpR r

    interp :: Reflective b a -> Gen a
    interp (Return a) = return a
    interp (Bind r f) = interpR r >>= interp . f
```

The free monad part of the syntax is implemented as expected, with `Return` implemented as `return` in the `Gen` monad, and `Bind` as the monad's bind. The rest of the syntax similarly straightforward,

with Lmap and Prune doing nothing (remember, those only really run in the backward direction) and Pick interpreted as a weighted random choice.

But the real value of a reflective generator is in its ability to run backward, focusing on sub-parts of a value and reflecting on how they are constructed. This process can be seen using the reflect function below, which interprets a generator to determine which choices lead to a given value:

```
reflect :: Reflective a a -> a -> [[String]]
reflect g = map snd . interp g
  where
    interpR :: R b a -> (b -> [(a, [String])])
    interpR (Pick gs) = \ b -> -- Record choices made.
      concatMap
        ( \ (_, ms, g') ->
            case ms of
              Nothing  -> interp g' b
              Just lbl -> map (\ (a, lbls) -> (a, lbl : lbls)) (interp g' b)
        ) rs
    interpR (Lmap f r) = \ b -> interpR r (f b) -- Adjust b according to f.
    interpR (Prune  r) = \ b -> case b of       -- Filter invalid branches.
      Nothing -> []
      Just a  -> interpR r a

    interp :: Reflective b a -> (b -> [(a, [String])]) -- Reader / Writer Monad interp.
    interp (Return a) = \ _ -> pure (a, [])
    interp (Bind r f) = \ b -> do
      (a,  cs ) <- interpR r b
      (a', cs') <- interp (f a) b
      pure (a', cs ++ cs')
```

Now, Lmap and Prune are not inert. The former applies a backward-direction function to the input that is being assembled, pulling some piece off of it for inspection by a sub-generator. And the latter filters out invalid branches by failing and returning Nothing in the case where the focused part of the structure does not exist. The Return and Bind operations map into a writer monad, threading around a list of recorded choices, and Pick actually records each choice it makes.

These interpretations give the essence of reflective generators, but they are far from the only ones. Moving forward, we will continue to discuss more interpretations, but we will only show interesting parts of interpretations, since they are fairly mechanical. Full interpretation functions can be found in our artifact.

3.4 Theory: Correctness Conditions and Design Decisions

Both interpretations and reflective generators themselves can be written incorrectly. In this section, we describe properties that the programmer should prove or test to gain confidence that their testing tools are correct.

Correctness of Interpretations. We would like reflective generators to obey the laws of monads [Moggi 1991],

(*M*1) `return a >>= f = f a`

(*M*2) `x >>= return = x`

(*M*3) `(x >>= f) >>= g = x >>= (\ a -> f a >>= g)`

of profunctors,

(*P*1) `dimap id id = id  (lmap id = id, rmap id = id)`
(*P*2) `dimap (f' . f) (g . g') = dimap f g . dimap f' g'`

and of PMPs:

(*PMP*1) `lmap Just . prune = id`
(*PMP*2) `lmap (f >=> g) . prune = lmap f . prune . lmap g . prune`
(*PMP*3) `(lmap f . prune) (return y) = return y`
(*PMP*4) `(lmap f . prune) (x >>= g) = (lmap f . prune) x >>= (lmap f . prune) . g`

Some of these laws are definitionally true for all reflective generators, thanks to the structure of freer monads:

Lemma 3.1. *Reflective generators always obey* (*M*1), (*M*3), (*PMP*3), *and* (*PMP*4).

Proof. By induction on the structure of the generator, using the definitions of `return` and `(>>=)` from Kiselyov and Ishii [2015] and the definitions of `lmap` and `prune` from §3.2. □

The other equations do not hold in general, since reflective generators are uninterpreted. Thus, we really want to talk about the required laws in the context of an interpretation.

We say an interpretation of a reflective generator is *lawful* if it implements a PMP homomorphism to some lawful partial monadic profunctor. Concretely:

Definition 1. An interpretation

```
⟦·⟧ :: Reflective b a -> p b a
```

is *lawful* iff `p` obeys the laws of monads, profunctors, and partial monadic profunctors and there exists

```
⟦·⟧R :: R b a -> p b a
```

such that the following equations hold:

```
⟦Return a⟧   = return a
⟦Bind r f⟧   = ⟦r⟧R >>= \ x -> ⟦f x⟧
⟦Lmap f r⟧R = lmap f ⟦r⟧R
⟦Prune  r⟧R = prune ⟦r⟧R
```

One might consider this presentation redundant, since it is possible to define a generic function `interp` that takes an interpretation for `Pick` and gives a full interpretation based on the above equations. Such a function is actually available in our accompanying artifact. But we found this style harder to program against: rather than just write a single interpretation function, the interpretation gets split over half a dozen type-class instances. This presentation allows both styles.

We prove the aforementioned interpretations to be lawful:

Theorem 3.2. *The* `generate` *interpretation is lawful up to distributional equivalence.*

Proof. Even the classic `Gen` "monad" is not actually a lawful monad, but it is lawful up to distributional equivalence [Claessen and Hughes 2000]. Since `Lmap` and `Prune` are both ignored, the other laws are trivial. □

Theorem 3.3. *The* `reflect` *interpretation is lawful.*

Proof. By induction on the structure of the generator. See Appendix A. □

Internal Correctness of a Reflective Generator. Ensuring lawful interpretations will set users up for success, but it is still possible to write an incorrect reflective generator. Thus, we also define properties that say what it means for a reflective generator to be correct and implement those properties as QuickCheck tests.

For these definitions, we use a modified version of `reflect`:

```
reflect' :: Reflective b a -> b -> [a]
```

that does not track choices; it simply captures all the `a` values that can be reached while focusing on different parts of a `b`. Inspired by Xia et al., we define soundness of a reflective generator as follows:

Definition 2. A reflective generator `g` is *sound* iff

```
a ~ generate g ==> (not . null) (reflect' g a).
```

In other words, if the generator interpretation can produce a value, we can reflect on that value to get some set of choices.

This definition is technically correct, but it will occasionally fail (spuriously) if tested using QuickCheck. The problem is *size*: QuickCheck varies the generator's size parameter while testing, but it does not know to vary the size of the `reflect'` interpretation to match. Concretely, this means that QuickCheck may test

```
a ~ resize 100 (generate g) ==> (not . null) (reflect' g a).
```

which is effectively evaluating two different generators. To get around this, a QuickCheck user should instead test

```
a ~ generate (resize n g) ==> (not . null) (reflect' (resize n g) a).
```

for all `n` in a reasonable range.

With soundness done, we also define completeness as follows:

Definition 3. A reflective generator `g` is *complete* iff

```
(not . null) (reflect' g a) ==> a ~ generate g.
```

This property is difficult to test as-is, but Xia et al. give an alternative. First they define *weak completeness*:

Definition 4. A reflective generator `g` is *weak-complete* iff

```
a ∈ reflect' g b ==> a ~ generate g.
```

This property is *compositional*, meaning it is true of a generator if it is true of its sub-generators. Since the only sub-generator reflective generators are built from is `Pick`, we can prove this once-and-for-all:

Lemma 3.4. *All reflective generators are weak complete.*

Proof. By induction on the structure of the generator, see Appendix B. Note that this relies on the weights in every `Pick` being strictly positive. □

Then, the PMP paper gives a so-called *pure projection property*, which is testable (albeit slowly):

Definition 5. A reflective generator satisfies *pure projection* iff

```
a' ∈ reflect' g a ==> a = a'.
```

To complete the picture, we prove the following:

Theorem 3.5. *Any weak-complete reflective generator satisfying the pure projection property above is complete.*

Proof. Assume `(not . null) (reflect g a)`, so there is some `a'` in `reflect g a`. By pure projection, `a = a'` so `a` is in `reflect g a`. then by weak completeness we have `a ~ generate g` as desired. □

Thus, it suffices to test pure projection when checking completeness.

External Correctness of a Reflective Generator. The notions of soundness and completeness above are internal, focused on only the reflective generator itself, but we can also define *external* soundness and completeness with respect to some predicate on the generator's outputs.

We define the following properties:

Definition 6. A reflective generator `g` is *externally sound* with respect to `p` iff

```
x ∈ gen g ==> p x.
```

Definition 7. A reflective generator `g` is *externally complete* with respect to `p` iff

```
p x ==> (not . null) (reflect x g).
```

Unlike the internal soundness and completeness, external soundness and completeness may not necessarily be reasonable to check for every reflective generator. Sometimes there is no external predicate to check against; other times there is a predicate, but the generator is intentionally incomplete. But it is interesting and useful that both of these are *testable*; normal QuickCheck generators cannot test their own completeness.

Fan-Out. One last theoretical property of a reflective generator worth noting is its *fan-out*.

Definition 8. A reflective generator's *fan-out* for a given value is the number of different ways that the value could be produced.

Many reflective generators have a fan-out of 1, meaning that there is only one way to generate any given value, but generators benefit from higher fan-out. For example, a generator might pick between two high-level strategies for generating values that happen to overlap if the combination has a favorable distribution.

Fan-out is important to consider for certain kinds of backward interpretations, since they may need to simultaneously consider all possible choice sequences at once. This may lead to exponential blowup for generators with fan-out greater than 1.

3.5 Writing Reflective Generators

Figure 1 shows the before and after of a QuickCheck generator being upgraded to a reflective one. One could get from point A to point B in a single step, but it may be easier to take a detour through a slightly simpler generator:

```
bst :: (Int, Int) -> Reflective Void Tree
bst (lo, hi) | lo > hi = return Leaf
bst (lo, hi) =
  frequency
    [ ( 1, return Leaf),
      ( 5, do
        x <- fwdOnly (choose (lo, hi))
        l <- fwdOnly (bst (lo, x - 1))
        r <- fwdOnly (bst (x + 1, hi))
```

```
          return (Node l x r) ) ]
```

The type `Reflective Void a` represents a *forward-only* reflective generator. In general, forward interpretations ignore their first type parameter, so for example `gen (bst (-10, 10))` be fine, but backward interpretations need an aligned generator. Under the hood,

```
fwdOnly = lmap (\ x -> case x of)
```

which is rather nice and works for any type.

From there, the programmer can align the type, replace `fwdOnly` annotations with ones that do appropriate focusing, and replace `return` with `exact` where appropriate.

To wrap up, here is one more illustrative example of reflective generators:

Example 3.6 (Simply-Typed Lambda Calculus). We present a full example of a generator for STLC terms in Appendix F, but one aspect of the generator is particularly interesting. At a high level, the generator is made up of generators:

```
type_ :: Reflective Type Type
expr :: Type -> Reflective Expr Expr
```

The generator must first pick a type, and then generate a value of that type. Unfortunately,

```
type_ >>= expr
```

does not type-check, for the same reasons we discuss above (`Reflective`'s first parameter needs adjusted). What we need, is a mapping from `Expr` to `Type` that can focus on the type in the expression. Pleasingly, that focusing is precisely type-checking! The appropriate reflective generator is:

```
comap typeOf type_ >>= expr
```

4 EXAMPLE-BASED GENERATION

In this section, we demonstrate strengths of reflective generators in the context of work that was an early inspiration for their design: example-based generation.

4.1 *Inputs from Hell*

Inputs from Hell [Soremekun et al. 2020] (IFH) describes an approach to random testing that centers around user-provided examples. IFH starts with a set of examples test inputs and randomly produces values that are either similar to or different those examples. The hope is that the similar values represent "common" inputs and that the different ones represent "uncommon" inputs; by testing both of these classes, IFH finds bugs in realistic programs.

The IFH approach is based on *grammar-based generation*, using a context-free grammar representing the program's input format as the basis for producing test inputs. A set of examples provided by the user are parsed by the grammar, producing parse trees, and then those parse trees are used to derive weights for a probabilistic context-free grammar (pCFG) that generates more test inputs. For example, given a simple grammar for numbers

```
num -> "" | digit num;          digit -> "1" | "2" | "3"
```

and the example 12 the IFH technique can derive the following pCFGs:

```
num -> 33% "" | 66% digit num;   digit -> 50% "1" | 50% "2" | 0% "3" (common)
num -> 66% "" | 33% digit num;   digit -> 0% "1" | 0% "2" | 100% "3" (uncommon)
```

Each production is given weight based on the number of times it appears in the parse tree for the provided example (either more or less weight, depending on whether the goal is to generate common or uncommon inputs). The first grammar puts more weight on the 1 and 2, since it is

trying to generate more inputs like the initial example, whereas the second puts more weight on 3 because it is trying to generate inputs *unlike* 12.

This process—parse the input, analyze the parse tree, and re-weight the grammar—is fairly straightforward to implement in the context of grammar-based generation, but the IFH work does not extend to monadic generators. In the next section, we show that reflective generators can bridge this gap.

4.2 Reflecting on Examples

Reflective generators can replicate the ideas in IFH by using a reflective generator for the parsing and generation steps of the IFH algorithm.

Implementation. We use three functions functions—`reflect`, `analyzeWeights`, and `genWithWeights`—that line up with the parsing, analysis, and re-weighting operations for grammars in the IFH paper:

```
reflect :: Reflective a a -> a -> [[String]]
analyzeWeights :: [[String]] -> Weights
genWithWeights :: Reflective b a -> Weights -> Gen a
```

We have already seen `reflect`: it reflects on a generated value and produces a list of choices that were expanded to produce that value. This suggests a view of reflective generators as pretty-printers forward and parsers backward. When run backward, they break the input (string) down and track choices that correspond to productions in a grammar, and when run forward they can make those choices to pretty-print the original string. This works just the same as parsing examples with a CFG, but the backward annotations in the reflective generator allow for far more complex control flow.

Once the examples are parsed, `analyzeWeights` aggregates the choices together to produce a set of weights that say how often to expand one rule versus another. This lets a new "forward" interpretation, `genWithWeights` generate new values by to making choices with the same weights that they had in the examples. The principles here are exactly the same as the ones from IFH, but the replacing the context-free grammar with a reflective generator gives considerably more expressive power.

These interpretations rely heavily on choice labels, which are required in free generators but optional in reflective generators. This means that not all reflective generators work equally well under these interpretations. Concretely, unlabeled choices are invisible to `reflect` and `genWithWeights` so those choices are always made with the weights attached to the `Pick` node (not tuned based on the examples). The user should consider labelling or not labelling choices as a design decision which will impact the distribution of the final generator.

Evaluation. As part of their empirical analysis, Soremekun et al. present a grammar for generating JSON documents and show that the IFH method is effective at testing programs that operate on JSON. We sought to ensure that reflective generators can adequately implement parts of the IFH algorithm necessary to replicate those results.

We implemented a generator for JSON documents as a reflective generator. The structure of the generator, which is shown in full in Appendix C, has the same structure high-level as the grammar used in IFH.[3] Each rule in the grammar became a generator that produces the appropriate string when run forward and selects the parts of the JSON string that each sub-generator is responsible for when run backward.

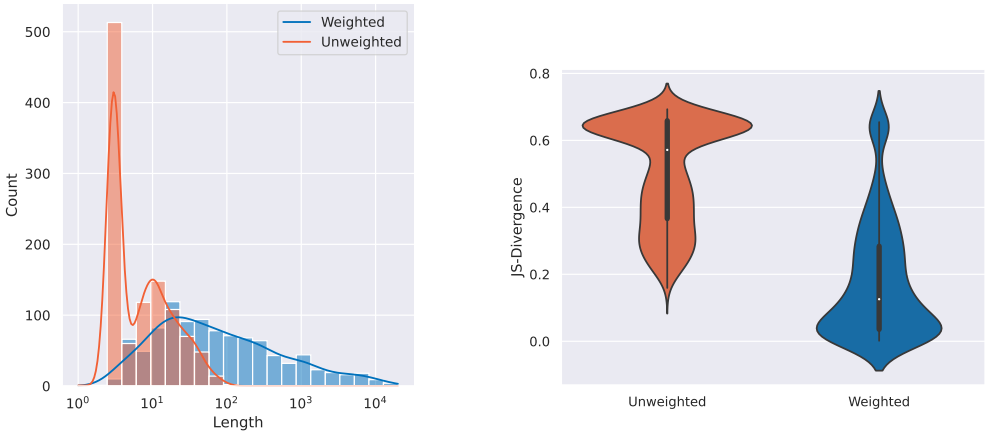
[3]We slightly simplified the grammar from the IFH repository, skipping a few rules for unicode support that were not used by any of the provided examples.

The JSON example is the basis for our evaluation, which answered the following research questions:

RQ1. Can reflective generators be used to produce values that are similar to provided examples, in the style of *Inputs from Hell*?

RQ2. Can reflective generators replicate inputs applicable in situations that the *Inputs from Hell* approach cannot?

To test RQ1, we sampled 10 JSON documents that were used in the full-scale IFH experiments, ranging from 200-1,200 bytes/characters long, and used them to weight a the JSON generator in the style of IFH. We generated 1,000 documents from that weighted generator, as well as from the unweighted generator, and compare the results in Figure 3.



(a) Length distributions of unweighted and weighted JSON generators.

(b) Jensen-Shannon divergence of character distributions. Unweighted vs. Examples and Weighted vs. Examples.

Fig. 3. Analysis of JSON generator tuned by example in the style of *Inputs from Hell*.

Figure 3a demonstrates that the weighted generator is far preferable to the unweighted version in terms of its length distribution. The unweighted distribution, shown in blue, is skewed to the left (smaller inputs) and has a huge spike at 2. Inspecting the data revealed that the length 2 inputs are all either `{}` or `[]`, both uninteresting. In contrast, the weighted generator has a varied length distribution that more closely mimics the examples. It generates very few trivial inputs, instead producing a wide distribution that covers more of the input space.

Figure 3b focuses on the samples' *character distributions*. We counted the occurrences of each character across all 10 of the example documents, resulting in a probability distribution over characters. Then, for each sample, we computed the Jensen-Shannon divergence[4] [Lin 1991] between the example distribution and the character distribution of the sample and plot those divergences in a violin plot. The plots clearly show that the unweighted samples are much farther from the example distribution than the weighted samples.

We conclude that this reflective generators adequately implement the example-replication in the style of IFH, confirming RQ1.

[4]Jensen-Shannon divergence is closely related to the more common Kullback-Leiber (KL) divergence [Kullback and Leibler 1951], but it works better for distributions with differing support because its value is never infinite

Replicating IFH-style example-based generation is a good first step, but reflective generators can do more. In answering RQ2, we explored situations where IFH fails to apply but reflective generators do. In fact, we have already seen one such situation: the `bst` example in Figure 1b. But as a second example we continue the theme of JSON generators. The generator in Figure 4 produces a JSON document that, at the top level, consists of an arbitrary payload object and a hash of that payload. This kind of computation is impossible to capture neatly with an abstraction like the one in IFH, but it is totally natural for reflective generators. It also illustrates that a potential concern with reflective generators—that there is no way to do interesting computations—is not always true. In this case, the hashing is fine because no further choices depend on its value. We revisit non-invertible functions in §8, but it is worth keeping in mind that generators like `withHashcode` are perfectly fine.

```
withHashcode :: Reflective String String
withHashcode = do
  let a = "{\"payload\":"
  let b = ",\"hashcode\":"
  let c = "}"
  consume a >>- \_ ->
    start >>- \payload -> do
      let hashcode = take 8 (show (abs (hash payload)))
      consume b >>- \_ ->
        consume hashcode >>- \_ ->
          consume c >>- \_ ->
            pure (a ++ payload ++ b ++ hashcode ++ c)
  where
    hash = foldl' (\h c -> 33 * h `xor` fromEnum c) 5381
    consume s = lmap (take (length s)) (exact s)
```

Fig. 4. A reflective generator for JSON objects with a hashcode; this could not be easily expressed as a context-free grammar.

In any case, `withHashcode`, `bst`, and other monadic generators with interesting computations and dependencies provide ample evidence that reflective generators generalize the method from IFH.

Discussion. Even if a user is not interested in the full IFH approach—collecting and generating common inputs, generating uncommon inputs, etc.—there is still a lot to gain from the `parse` and `genWithWeights` interpretations of a free generator. In particular, the chart in Figure 3a shows that example-based generation is a promising approach to generator tuning. In simple cases, a user need not spend time fiddling with weights in their to get a reasonable data distribution; they can simply ensure that the generator has important choices labeled, provide the generator with a few canonical examples, and the IFH algorithm can infer weights from there.

5 VALIDITY-PRESERVING SHRINKING AND MUTATION

The example-based generation in the previous illustrates some of the benefits of reflecting on choices, in this section we explore those benefits further, using them to implement input manipulation algorithms like shrinking and mutation. In this section, we discuss the "internal test-case reduction" algorithms implemented in the Hypothesis framework for PBT, show that reflective generators

make these algorithms much more flexible, and finally sketch the ways that these ideas also apply to test-case mutation.

5.1 Test-Case Reduction in Hypothesis

Shrinking is the process of turning a large counter-example into a smaller test case that still triggers a bug. Shrinking is critical because PBT generators often produce very large inputs that are nearly impossible to use for debugging—shrinking makes it much easier to understand the specific aspects of the input that are really necessary to trigger the bug.

In QuickCheck, users can either use GHC's Generics [Magalhães et al. 2010] to derive a shrinker for a given type, or they can write a shrinker by hand. The former is quite effective (as described later in this section) but it is not very general—the shrinkers have only know about type information, so they cannot ensure that the shrunken values satisfy important preconditions nor adequately shrink less structured data. The latter is totally general, but ongoing work has shown preliminary evidence that users find writing shrinkers by hand confusing and error-prone.

Hypothesis's approach to shrinking (which the authors refer to as "internal test-case reduction") is the best of both worlds, solving the generality issue without requiring user effort or understanding [MacIver and Donaldson 2020]. The key insight is that the generator already has all of the information it needs to produce precondition-satisfying inputs, so the generator should be used as part of the shrinking process. The system first generates an input, remembering the random bits that the generator used to create the input, and then it shrinks those random bits to produce smaller values.

```
unlabeled =
  oneof
    [ exact Leaf,
      Branch
        <$> focus (_Branch._1) unlabeled
        <*> focus (_Branch._2) unlabeled
    ]
```

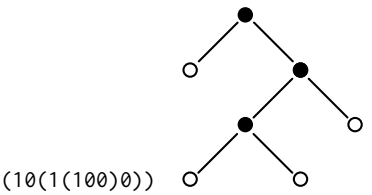


Fig. 5. A Hypothesis-inspired reflective generator and a tree that the generator might produce.

Concretely, Hypothesis represents its input randomness as a bracketed string of bits. For example `(10(1(100)0))` produces the tree in Figure 5. The first bit says to expand the top-level node, the second says that the left-hand subtree is a leaf, and so on. Each level of bracketing delineates some choices that are logically nested together (in this case, on a particular level of the tree). Hypothesis aims to shrink these bitstrings by finding the *shortlex minimum* string of choices that results in a valid counter-example; shortlex order considers shorter strings to be less than longer strings, and follows lexicographic ordering otherwise. In practice, shortlex order is an effective heuristic for final test-case complexity, meaning that smaller bitstrings tend to produce conceptually smaller values in practice.

The actual shrinking procedure uses a number of different passes, each of which attempts to shorten the choice string, swap `1`s with `0`s, or both, resulting in a shortlex-smaller choice string. The passes are described in the Hypothesis paper and available in the open source codebase.

5.2 Reflective Shrinking

The downside of the Hypothesis approach is that the shrinking procedure only works if the random bitstring that produced the target input is still available—without it, there is nothing to shrink. But

there are many reasons one might want to shrink an input that was not recently generated. Most notably, it is often the case that shrinking is useful for understanding externally provided inputs that were not produced by the generator at all; a user might submit a crash report containing a printout of some large program state, and it would be much easier to debug the problem with the help of a shrinker. Similarly, internal shrinking does not work if the input has been modified at all between generation and shrinking, which might be desirable when doing fuzzing-style testing. Luckily, reflective generators can help.

Extracting Bracketed Choices Sequences. We implement *reflective shrinking* with a new interpretation of reflective generators that has the following type:

```
data Choices = Choice Bool | Draw [Choices]
choices :: Reflective a a -> a -> [Choices]
```

The `Choices` type is a rose tree with two types of nodes: `Choice`, which represents a single-bit choice, and `Draw` which represents some nested sequence of choices;[5] this is isomorphic to the bracketed choice sequences that Hypothesis uses. The `choices` function takes a reflective generator and a value in its range, and produces the choice sequences that result in generating that value.

The implementation of `choices` goes similarly to the implementation of `reflect`. That is to say, it performs a "backward" interpretation of the generator, keeping track of choices as it disassembles a value. This interpretation ignores choice labels, since Hypothesis shrinks at a lower level of abstraction. Instead, the interpretation of a `Pick` node determines how many bits would be required to choose a branch (by taking the log of the length of the list of sub generators) and then assigns the appropriate choice sequences to each branch. For example:

```
choices (oneof [exact 1, exact 2, exact 3]) 2 = [Draw [Choice False, Choice True]]
```

This approach is not canonical, but it aligns closely with Hypothesis's implementation, making it easier to accurately replicate their results.

Shrinking Strategies. With an appropriate bracketed choice sequence in hand, shrinking can begin. For our purposes, it sufficed to implement only three of the shrinking passes described in the Hypothesis paper: one pass tries shrinking to a child sequence of the original, a second replaces a `Draw` node with zeroes, and a third swaps ones and zeroes to produce lexically smaller choices strings. The results of `subTrees`, `zeroDraws`, and `swapBits` are shown in Figures 6a, 6b, and 6c accordingly.

```
(10(1(100)0)) => (1(100)0)
                 (100)
```

(a) Shrinks from `subTrees`.

```
(10(1(100)0)) => (10(00000))
                 (10(0000))
                 (10(000))
                 (10(00))
                 (10(0))
                 (10)
```

(b) Shrinks from `zeroDraws`.

```
(10(1(100)0)) => (0111000)
                 (1010100)
```

(c) Shrinks from `swapBits`.

Fig. 6. Shrinking strategies.

[5] The actual type is a bit more complicated; we also cache size information to make shortlex comparisons faster.

	Reflective	Hypothesis*	QuickCheck	Baseline
binheap	9.15 (8.00–10.30)	9.02 (9.01–9.03)	9.14 (8.12–10.16)	14.89 (7.01–22.77)
bound5	3.06 (0.60–5.52)	2.08 (2.07–2.10)	17.75 (0.00–62.32)	131.48 (0.38–262.59)
calculator	5.03 (4.54–5.52)	5.00 (5.00–5.00)	5.07 (4.21–5.92)	13.75 (1.60–25.90)
parser	3.70 (2.21–5.20)	3.31 (3.28–3.34)	3.67 (2.69–4.64)	40.04 (0.00–127.51)
reverse	2.00 (2.00–2.00)	2.00 (2.00–2.00)	2.00 (2.00–2.00)	2.67 (0.76–4.57)

Table 1. Average size of shrunk outputs after reflective shrinking, compared with Hypothesis shrinking, QuickCheck's `genericShrink`, and un-shrunk inputs. (Mean and two standard-deviation range.)
*Hypothesis experiments not re-run, data taken from [MacIver and Donaldson 2020].

Evaluation. To show that reflective shrinkers are a competitive option for shrinking, we asked the following research questions:

RQ1. Do reflective shrinkers shrink inputs at least as effectively as `genericShrink`, the state of the art for automatically-derived shrinkers in Haskell?

RQ2. Do reflective shrinkers shrink inputs at least as effectively as Hypothesis on examples from the paper?

RQ3. Are reflective shrinkers applicable in situations that `genericShrink` and Hypothesis are not?

Our hope was not for reflective shrinkers to be better at shrinking than existing approaches; instead, we simply wanted to ensure that they are at least as good *and* that they can be applied in situations where other options cannot.

To answer RQ1 and RQ2, we replicated one of the experiments from the Hypothesis paper. They borrowed five examples from the SmartCheck repository [Pike 2014] that represent a varied range of shrinking scenarios. Each example comes with a property, a buggy implementation, and a QuickCheck generator; the goal was to run the property to find a counter-example and shrink that counter-example to the smallest possible value.

We upgraded the existing QuickCheck generators to reflective ones, making minor modifications where necessary: we replaced uses of `suchThat` with generators that satisfied invariants constructively, modified some of the approaches to distribution management, and added appropriate reflective annotations. We discuss the motivation for these modifications in detail in §8. Then, we ran each experiment 1,000 times and reported the average size of the resulting counter-examples in Table 1. Note that the QuickCheck and baseline numbers come from the `generate` interpretation of the upgraded reflective generator, rather than the original generator.

These results clearly point to an affirmative answer to RQ1. Reflective shrinkers perform just as well as QuickCheck's `genericShrink` in all cases, and significantly better in `bound5`.

With a few caveats, reflective shrinkers also match Hypothesis. They exhibit a higher variance in the size of counter-examples that they produce, likely because they only implement a subset of Hypothesis's shrinking strategies, but nevertheless their counter-examples are on average within 1 unit of Hypothesis (and usually much closer). The worst experiment relative to Hypothesis is `bound5`; in that example, we suspect the difference is due to differing strategies for generating integers, rather than anything to do with shrinking directly. Thus, we can also give a qualified "yes" to answer RQ2.

For RQ3, we return to a modified version of the JSON example used in §4. Consider a program that processes dependencies in "`package.json`" files, which are used as a configuration format in Node.js. Suppose the program behaves incorrectly when the file specifies a specific version of a specific package, but the bug has eluded the developers thus far. The program is tested using PBT,

and the developers have a `package.json` reflective generator that they use for testing, but the bug in question has not yet been found that way.

When user approaches the developers with a file that causes the dependency processor to crash (shown in Appendix E), the developers can use their reflective shrinker to reduce it to a far simpler file. The new file (also in Appendix E) has only one non-trivial field: the one causing the bug. The developers can then use this information to find the problem much more quickly than before.

This situation would not have been possible with either `genericShrink` or Hypothesis. The former would not work because the format is too unstructured: the generator produces JSON strings, rather than a Haskell data type, so the best the shrinker could do is shorten the string (which would result in invalid JSON). The latter could not even start to shrink, since the JSON file came from a user, and therefore there is no random bitstring to shrink. In this way, we have demonstrated that, RQ3 can also be answered in the affirmative.

5.3 Reflective Mutation

HypoFuzz, a tool for doing coverage-guided fuzzing [Fioraldi et al. 2020] of PBT properties, is a newer and lesser-known part of the Hypothesis ecosystem. Similarly to Crowbar [Dolan and Preston 2017], HypoFuzz uses a PBT generator to aid the fuzzer. The fuzzer provides random choices that the generator can use to build inputs.

Fuzzers try to maximize code coverage by keeping track of a set of interesting inputs and *mutating* them, attempting to explore similar values and hopefully continue to cover new branches of the program. This mutation can be difficult, since naïve mutations will rarely produce values that are valid inputs to the program, but HypoFuzz gets around the concern with the same trick Hypothesis uses for fuzzing: mutate the randomness, not the value.

Internal mutation has all of the same benefits and drawbacks as internal shrinking. On one hand, it is type agnostic, easy to use, and guarantees validity of the mutated values. On the other hand, it assumes that the randomness used to produce a given value is available. It may seem like this drawback is less of an issue for mutation than it is for shrinking, since the fuzzer can just keep track of the random choices associated with each value it wants to mutate, but this is not true of the initial set of *seed inputs*. For optimal fuzzing, the seeds are provided by the user and represent some set of initially interesting values that the fuzzer can play with. This does not work with Hypothesis-style mutation!

Once again, reflective generators provide a compelling solution. Simply extract a choice sequence from each seed using the `choices` interpretation and start mutating in much the same way that Hypothesis does shrinking.

6 IMPROVING THE TESTING WORKFLOW

So far we have seen reflective generators in the context of example-based generation, shrinking, and mutation. But reflective generators are even more flexible than that; both their forward and backward directions have a wealth of uses throughout the testing process. In this section, we explore a few more of those use-cases, demonstrating reflective generators' value and flexibility.

6.1 Reflective Checkers

Reflective generators go far beyond the "bigenerators" discussed by Xia et al. in their original work on PMPs, but reflective generators can indeed be used as bigenerators if needed.

Recall that the backward direction of a bigenerator is simpler than the backward direction of a reflective generator. Rather than reflect on choices, a bigenerator simply checks whether a value is in the range of the generator, effectively checking if the value satisfies the invariant that the generator enforces. Reflective generators can do this too. They can simply reflect on the generator's

choices and ask whether or not there exists a set of choices that results in the desired value. If there is a set of choices, the value must satisfy the generator's precondition; if not, it must not. Of course, this only makes sense if there is no ground-truth predicate against which to test *external soundness*, but in that case it saves the programmer the trouble of writing that predicate.

Furthermore, a reflective generator can actually track the *probability* of generating a particular value with the `generate` interpretaion. We implement this in our artifact as a backward interpretation, `probabilityOf`, which tracks the different ways of generating a particular value and the likelihood of choosing those different ways. Obviously this may run slowly for generators with high fan-out, but it can still be quite useful for understanding how realistic it is to expect the generator to produce certain shapes of values.

6.2 Reflective Completers

A more unique use-case for reflective generators is generation based on a *partial value*. As a simple example, imagine a binary search tree with holes:

```
Node (Node _ 1 _) 5 _
```

Reflective generators provide a way to *randomly complete* a value like this, filling the holes with appropriate randomly generated values:

```
Node (Node Leaf 1 Leaf) 5 Leaf
Node (Node Leaf 1 (Node Leaf 3 Leaf)) 5 Leaf
Node (Node Leaf 1 (Node Leaf 3 Leaf)) 5 (Node (Node Leaf 6 Leaf) 7 Leaf)
```

This technique lets the user pick out a sub-space of a generator, defined by some value prefix, and explore that sub-space while maintaining any preconditions that generator enforces.

We accomplish this with some small, but reasonable, hacks. A partial value is represented directly as a value containing `undefined`, so the value above would look like:

```
Node (Node undefined 1 undefined) 5 undefined
```

Due to laziness, there is nothing wrong with this computing this value, but attempts to match on the undefined values will crash. Suppose, then, that this value were passed into a backward interpretation of `bst` from §1—when does the program fail?

The key insight is that the only place a reflective generator manipulates its focused value is when re-focusing. In other words, the only place a backward interpretation would crash on a partial value is while interpreting `Lmap`. Capitalizing on this insight, we wrap the standard `Lmap` interpretation in a call to `catch`, Haskell's limited mechanism for exception handling:

```
complete :: Reflective a a -> a -> IO (Maybe a)
...
  interpR (Lmap f x) b =
    catch
      (evaluate (f b) >>= interpR x)
      (\(_ :: SomeException) -> (: []) <$> QC.generate (generate (Bind x Return)))
```

If no exception occurs, the program continues as planned, but, if there is an exception, the current value is abandoned and the rest of the value is generated via the `generate` interpretation. This means that `complete` mixes both backward and forward interpretation to achieve its goals.

Of course, this approach does rely on laziness. While it is powerful and works well for structural generators that only do shallow pattern matching in `Lmaps`, things fall apart if the backward direction needs to evaluate the whole term. The clearest example of this is `comap typeOf type_ >>= expr` (recall, it generates a type and then a program of that type); in the backward direction, this generator

immediately evaluates the whole term to compute its type. For this generator, `complete` would just generate a totally fresh program.

Users can work around this—for example, `comap optimisticTypeOf type_ >>= expr`, successfully completes the value `App (Ann (Int :-> Int) undefined) (Ann Int undefined)` by trusting the type annotations, picking the type `Int` for the expression without evaluating the `undefined`s, and then completing the sub-expressions.

6.3 Reflective Producers

When PBT started in QuickCheck, random generators were the standard way to get test inputs. But weighted random generation in the style of QuickCheck is not the only option anymore: both enumeration and guided generation have appeared as powerful alternatives. Indeed, much of the literature has moved from talking about *generators* to talking about *producers* of test data, when the specific strategy does not matter [Paraskevopoulou et al. 2022; Soremekun et al. 2020]. We still use the language of "generators" because it is familiar and concrete, but reflective generators are flexible enough to be considered *reflective producers* instead.

A reflective generator can be made into an enumerator by simply re-interpreting `Pick` as an exhaustive choice rather than a random one. We implement an interpretation

```
enumerate :: Reflective b a -> [[a]]
```

leaning heavily on the combinators and techniques used by LeanCheck [Braquehais 2017] for roughly size-based enumeration. We say "roughly" because whereas LeanCheck enumerators allow the user to define their own notion of size for each enumerator, reflective generators are limited to a single notion of size based on the number and order of choices needed to produce a given value. A thorough evaluation of this discrepancy would require its own study (see §9), but early observations are promising. For example, `enumerate (bst (1, 10))` reaches size-4 BSTs within just 10 and matches the size order of an idiomatic LeanCheck enumerator given in Appendix D.

While fuzzing is generally considered separate from PBT, focusing on finding crash failures by generating inputs external to a system rather than finding subtle logic errors in individual functions, many modern projects are attempting to bridge the gap. We already saw that reflective mutators are helpful in the context of HypoFuzz-style mutation, but reflective generators can also be used to interface with an external fuzzer in the style of Crowbar [Dolan and Preston 2017], which is designed to get its inputs from popular fuzzers like AFL or AFL++ [noa 2022; Fioraldi et al. 2020]. Since Crowbar already uses a free-monad-like structure to represent its generators, we can write an equivalent reflective generator interpretation that works the same way. More generally, reflective generators can be used in any producer algorithm that relies on the generator-as-parser perspective.

7 RELATED WORK

This work is primarily built on results relating to free generators [Goldstein and Pierce 2022] and partial monadic profunctors [Xia et al. 2019]. Free generators are, in turn, built on top of freer monads [Kiselyov and Ishii 2015], which were initially invented as a way better way to represent effectful code in pure languages. While our implementation remains faithful to the most basic conception of freer monads, there are many insights from Kiselyov and Ishii that we have not yet explored. Likewise, PMPs are part of the long tradition of bidirectional programming [Foster 2009], and it remains to be seen if there are stronger ways to tie reflective generators to work on lenses and other bidirectional abstractions.

The concrete realization of reflective generators is also similar to the implementation of Crowbar [Dolan and Preston 2017]. Both libraries use a syntactic, uninterpreted representation for

generators, although the Crowbar version does not incorporate any ideas from monadic profunctors and uses different type for bind that does not normalize as aggressively.

Reflective generators were originally designed to replicate the tools developed in *Inputs from Hell* [Soremekun et al. 2020], and those tools tie into the broader world of grammar-based generation [Aschermann et al. 2019; Godefroid et al. 2017; Holler et al. [n. d.]; Srivastava and Payer 2021; Veggalam et al. 2016; Wang et al. 2019]. Grammar-based approaches are necessarily less expressive than monadic ones, since they can only generate strings in a context-free grammar, and therefore cannot generate complex data structures with internal dependencies. Still, there may be other results from grammar-based generation that can be transported to monadic generators via tools like reflective generators.

Replicating example distributions is a classic problem in the space of *probabilistic programming* [Gordon et al. 2014]. While the goals of probabilistic programs are usually quite different from those of PBT generators, there is some overlap in the formalisms used to express these ideas. In particular, one representation of probabilistic programs in the functional programming literature [Ścibior et al. 2018] uses a free monad that is similar to free and reflective generators.

Reflective shrinking and mutation build heavily on work done in the Hypothesis framework in Python [MacIver and Donaldson 2020; MacIver et al. 2019], but Hypothesis-style internal test-case reduction is not the only approach to automated shrinking. As discussed briefly in §5, QuickCheck provides `genericShrink`, which provides a competent shrinker for any Haskell type that implements `Generic`. We note above that while `genericShrink` is a nice starting point, it fails to shrink unstructured data (like strings) and values with complex preconditions. Yet another alternative is provided by Hedgehog [Stanley [n. d.]]. Hedgehog is a QuickCheck competitor in Haskell, and implements automatic shrinking by combining shrinking with generation. Library-provided generators contain information about how to shrink the values they generate, and composite generators rely on sub-generators to shrink sub-structures. This approach is morally related to reflective shrinking, but it is not clear that there are deep theoretical similarities. In any case, Hedgehog shrinking is also impressive, but it requires programmer effort and also does not handle values with complex preconditions.

8 LIMITATIONS

Reflective generators are powerful, but they are not quite as powerful as arbitrary monadic generators. In particular, they cannot express generators whose choices depend on functions that are fundamentally unidirectional. For example, there is no way to turn a QuickCheck generator that computes an un-invertible hash and makes a choice based on it, since there would be no way to recover the original value from the hash to analyze the choices made when producing that value. This is certainly a downside, and it is hard say for sure how many generators this rules out, but in practice reflective generators have considerable expressive power. They can generate and reflect on structures as complex as well-typed terms in the polymorphic lambda calculus [Reynolds 1974] (see Appendix F), which is known to be one of the more complicated kinds of structures to express [Pałka et al. 2011], so we are confident that they can tackle a significant subset of the generators that PBT users write.

A slightly subtler limitation is that reflective generators cannot generate values that do not end up as part of the final structure in some way. This is not a common occurrence but two such case came up in the course of our case studies. First, QuickCheck has a generator called `suchThat` that samples a value, and, if it does not satisfy a some predicate, throws it away and samples another. Even this is technically fine, since the backward direction should never actually need to run the generator more than once, but the real version of `suchThat` also slightly increases the generator's size parameter with every iteration. In order to correctly reflect on a value, the backward direction would need

to keep trying generators, recording and throwing away choices, until one succeeds; as far as we know this is not possible with the current structure. The solution is simply to avoid `suchThat` in favor of generators that satisfy predicates constructively—this would be our recommendation anyway, since `suchThat` can be extremely slow in complex cases.

A second case reflective generators struggle to replicate is using a generated integer to bias generation in some way. For example, some generators we encountered generate an integer, and based on that decide what else to generate. This is problematic because there is no way to recover that integer in the backward direction. Luckily, this should never be necessary—the weights on the `Pick` nodes do the same job.

9 CONCLUSION AND FUTURE DIRECTIONS

Reflective generators are a powerful abstraction for test input manipulation and production. We have shown their utility in a variety of testing scenarios, including example-based generation, shrinking, mutation, precondition checking, value completion, enumeration, fuzzing, and more. We plan to use reflective generators as a foundation for a larger research program for improving PBT tooling, and we are excited to see where that program takes us. In this section, we describe a few ideas for further work on reflective generators.

9.1 Automation and Synthesis of Annotations

The `Lmap` annotations in reflective generators can be arbitrarily complex, but in practice they are usually simple, predictable functions that operate on the input's structure. We are optimistic that, in a large variety of cases, the annotations could be synthesized automatically.

We plan to work with Hoogle+ [James et al. 2020], using its type-based synthesis algorithm to synthesize candidate programs for the annotations without any user intervention. This is an especially compelling opportunity because it is easy to tell when the annotations are correct: they are sound and complete, as described in §3.4. When synthesizing multiple annotations at the same time, the system can even use the number of examples that pass or fail the soundness and completeness properties as a way to infer which annotations are correct, and which need to be re-synthesized.

If this algorithm is successful, it will make transitioning from QuickCheck generators to reflective generators almost entirely automatic, and this approach may suggest other opportunities for program synthesis for bidirectional programs.

9.2 Usability of Reflective Generators

We have taken great care to design an API for reflective generators that aligns with existing QuickCheck functions and minimizes programmer effort. Our experience writing reflective generators for our case studies was very positive, and, barring the aforementioned limitations (§8), ran into no issues upgrading existing generators. The automation techniques hypothesized above should make reflective generators even more usable. Still, the authors of this paper certainly do not constitute a representative sample of PBT users, so the usability of reflective generators remains to be empirically studied.

Luckily, there is a growing push in the PL community to incorporate ideas and techniques from human-computer interaction (HCI) [Chasins et al. 2021], and this is a perfect opportunity to join that movement. We plan to collaborate with HCI researchers to do a thorough usability analysis of reflective generators. Inspired by prior work [Coblenz et al. 2021], our analysis will be used to both refine and assess the design of the language, evaluating its usability in a vacuum (relative to standard QuickCheck generators) and its usability in the context of a realistic testing task.

9.3 Reflective Generators as an Enabling Technology

We envision a new generation of PBT tools, powered by reflective generators. Their design is influenced by and compatible with current tooling, they enable generalized algorithms that are more useful in realistic testing scenarios, and they are flexible enough to support multiple test-case production paradigms at once. But getting there will require significant engineering work, supported by strong experimental evaluation.

We plan to grow reflective generators into a model for a production-ready PBT framework. We will build on the shrinkers and mutators from §5, implementing the full suite of Hypothesis shrinking and mutation strategies. We will optimize those shrinkers and mutators for performance and look for opportunities to improve on the choice-manipulation strategies in the context of the kinds of data structures that functional programmers work with. We will also build on the example-based generators, completers, and producers throughout the rest of the paper, evaluating them against the state of the art; in particular, we hope to be able to make suggestions about when a user should pick one producer interpretation over another for optimal testing.

With a model implementation in place, we then hope to to implement reflective generators in a number of popular PBT frameworks, including Hypothesis, OCaml's `Base_Quickcheck`, and more. Reflective generators will bring flexibility and power to the next generation of PBT tools.

REFERENCES

2022. American Fuzzy Lop (AFL). https://github.com/google/AFL original-date: 2019-07-25T16:50:06Z.

Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Proceedings 2019 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2019.23412

Rudy Matela Braquehais. 2017. Tools for Discovery, Refinement and Generalization of Functional Properties by Enumerative Testing. (Oct. 2017). http://etheses.whiterose.ac.uk/19178/ Publisher: University of York.

Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine. 2021. PL and HCI: better together. *Commun. ACM* 64, 8 (Aug. 2021), 98–106. https://doi.org/10.1145/3469279

Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. 2018. Functional programming for modular Bayesian inference. *Proceedings of the ACM on Programming Languages* 2, ICFP (July 2018), 83:1–83:29. https://doi.org/10.1145/3236778

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

Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2021. PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design. *ACM Transactions on Computer-Human Interaction* 28, 4 (July 2021), 28:1–28:53. https://doi.org/10.1145/3452379

Zac Hatfield Dodds. 2022. current maintainer of Hypothesis (https://github.com/HypothesisWorks/hypothesis). Personal communication.

Stephen Dolan and Mindy Preston. 2017. Testing with crowbar. In *OCaml Workshop*. https://github.com/ocaml/ocaml.org-media/blob/master/meetings/ocaml/2017/extended-abstract__2017__stephen-dolan_mindy-preston__testing-with-crowbar.pdf

Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++} : Combining Incremental Steps of Fuzzing Research. https://www.usenix.org/conference/woot20/presentation/fioraldi

John Nathan Foster. 2009. *Bidirectional programming languages*. Ph.D. University of Pennsylvania, United States – Pennsylvania. https://www.proquest.com/docview/304986072/abstract/11884B3FBDDB4DCFPQ/1 ISBN: 9781109710137.

Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 50–59. https://dl.acm.org/doi/10.5555/3155562.3155573

Harrison Goldstein and Benjamin C. Pierce. 2022. Parsing Randomness. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (Oct. 2022), 128:89–128:113. https://doi.org/10.1145/3563291

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic programming. In *Future of Software Engineering Proceedings (FOSE 2014)*. Association for Computing Machinery, New York, NY, USA, 167–181. https://doi.org/10.1145/2593882.2593900

Christian Holler, Kim Herzig, and Andreas Zeller. [n. d.]. Fuzzing with Code Fragments (-2). ([n. d.]).

Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. 2020. Digging for fold: synthesis-aided API discovery for Haskell. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 205:1–205:27. https://doi.org/10.1145/3428273

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. *ACM SIGPLAN Notices* 50, 12 (2015), 94–105. https://dl.acm.org/doi/10.1145/2804302.2804319 Publisher: ACM New York, NY, USA.

S. Kullback and R. A. Leibler. 1951. On Information and Sufficiency. *The Annals of Mathematical Statistics* 22, 1 (1951), 79–86. https://www.jstor.org/stable/2236703 Publisher: Institute of Mathematical Statistics.

J. Lin. 1991. Divergence measures based on the Shannon entropy. *IEEE Transactions on Information Theory* 37, 1 (Jan. 1991), 145–151. https://doi.org/10.1109/18.61115 Conference Name: IEEE Transactions on Information Theory.

David R. MacIver and Alastair F. Donaldson. 2020. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer (Tool Insights Paper). In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 13:1–13:27. https://doi.org/10.4230/LIPIcs.ECOOP.2020.13 ISSN: 1868-8969.

David R MacIver, Zac Hatfield-Dodds, and others. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891. https://joss.theoj.org/papers/10.21105/joss.01891.pdf

José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. 2010. A generic deriving mechanism for Haskell. *ACM SIGPLAN Notices* 45, 11 (Sept. 2010), 37–48. https://doi.org/10.1145/2088456.1863529

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (July 1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. ACM, New York, NY, USA, 91–97. https://doi.org/10.1145/1982595.1982615 event-place: Waikiki, Honolulu, HI, USA.

Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. 2022. Computing correctly with inductive relations. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 966–980. https://doi.org/10.1145/3519939.3523707

M. Pickering, J. Gibbons, and N. Wu. 2017. Profunctor optics: Modular data accessors. *Art, Science, and Engineering of Programming* 1, 2 (2017). https://ora.ox.ac.uk/objects/uuid:9989be57-a045-4504-b9d7-dc93fd508365 Publisher: Aspect-Oriented Software Association.

Lee Pike. 2014. SmartCheck: automatic and efficient counterexample reduction and generalization. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell (Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 53–64. https://doi.org/10.1145/2633357.2633365

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974 (Lecture Notes in Computer Science, Vol. 19)*, Bernard Robinet (Ed.). Springer, 408–423. https://doi.org/10.1007/3-540-06859-7_148

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). https://doi.org/10.1109/TSE.2020.3013716 Publisher: IEEE.

Prashast Srivastava and Mathias Payer. 2021. Gramatron: effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 244–256. https://doi.org/10.1145/3460319.3464814

Jacob Stanley. [n. d.]. Hedgehog will eat all your bugs. https://hedgehog.qa/

Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. In *Computer Security – ESORICS 2016 (Lecture Notes in Computer Science)*, Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows (Eds.). Springer International Publishing, Cham, 581–601. https://doi.org/10.1007/978-3-319-45744-4_29

Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 724–735. https://doi.org/10.1109/ICSE.2019.00081 ISSN: 1558-1225.

Li-yao Xia, Dominic Orchard, and Meng Wang. 2019. Composing bidirectional programs monadically. In *European Symposium on Programming*. Springer, 147–175. https://doi.org/10.1007/978-3-030-17184-1_6

APPENDIX

A PROOFS OF LEMMA 3.1 (LAWS)

Recall the equations from Lemma 3.1.

($M1$) `return a >>= f = f a`
($M3$) `(x >>= f) >>= g = x >>= (\ a -> f a >>= g)`
($PMP3$) `(lmap f . prune) (return y) = return y`
($PMP4$) `(lmap f . prune) (x >>= g) = (lmap f . prune) x >>= lmap f . prune . g`

Proofs of ($M1$) and ($M3$). Immediate, by definition. □

Proof of ($PMP3$). By rewriting.

```
(lmap f . prune) (return y)
  = (lmap f . prune) (Return y)
  = lmap f (Return y)
  = Return y
  = return y
```

Thus ($PMP3$) holds. □

Proof of ($PMP4$). By induction over the structure of x.

Case `x = Return a`:

```
(lmap f . prune) (Return a >>= g)
  = (lmap f . prune) (g a)
  = Return a >>= (lmap f . prune) . g
  = (lmap f . prune) (Return a) >>= lmap f . prune . g
```

Case `x = Bind r h`:

```
(lmap f . prune) (Bind r h >>= g)
  = (lmap f . prune) (Bind r (h >=> g))
  = Bind (Lmap f (Prune r)) (lmap f . prune . (h >=> g))
  = Bind (Lmap f (Prune r)) (lmap f . prune . h >=> lmap f . prune . g)
  = (Bind (Lmap f (Prune r)) (lmap f . prune . h)) >>= (lmap f . prune . g)
  = (lmap f . prune) (Bind r h) >>= lmap f . prune . g
```

Thus ($PMP4$) holds. □

B PROOF OF LEMMA 3.4 (WEAK COMPLETENESS)

Recall that a reflective generator g is weak-complete iff

```
a ∈ reflect' g b ==> a ~ generate g.
```

We claim that every reflective generator is weak-complete.

Proof. By mutual induction over the structure of `Freer` and `R`.

Given a reflective generator `g` and a value `a`:

Case `g = Return a'`:
Assume `a ∈ reflect' (Return a') b = [a']`, thus `a = a'`.
By definition, `a' ~ return a'`, so `a ~ return a' = reflect' (Return a')`.

Case `g = Bind r f`:
Assume `a ∈ reflect' (Bind r f) b`.
Thus, `a ∈ (interpR`$_{\texttt{reflect'}}$ `r b >>= \ α -> reflect' (f α) b)`.
Thus, ∃`a'` such that `a' ∈ interpR`$_{\texttt{reflect'}}$ `r b` and `a ∈ reflect' (f a') b`.
By IH$_R$, `a' ~ interpR`$_{\texttt{generate}}$ `r`.
By IH, `a ~ generate (f a')`.
Thus, `a ∈ (interpR`$_{\texttt{generate}}$ `r >>= \ α -> generate (f α))`.
Thus, `a ∈ generate (Bind r f)`.

Simultaneously, given an `R r` and a value `a`:

Case `r = Lmap f r'`:
Assume `a ∈ interpR`$_{\texttt{reflect'}}$ `(Lmap f r') b`.
Thus, `a ∈ interpR`$_{\texttt{reflect'}}$ `r' (f b)`.
By IH$_R$, `a ~ interpR`$_{\texttt{generate}}$ `r'`.
Thus, `a ~ interpR`$_{\texttt{generate}}$ `(Lmap f r')`.

Case `r = Prune r'`:
Assume `a ∈ interpR`$_{\texttt{reflect'}}$ `(Prune r') b`.
Thus, `a ∈ interpR`$_{\texttt{reflect'}}$ `r' (f b)`.
By IH$_R$, `a ~ interpR`$_{\texttt{generate}}$ `r'`.
Thus, `a ~ interpR`$_{\texttt{generate}}$ `(Prune r')`.

Case `r = Pick gs`:
Assume `a ∈ interpR`$_{\texttt{reflect'}}$ `(Pick gs) b`.
Thus, `a ∈ (gs >>= \ (_, _, γ) -> reflect' γ b)`.
Thus, ∃`g'` such that `(_, _, g') ∈ gs` and `a ∈ reflect' g' b`.
By IH, `a ~ generate g'`.
Thus, `a ~ QC.frequency [(w, interp γ) | (w, _, γ) <- gs]`. (Recall, we assume weights are positive.)
Thus, `a ~ interpR`$_{\texttt{generate}}$ `(Pick gs)`.

This completes the proof. □

C GENERATOR FOR JSON DOCUMENTS

```
token :: Char -> Reflective b ()
token s = labelled [(['\'', s, '\''], pure ())]

label :: String -> Reflective b ()
label s = labelled [(s, pure ())]

(>>-) :: Reflective String String
      -> (String -> Reflective String String)
      -> Reflective String String
p >>- f = do
  x <- p
  lmap (drop (length x)) (f x)

-- start = array | object ;
start :: Reflective String String
start =
  labelled
    [ ("array", array),
      ("object", object)
    ]

-- object = "{" "}" | "{" members "}" ;
object :: Reflective String String
object =
  labelled
    [ ("'{' '}'", lmap (take 2) (exact "{}")),
      ( "'{' members '}'",
        lmap (take 1) (exact "{") >>- \b1 ->
          members >>- \ms ->
            lmap (take 1) (exact "}") >>- \b2 ->
              pure (b1 ++ ms ++ b2)
      )
    ]

-- members = pair | pair ',' members ;
members :: Reflective String String
members =
  labelled
    [ ("pair", pair),
      ( "pair ',' members",
        pair >>- \p ->
          lmap (take 1) (exact ",") >>- \c ->
            members >>- \ps ->
              pure (p ++ c ++ ps)
      )
    ]
```

```
  -- pair = string ':' value ;
  pair :: Reflective String String
  pair =
    string >>- \s ->
      lmap (take 1) (exact ":") >>- \c ->
        value >>- \v ->
          pure (s ++ c ++ v)

  -- array = "[" elements "]" | "[" "]" ;
  array :: Reflective String String
  array =
    labelled
      [ ("'[' ']'", lmap (take 2) (exact "[]")),
        ( "'[' elements ']'",
          lmap (take 1) (exact "[") >>- \b1 ->
            elements >>- \ms ->
              lmap (take 1) (exact "]") >>- \b2 ->
                pure (b1 ++ ms ++ b2)
        )
      ]

  -- elements = value ',' elements | value ;
  elements :: Reflective String String
  elements =
    labelled
      [ ("value", value),
        ( "value ',' elements",
          value >>- \el ->
            lmap (take 1) (exact ",") >>- \c ->
              elements >>- \es ->
                pure (el ++ c ++ es)
        )
      ]

  -- value = "f" "a" "l" "s" "e" | string | array | "t" "r" "u" "e" | number | object | "n" "u" "l" "l" ;
  value :: Reflective String String
  value =
    labelled
      [ ("false", lmap (take 5) (exact "false")),
        ("string", string),
        ("array", array),
        ("number", number),
        ("true", lmap (take 4) (exact "true")),
        ("object", object),
        ("null", lmap (take 4) (exact "null"))
      ]
```

```
  -- string = "\""  "\"" | "\""  chars  "\""  ;
  string :: Reflective String String
  string =
    labelled
      [ ("'\"' '\"'", lmap (take 2) (exact "\"\"")),
        ( "'\"' chars '\"'",
          lmap (take 1) (exact ['"']) >>- \q1 ->
            chars >>- \cs ->
              lmap (take 1) (exact ['"']) >>- \q2 ->
                pure (q1 ++ cs ++ q2)
        )
      ]

  -- chars = char_  chars | char_ ;
  chars :: Reflective String String
  chars =
    labelled
      [ ("char_", (: []) <$> focus _head char_),
        ("char_ chars", (:) <$> focus _head char_ <*> focus _tail chars)
      ]

  -- char_ = digit | unescapedspecial | letter | escapedspecial ;
  char_ :: Reflective Char Char
  char_ =
    labelled
      [ ("letter", letter),
        ("digit", digit),
        ("unescapedspecial", unescapedspecial),
        ("escapedspecial", escapedspecial)
      ]

  letters :: [Char]
  letters = ['a' .. 'z'] ++ ['A' .. 'Z']

  -- letter = "a" | .. | "z" | "A" | .. | "Z"
  letter :: Reflective Char Char
  letter = labelled (map (\c -> ([c], exact c)) letters)

  unescapedspecials :: [Char]
  unescapedspecials = ['/', '+', ':', '@', '$', '!', '\'', '(', ',', '.', ')', '-', '#', '_']

  -- unescapedspecial = "/" | "+" | ":" | "@" | "$" | "!" | "'" | "(" | "," | "." | ")" | "-" | "#" | "_"
  unescapedspecial :: Reflective Char Char
  unescapedspecial = labelled (map (\c -> ([c], exact c)) unescapedspecials)

  escapedspecials :: [Char]
  escapedspecials = ['\b', '\n', '\r', '\\', '\t', '\f']
```

```
-- escapedspecial = "\\b" | "\\n" | "\\r" | "\\/" | "\\\\" | "\\t" | "\\\"" | "\\f" ;
escapedspecial :: Reflective Char Char
escapedspecial = labelled (map (\c -> ([c], exact c)) escapedspecials)

-- number = int_ frac exp | int_ frac | int_ exp | int_ ;
number :: Reflective String String
number =
  labelled
    [ ("int_", int_),
      ("int_ exp", int_ >>- \i -> expo >>- \ex -> pure (i ++ ex)),
      ("int_ frac", int_ >>- \i -> frac >>- \f -> pure (i ++ f)),
      ("int_ frac exp", int_ >>- \i -> frac >>- \f -> expo >>- \ex -> pure (i ++ f ++ ex))
    ]

-- int_ = nonzerodigit digits | "-" digit digits | digit | "-" digit ;
int_ :: Reflective String String
int_ =
  labelled
    [ ("nonzero digits", (:) <$> focus _head nonzerodigit <*> focus _tail digits),
      ("digit", (: []) <$> focus _head digit),
      ( "'-' digit",
        (\x y -> x : [y])
          <$> focus _head (exact '-')
          <*> focus (_tail . _head) digit
      ),
      ("'-' digit digits", (:) <$> focus _head (exact '-')
            <*> focus _tail ((:) <$> focus _head digit <*> focus _tail digits))
    ]

-- frac = "." digits ;
frac :: Reflective String String
frac = label "'.' digits" >> (:) <$> focus _head (exact '.') <*> focus _tail digits

-- exp = e digits ;
expo :: Reflective String String
expo =
  label "e digits"
    >> ( e >>- \e' ->
           digits >>- \d ->
             pure (e' ++ d)
       )

-- digits = digit digits | digit ;
digits :: Reflective String String
digits =
  labelled
    [ ("digit", (: []) <$> focus _head digit),
      ("digit digits", (:) <$> focus _head digit <*> focus _tail digits)
```

```
      ]

  -- digit = nonzerodigit | "0" ;
  digit :: Reflective Char Char
  digit =
    labelled [("nonzerodigit", nonzerodigit), ("'0'", exact '0')]

  -- nonzerodigit = "3" | "4" | "7" | "8" | "1" | "9" | "5" | "6" | "2" ;
  nonzerodigit :: Reflective Char Char
  nonzerodigit =
    labelled (map (\c -> ([c], exact c)) ['1', '2', '3', '4', '5', '6', '7', '8', '9'])

  -- e = "e" | "E" | "e" "-" | "E" "-" | "E" "+" | "e" "+" ;
  e :: Reflective String String
  e =
    labelled
      [ ("'e'", lmap (take 1) (exact "e")),
        ("'E'", lmap (take 1) (exact "E")),
        ("'e-'", lmap (take 2) (exact "e-")),
        ("'E-'", lmap (take 2) (exact "E-")),
        ("'e+'", lmap (take 2) (exact "e+")),
        ("'E+'", lmap (take 2) (exact "E+"))
      ]
```

D LEANCHECK BST ENUMERATOR

```
leanBST :: (Int, Int) -> [[Tree]]
leanBST (lo, hi) | lo > hi = [[Leaf]]
leanBST (lo, hi) =
  cons0 Leaf
    \/ ( choose (lo, hi) >>- \ x ->
           leanBST (lo, x - 1) >>- \ l ->
             leanBST (x + 1, hi) >>- \ r ->
               delay [[Node l x r]]
       )
  where
    (>>-) = flip concatMapT
    choose = concatT [zipWith (\i x -> [[x]] `ofWeight` i) [0 ..] [lo .. hi]]
```

E EXAMPLE OF REDUCED PACKAGE.JSON

```
{
  "name": "reflective-generators",
  "description": "What a great project",
  "scripts": {
    "start": "node ./src/server.js",
    "build": "babel ./src -out-dir ./dist",
    "test": "mocha ./test"
  },
  "repository": {
    "type": "git",
    "url": "https://example.com"
  },
  "keywords": [
    "reflective",
    "generators"
  ],
  "author": "test",
  "license": "mit",
  "devDependencies": {
    "babel-cli": "^6.24.1",
    "babel-core": "^6.24.1",
    "babel-preset-es2015": "^6.24.1"
  },
  "dependencies": {
    "express": "^4.15.3",
    "reflective": "^0.0.1"
  }
}
```

```
{
  "name": "a",
  "description": "a",
  "scripts": {
    "start": "a",
    "build": "a",
    "test": "a"
  },
  "repository": {
    "type": "a",
    "url": "a"
  },
  "keywords": [],
  "author": "a",
  "license": "a",
  "devDependencies": {},
  "dependencies": {
    "express": "^4.15.3"
  }
}
```

F REFLECTIVE GENERATOR FOR POLYMORPHIC LAMBDA CALCULUS TERMS