Reflecting on Random Generation

ANONYMOUS AUTHOR(S)

The random data generators used for property-based testing are often painstakingly crafted programs that encode what it means for a test input to be valid and interesting. This has led developers of frameworks like Python's Hypothesis library to repurpose generator programs for things like 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 propose *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 value that *could* have been produced by the generator. Besides shrinking and mutation, reflective generators also generalize a published algorithm for example-based generation, and they can also be used as enumerators, validity checkers, and more.

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 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 *shrinking* and *mutation*. 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*, taking unstructured random choices and *parsing* them into structured values; this capitalizes 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 the shrinker has the original random choices that the generator made when producing the value it plans to shrink: shrinking does not work 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

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.

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.

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 a value, determining which *productions* must be expanded to produce that value. Then, they bias a generator to expand those productions more often, thus resulting in more values 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 *Inputs from Hell* approach works well for grammar-based generators, but does not apply to generators that enforce validity. We need a solution that works for the kinds of *monadic* generators used in QuickCheck. Such a solution can be found in 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 a value to analyze which choices produce that 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 a monadic 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 a value to obtain choices that produce it. (§3)
- We develop the theory of reflective generators, defining what it means for reflective generators to be correct along a number of axes and demonstrating their expressive power relative to other generation abstractions. (§4)

```
class Monad m where                          do { x <- m; f x } = m >>= f
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b          type Gen a = Int -> a
```

(a) Definitions for monadic generators.

```
data Freer f a where                       class Profunctor p where
  Return :: a -> Freer f a                   lmap :: (d -> c) -> p c a -> p d a
  Bind :: f a                                rmap :: (a -> b) -> p c a -> p c b
       -> (a -> Freer f b)
       -> Freer f b                        class Profunctor p => PartialProf p where
                                             prune :: p b a -> p (Maybe b) a
data Pick a where
  Pick :: [(Weight, Choice, Freer Pick a)]  class (forall b. Monad (p b), PartialProf p)
       -> Pick a                              => PMP p
```

(b) Definitions for free generators. (c) Definitions for partial monadic profunctors.

Fig. 2. Background definitions.

- 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 and extends it to work with monadic generators. (§5)
- 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. We show that our shrinkers are at least as effective as other automated shrinking techniques. (§6)
- 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, "completers" that can randomly complete a partially defined value, and other test input producers. (§7)

We conclude by discussing related work (§8) and 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: monadic generators (§2.1), free generators (§2.2), and partial monadic profunctors (§2.3).

2.1 Monadic Random Generators

The basic idea of testing executable properties using monadic generators was popularized by QuickCheck [Claessen and Hughes 2000] and has endured for more than two decades. PBT frameworks have appeared in many mainstream programming languages, including Hypothesis [MacIver et al. 2019] in Python. The core structures QuickCheck relies on are shown in Figure 2a: `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], which provide a neat interface for chaining effectful computations.

2.2 Free Generators

Interfaces like the Monad type class can be reified into "free" structures that represent each operation of the type class as a data constructor; these data constructors can then be interpreted in multiple ways. There are actually several such free structures for the monad interface; we focus on the *freer monad* [Kiselyov and Ishii 2015] structure shown in Figure 2b, which reifies the return operation as the constructor Return and (>>=) as Bind; an extra type constructor f ranges over the operations that are specific to each given monad.

Free generators [Goldstein and Pierce 2022] are an instance of this scheme, instantiating f as the type constructor Pick, which represents a choice between sub-generators. Pick 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 draw a formal connection between random generation and parsing, interpreting the same free generator as both a generator and a parser.

2.3 Partial Monadic Profunctors

Profunctors are a standard category theoretic construction, generalizing ordinary functors—structure-preserving maps—to allow for both covariant and contravariant mapping operations. Profunctors are realized as the Profunctor class (Figure 2c), popularized in Haskell by Pickering et al. [2017], where the mapping operations are called rmap and lmap respectively. The quintessential example of a profunctor is the (->) (function) type constructor; this makes sense, since b -> a is contravariant in b and covariant in a. In that case, rmap implements post-composition (of some function a -> a') and lmap implements pre-composition (of a function b' -> b). Indeed, it is often useful to think of profunctors as function-like: a value of type p b a "examines a value of type b to produce a value of type a" (potentially doing some other effects).

A *monadic profunctor* is a profunctor that is also a monad (i.e., a profunctor p such that for any b, p b is a monad). Xia et al. [2019] use this extra structure to implement composable bidirectional computations. For example, consider the classic bidirectional programming example of a parser and a pretty printer, which invert one-another. These dual functions can be implemented using the same monadic profunctor:

```
data Biparser b a = { parse :: String -> (a, String); print :: b -> (a, String) }
```

How does it work? A Biparser is a single program that gets "interpreted" in two different ways. As a parser, it ignores the b parameter entirely, and simply acts as a parser in the style of Parsec [Leijen and Meijer 2001] to produce a value of type a. As a pretty printer, it still needs to produce a value of type a—after all, the two interpretations share the same code—but now there is no input String to parse. Instead, the pretty printer interpretation has a value of type b uses as instructions to produce both the a and a String. This scheme makes much more sense when the profunctor is *aligned*, that is of type Biparser a a; in that case, a properly written pretty printer acts as an identity function, taking a value and reproducing it while also recording its String representation.

Xia et al. call the first type of interpretation, which ignores its contravariant parameter and just produces an output, a "forward" interpretation and the second type, which acts as an identity function and follows the structure of its contravariant parameter, a "backward" interpretation. We say parse goes forward and print goes backward. This is an arbitrary choice, but it is helpful as an analogy for the way that certain monadic profunctors are duals of one other.

Writing a Biparser, or any other monadic profunctor, is a game of type alignment. In general, aligned programs are desirable, but aligning the types gets tricky around monadic binds. Suppose we have an aligned p a a and we want to get an a out and continue with a function of type a -> p b b (whose codomain is also aligned). We *cannot* simply use (>>=) whose type is:

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

The problem is the first argument: we have a value of type `p a a` but we need one of type `p b a` since `p b` is the type of the monad. Luckily, the `lmap` operation (§2c) 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` that says how to focus on some part of a value of type `b` that has type `a`. We can thus build up a `Biparser` by writing a program that looks essentially like a standard monadic parser, but with monadic binds annotated with calls to `lmap` that fix the alignment.

This story is almost complete, but it leaves out cases where annotations need to be partial. Consider a `Biparser` like this one, which parses either a letter or a number:

```
letter :: Biparser Char Char
number :: Biparser Int Int
data LorN = Letter Char | Number Int

lOrN :: Biparser LorN LorN
lOrN = Letter <$> lmap ??? letter   <|>   Number <$> lmap ??? number
```

The type of the first annotation should be of type `LorN -> Char`, but there is no way to write that function: what happens if the `LorN` is a `Number`? The more appropriate annotation type would be `LorN -> Maybe Char`. Xia et al. make this possible with *partial monadic profunctors* (PMPs), which add one more operation, `prune`, to capture failure. (We have renamed it `prune` from the original "`internaliseMaybe`", as it is more intuitive for our purposes.) Unlike with monadic profunctors, which can only be annotated with total functions using `lmap`, 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
```

When a PMP like `print` branches (e.g., in `lOrN`) the execution follows both sides (e.g., trying to pretty print both a `Letter` and a `Number`). The partial annotations tell the computation when to *prune* a particular branch, keeping the search space small and ensuring that PMPs like `print` are efficient.

PMPs are complex, and they can be used in a wide variety of ways. We highly recommend reading *Composing Bidirectional Programs Monadically* [Xia et al. 2019] if this explanation was not yet clear. Next, we discuss an extension to PMP that is incredibly useful for PBT.

3 THE REFLECTIVE GENERATOR LANGUAGE

Reflective generators combine free generators with PMPs, enabling a host of generalized testing algorithms. In this section, we explain the intuition behind reflective generators (§3.1), describe their implementation (§3.2), and discuss their various interpretations (§3.3).

The basic structure of a reflective generator comes from adding the partial monadic profunctor operations, `lmap` 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 here 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 contravariant annotations—there is no need to explicitly represent `rmap`, as we will be able to encode it using the monad structure. We also add `Prune`, with an analogous type to the PMP operation from Figure 2c. A reflective generator is then a freer monad over `R b`:

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

3.1 Intuition

Specializing the intuition from §2, the type `Reflective b a` of reflective generators should be understood to mean a program that can "reflect on a value `b`, recording choices, 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`; as discussed in §2.3 we need to add an annotation on the 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 })
```

As with PMPs like `Biparser`, the process of reflecting on choices is all about the interaction between binds and `comap`s. 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, that branch gets pruned—there is no way to produce the desired value—but if it 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 Implementation

The basic structure of reflective generators is shown at the beginning of the section. Here, we describe some details of how reflective generators are implemented, and what we have done to make them feel both familiar and easy to work with. (In §9, we discuss plans to validate this belief with a proper user study.)

[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 4.4.

The Full Story. The actual type R defined in the Haskell artifact is a bit more complicated than the one explained above. The actual implementation looks like:

```
data R b a where
  Pick  :: [(Weight, Choice, Freer (R b) a)] -> R b a          -- as before
  Lmap  :: (c -> d) -> R d a -> R c a                          -- as before
  Prune :: R b a -> R (Maybe b) a                              -- as before
  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 function for choosing within a range of integers allows us to bootstrap other generators over large ranges, and it it makes it easy to implement much more efficient interpretation functions later on.

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, and, although it is possible to implement sized generators by passing size parameters around manually, internalizing size control cleans up the API of the combinator library and makes generators more readable.

In future sections, we often elide parts of definitions pertaining to these operations, to streamline the presentation, but they do present a couple of theoretical complications that we note in §4.

Building a Domain-Specific Language. We implement a variety of combinators that make reflective generators easier to read and write, aiming for an interface that captures the full power of reflective generation without straying too far from QuickCheck syntax.

The most important reflective generator operation is `Pick`, so we provide a number of choice combinators that are built on top of it:

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

The most flexible, `pick`, just passes through to the `Pick` constructor, wielding its full power. The rest are simplifications of this operation that represent common use cases. A bit simpler, `labeled` takes only choice labels and no weights—it sets all weights to 1. Finally, `frequency`, `oneof`, and `choose` have the same API as their counterparts in QuickCheck, forgoing choice labels.

A brief aside on choice labels: whether or not a user decides to label the choices in a generator depends on two factors. First, it depends on how the generators will be used. In §5.1, we discuss a use case that relies heavily on choice labels, and in §6 we discuss one that ignores them. Second, whether or not a particular sub-generator is labeled can impact the behavior of use cases that pay attention to labels; in §5.1 we discuss intentionally eliding labels as a way of marking parts of the generator whose distributions should not be tuned. As a general rule, we recommend labeling choices in generators, and all generators provided by the reflective generators library are labeled by default, but it is convenient to be able to elide labels when upgrading from a QuickCheck generator.

Choice operators alone are not enough to build a reflective generators, we need infrastructure to glue them together. The bulk of these glue operations follow from the fact that reflective generators are, as expected, PMPs:[2]

```
instance Profunctor Reflective where
  lmap _ (Return a) = Return a
  lmap f (Bind x h) = Bind (Lmap f x) (lmap f . h)
  rmap = fmap

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

Both `lmap` and `prune` commute over `Bind` and do nothing to a `Return` (see the laws in §4), so these implementations are straightforward. Behind the scenes, the `Functor`, `Applicative`, and `Monad` operations are implemented for free from the freer monad.

Using `lmap` and `prune` on their own is a bit tedious, so we give two combinators that make common use cases much simpler. The `focus` combinator makes it possible to replace pattern matches in `lmap` annotations with *lenses* [Foster 2009].

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

The curious reader can dig into the gory details of the types[3] involved, but it suffices to understand `focus` as a notational convenience that gives a terse syntax for pattern matches:

```
focus (_Node._2) = lmap (\ case { Node _ x _ -> Just x; _ -> Nothing }) . prune
```

Another convenient helper built from `lmap` and `prune` 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 =
  lmap (\ a' -> if a == a' then Just a else Nothing) .
  prune $ (Pick [(1, Nothing, return x)])
```

Using this function (or manually `prune`ing) at the leaves of a reflective generator is critical: without it, the generator may incorrectly claim to be able to produce an invalid value.

We saw above that combinators like `oneof`, `frequency`, and `choose` align closely with the QuickCheck API to make upgrading easier. We provide one more combinator to simplify the upgrade process, `noAnn`, which can be used in place of an annotation:

```
noAnn :: Reflective b a -> Reflective Void a
noAnn = lmap (\ x -> case x of)
```

The `Void` type in Haskell is uninhabited, and thus a reflective generator of type `Reflective Void a` can only be used in limited cases, but `noAnn` makes it possible to perform the upgrade from Figure 1 in stages. First, go from Figure 1a to the one in Figure 3. Then go from Figure 3 to Figure 1b by replacing `Void` with the correct output type, replacing `noAnn` annotations with ones that do appropriate focusing, and replacing `return` with `exact` where appropriate. Experienced reflective

[2]Technically, these definitions are not legal Haskell, since both partially apply the `Reflective` type constructor, which is not supported by GHC. In the Haskell artifact we implement the operations as normal functions (rather than type-class methods).

[3]https://hackage.haskell.org/package/lens-5.2/docs/Control-Lens-Getter.html

generator writers do the upgrade in a single step, but when starting out it may be easier to take a detour through a simpler intermediate generator.

There are a number of other combinators implemented in the artifact, including `getSize` and `resize`, standard generators for base types, and higher-order combinators for lists and tuples.

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

Fig. 3. An intermediate generator with Void as its contravariant type, using noAnn.

3.3 Interpretation

Like 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. As with PMPs, most reflective generator interpretations can be thought of as either "forward" interpretations, which simply produce an output of their covariant type, or "backward" interpretations, which reflect on a value (and reproduce it *en passant*) while tracking choices. However, unlike PMPs, reflective generators do not explicitly pair forward and backward interpretations together—in fact, the interpretation in §7.2 actually uses both directions at once. Instead, directionality for interpretations of reflective generators is simply a useful intuition.

The simplest "forward" interpretation turns a reflective generator into 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 and `Pick` interpreted as a weighted random choice.

Of course, the value of reflective generators lies in their 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 in Figure 4, which interprets a generator to determine which choices could lead to a given value. When interpreting `Pick` the computation splits, each branch representing making one particular choice. In each branch, `Lmap` nodes focus on parts of the value being reflected on; if the focusing fails, a following `Prune` node will filter that branch out of the computation. The monad structure, `Return` and `Bind`, threads the list of recorded choices through the computation, so the final result is a list of the different branches of the computation that were not pruned, along with the choices made in each of those branches.

```
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])])
    interp (Return a) = \ _ -> return (a, [])
    interp (Bind r f) = \ b -> do               -- Thread choices around.
      (a,  cs ) <- interpR r b
      (a', cs') <- interp (f a) b
      return (a', cs ++ cs')
```

Fig. 4. The "reflect" interpretation.

These two interpretations demonstrate the essence of reflective generators, but they are far from the only ones—in total we present 10, all of which can be found in our artifact, and we expect there are use-cases for many more. As a user, this gives an amazing amount of flexibility, since a single reflective generator can be interpreted in all of these ways.

4 THEORY OF REFLECTIVE GENERATORS

In this section, we describe more of the theory underlying reflective generators. We discuss various formulations of correctness, including defining what it means to correctly interpret a reflective generator and what it means to correctly write an individual reflective generator (§4.1). Next, we explore an interesting property of reflective generators—*overlap*—which has implications for generator efficiency (§4.2). Finally, we discuss the expressive power of reflective generators, comparing it to grammar-based generators and to standard monadic ones (§4.3).

4.1 Correctness

Both interpretations and individual reflective generators can be written incorrectly—the types involved are not strong enough. Here, we describe algebraic properties that the programmer should prove (or test) to ensure good behavior.

Correctness of Interpretations. Reflective generators should obey the laws of monads [Moggi 1991],

```
(M1)    return a >>= f = f a
(M2)    x >>= return = x
(M3)    (x >>= f) >>= g = x >>= (\ a -> f a >>= g)
```

of profunctors,

```
(P1)    lmap id = id
(P2)    lmap (f' . f) = lmap f . lmap f'
```

and of PMPs:

```
(PMP1)  lmap Just . prune = id
(PMP2)  lmap (f >=> g) . prune = lmap f . prune . lmap g . prune
(PMP3)  (lmap f . prune) (return y) = return y
(PMP4)  (lmap f . prune) (x >>= g) = (lmap f . prune) x >>= (lmap f . prune) . g
```

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

Lemma 4.1. *Reflective generators always obey* ($M1$)*,* ($M3$)*,* ($PMP3$)*, and* ($PMP4$)*.*

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: they must be established for each 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 an *R-interpretation*

```
⟦·⟧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
```

An alternative approach would be to simply define an interpretation of a reflective generator as a PMP homomorphism along with an interpretation for `Pick`, rather than giving the programmer the freedom to implement lawless interpretations. From a programming perspective, this would behave like a tagless-final embedding [Kiselyov 2012]. We found this tagless-final approach more tedious to program with, but it is available to users if desired (see Appendix A).

The `generate` is indeed lawful, modulo one technical caveat. The classic `Gen` "monad" itself is not actually a lawful monad, but it *is* lawful up to distributional equivalence [Claessen and Hughes 2000]—i.e., it generators that produce equivalent probability distributions of values are equivalent, even if they are not equal as Haskell terms. The same caveat applies to the other laws.

Theorem 4.2. *The* `generate` *interpretation is lawful up to distributional equivalence.*

Proof. Since `Lmap` and `Prune` are both ignored, the other laws are trivial. □

The `reflect` interpretation needs no such technical caveats: its laws hold on the nose.

THEOREM 4.3. *The* `reflect` *interpretation is lawful.*

PROOF. By induction on the structure of the generator. See Appendix B. □

Correctness of a Reflective Generator. The proofs of lawfulness for each of the interpretations we want to use can be carried out once and for all, but there is also some work to do for each individual reflective generator, to ensure that its various interpretations will behave the way we expect. We next characterize what it is for a reflective generator to be *correct* and implement this characterization as a QuickCheck test.

Our correctness criteria are based on similar notions set forth by Xia et al. [2019]. We formulate correctness using two interpretations. The `generate` interpretation is the canonical "forward" interpretation, characterisizng the set of values that can be produced by a reflective generator when it ignores its input. The canonical "backward" interpretation should characterize the generator's operation as a generalized identity function, taking a value and reproducing it—`reflect` is almost the right interpretation, but it does extra work to keep track of choices. Thus, we define: `reflect`

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

The `reflect'` interpretation has the same behavior as `reflect`, but it simply skips the code that tracks choices. The full code for this and all other interpretation functions can be found in our artifact.

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).`[4]

Where $a \sim \gamma$ means "a can be sampled from QuickCheck generator γ."

In other words, if the `generate` interpretation can produce a value, then the `reflect'` interpretation can reflect on that value without failing.

Conversely:

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

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

In other words, if the `reflect'` interpretation successfully reflects on a value, then that value should be able to be sampled from the `generate` interpretation.

Of course, completeness is impossible to test: there is no way to check `a ∼ generate g` directly. Luckily, 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.`

Weak completeness is still impossible to test, but it is *compositional*, meaning it is true of a generator if it is true of its sub-generators. Since the only kind of sub-generator reflective generators can be built from is `Pick`, we can prove this once and for all:

[4]The definition we give for soundness is morally 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, we should instead test

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

for all n in a reasonable range.

Lemma 4.4. *All reflective generators are weak complete.*

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

Xia et al. also gives a so-called *pure projection property*, which is testable (albeit sometimes intractably[5]):

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

```
a' ∈ reflect' g a ==> a = a'.
```

To complete the picture, we prove the following:

Theorem 4.5. *Any weak-complete reflective generator satisfying pure projection 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. □

The take-away is that testing completeness of a reflective generator directly is impossible, but testing pure projection suffices, where tractable. When determining the correctness of a reflective generator, one should definitely test soundness and test pure projection, where tractable.

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

4.2 Overlap

One last theoretical property of a reflective generator worth noting is its *overlap*.

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

Many reflective generators naturally have an overlap of 1, meaning that there is only one way to generate any given value, but some generators benefit from higher overlap. For example, a generator might pick between two high-level strategies for generating values for the sake of distribution control.

But overlap can cause problems for backward interpretations that care about examining all ways of producing a particular value (e.g., `probabilityOf` which we will define in §7). In these cases,

[5]The issue with testing this property is that it is sometimes unlikely that the antecedent of ==> is fulfilled, leading to QuickCheck giving up. Increasing the max discard ratio can help alleviate this.

```
g1 = labelled [ ("Z", exact Z), ("S", fmap S (focus _S g1)) ]

gE =                                                gI =
  labelled                                            labelled
    [ ("Z", exact Z),                                   [ ("Z", exact Z),
      ("S", fmap S (focus _S gE)),k                       ("S", fmap S (focus _S gI)),
      ("2", fmap (S.S) (focus (_S._S) gE))                ("inf", gI)
    ]                                                   ]
```

Fig. 5. Reflective generators with unit, exponential, and infinite overlap.

overlap may lead to exponential blowup or even nontermination. For example, consider the three generators in Figure 5. The first, g1, generates natural numbers, each in exactly one way:

```
ghci> reflect g1 (S (S (S (S (S Z))))) -- 5
[["S","S","S","S","S","Z"]]
```

The second, gE, can generate numbers in exponentially many ways; specifically, it can generate a number by generating any sum of 1s and 2s that add to the desired total:

```
ghci> length (reflect gE (S (S (S (S (S Z)))))) -- 5
8
ghci> length (reflect gE (S (S (S (S (S (S ...))))))) -- 10
89
```

Computing `reflect gE` of a large number could take a very long time, which may be a problem for some use-cases. Finally, we have `gI` which includes the option to make a no-op choice, `"inf"`:

```
ghci> length (reflect gI (S (S (S (S (S Z)))))) -- 5
...
```

Calling `reflect gI` does not terminate—there are infinitely many ways to generate 5. However, we conjecture that any generator with infinite overlap can be made into one that does not, by ensuring that any loop is guarded by some change to the generated structure.

4.3 Expressiveness

Reflective generators fall on a spectrum between simple grammar-based generators and complex monadic ones. Here, we show off the different kinds of data constraints that reflective generators can express and discuss a few idioms that they cannot express.

Grammar-Based and Monadic Generators. Grammar-based generators [Godefroid et al. 2008] use a context-free grammar describing the program's input format as the basis for generating test inputs. For example, the following grammar fragment can be thought of as defining a generator of expression parse trees:

```
term -> factor | term "*" factor | term "/" factor
```

Read as a generator, this says "to generate a `term`, choose either a `factor`, a `"*"` node, or a `"/"` node." Grammar-based generators are useful for generating inputs to a program with a context-free input structure, like expression evaluators, JSON minifier, or even some compilers; they are often used in fuzzing for this reason. But grammar-based generators cannot ensure that the values they generate satisfy context-sensitive constraints. One might, for example, want to ensure that the lefthand side of a division does not evaluate to zero:

```
term -> factor | term "*" factor | term "/" nonzero(factor)
```

A complete generator of these expressions would require evaluation to take place during generation, which is not possible as part of a context-free grammar. And this is just the tip of the iceberg: there are a host of context-sensitive constraints that a generator might need to satisfy.

Enter monadic generators. As described in §2.1, monadic generators were introduced with QuickCheck and are a domain-specific language for writing generators that produce inputs satisfying arbitrary computable constraints. Monadic generators can generate binary search trees [Hughes 2019], well-typed terms in a simply-typed lambda calculus, and more. Monadic generators subsume context-free generators, for example, the following generator subsumes the term generator above:

```
term = oneof [fmap Factor factor, liftM2 Mul term factor, liftM2 Div term factor]
```

And with a bit more effort, we can exclude the parse trees with a divide-by-zero error:

```
term = oneof [ fmap Factor factor, liftM2 Mul term factor,
  do f <- factor
     if eval f == 0 then liftM2 Mul term (return f) else liftM2 Div term (return f) ]
```

There is technically one more rung on this ladder: Hypothesis generators. While they are not computationally more powerful than monadic ones, they are implmented in Python and can thus perform arbitrary side-effects while generating. As we move on to analyzing reflective generators' expressiveness, we continue to focus on pure generators, but incorporating an extra monad argument (and thus, arbitrary effects) to reflective generators is compelling future work.

What Reflective Generators Can Do. As a start, reflective generators are certainly at least as powerful as grammar-based generators.

Claim 1. Every grammar-based generator can be turned into a reflective generator via an analogous procedure to the one for monadic generators.

Justification. A grammar can be made into a monadic generator in the following way. For every rule $S \longrightarrow \alpha_1 \mid \cdots \mid \alpha_n$, we can write a generator

$$\texttt{s = oneof [liftMi } \texttt{C}_1\ \mathcal{T}(\alpha_1)\texttt{, ..., liftMj } \texttt{C}_n\ \mathcal{T}(\alpha_n)\texttt{]}$$

where $\texttt{C}_1$ through $\texttt{C}_n$ are fresh data constructors, and $\mathcal{T}$ translates each production by turning non-terminals into the appropriate sub-generator and turning terminals into terminals into Haskell strings. To turn that monadic generator into a reflective generator, simply add `focus` annotations that extract each argument from each constructor (`C`). □

For example, this is the `term` generator that results from translating the grammar-based generator to a reflective generator:

```
term = oneof [
  fmap Factor (focus _Factor factor),
  liftM2 Mul (focus (_Mul._1) term) (focus (_Mul._2) factor),
  liftM2 Div (focus (_Div._1) term) (focus (_Div._2) factor) ]
```

In fact, reflective generators can implement all of the examples we previously listed as the motivation for monadic generators (see the binary search tree generator in Figure 1b and the STLC generator fragment below). There are also a variety of examples in §5.1 and §6 that are expressible by monadic generators and not context-free ones.

To see how reflective generators fare in a complex case, consider a reflective generator for terms in a simply-typed lambda calculus (STLC). The STLC generator is built from two sub-generators:

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

(The underscore prevents `type_` from being interpreted as a keyword.) The STLC generator works by picking a type, then generating a value of that type. The monadic version of the generator would simply write `type_ >>= expr`. But this does not work for a reflective generator; it needs an annotation. Specifically, what's needed is a mapping from `Expr` to `Type` that can focus on the type in the expression. Pleasingly, this focusing is precisely type inference! The type-correct reflective generator is: `comap typeOf type_ >>= expr`.

What Reflective Generators Can't Do. Given that reflective generators seem to be able to express so much, it may be easier to characterize what they *can't* express. The biggest limitation of reflective generators is that they cannot represent any approach to generation that fundamentally loses information about previously generated data. For example, in `lmap ??? g >>= \ _ -> g'` there is no valid annotation to write, because the value generated by `g` cannot be "focused" on as part of the final structure. Why might this come up? One case is when the generator generates a value and then computes some un-invertible function on it; there would be no way to recover the original value to analyze the choices made when producing that value.

We have run across very few generators that fundamentally require an un-invertible function, but one interesting examples is some formulations of System F, or the polymorphic lambda calculus [Girard 1986; Reynolds 1974]. It is possible (though challenging) to write a monadic generator for System F [Pałka et al. 2011], but impossible to do so for reflective generators. The problem can be seen when referring back to the reflective generator for STLC terms, which uses type inference to recover a type from an expression. If we tried to translate this generator to one for System F, we would have a problem: type inference for System F is undecidable! Thus, the generator may fail to run backward, even if it works correctly when run forward. Of course, this is not a deal-breaker in practice—we have a System F reflective generator in the artifact, and modulo some time-outs it works—but this is a neat example of the dividing line between monadic and reflective generators.

System F is a rare example of fundamantal limitations of reflective generators, but there are a few common generation idioms that reflective generators need to work around. First, reflective generators cannot use the QuickCheck combinator "`suchThat`" that samples a value, and, if it does not satisfy a some predicate, throws it away, *increases the size parameter*, and samples another. This size manipulation is the problem: 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. Second, reflective generators do not support a relatively common idiom where generators pick an integer and then use that integer to bias the generation distribution in some way. 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.

Ultimately, we consider reflective generators to be *almost as powerful* as monadic generators in practice.

5 EXAMPLE-BASED GENERATION

In this section, we demonstrate the power of reflective generators in the context of a clever generation technique that was an early inspiration for their design—example-based generation (§5.1). We replicate and generalize the prior work using reflective generators (§5.2).

5.1 *Inputs from Hell*

Inputs from Hell [Soremekun et al. 2020] (IFH) describes an approach to random testing that starts with a set of user-provided example test inputs and randomly produces values that are either quite similar to or quite different from those examples—the idea being that similar values represent "common" inputs and that different ones represent interestingly "uncommon" inputs; by drawing test cases from both of these classes, IFH is able to find bugs in realistic programs.

The IFH approach is based on grammar-based generation. Examples provided by the user are parsed by the grammar, and the resulting parse trees are used to derive weights for a probabilistic context-free grammar (pCFG) that generates the actual test inputs. For example, given a simple grammar for numbers

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

and the example 12, the IFH technique might 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 a weight based on the number of times it appears in the parse tree for the provided example (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.

5.2 Reflecting on Examples

This process—parse the input, analyze the parse tree, and re-weight the grammar—is fairly straightforward to implement in the setting of grammar-based generation, but the IFH work does not extend to monadic generators. As we discuss in §4.3, this is a significant limitation. Reflective generators can bridge this gap by recapitulating the ideas in IFH but using a reflective generator for IFH's parsing and generation steps.

Implementation. We define three functions, corresponding to 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 lists of choices that were made to produce that value. With the choice sequences in hand, `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 interpretation, `genWithWeights`, generate new values by making choices with the weights calculated from the user-provided examples.

When a reflective generator looks like a grammar (as with `term` in §4.3), this process replicates the IFH algorithm exactly. But we have seen that reflective generators are far more powerful than grammar-based generators, so the new algorithm both replicates *and generalizes* IFH, enabling example-based generation for a much larger class of generators.

These interpretations rely heavily on choice labels, which we discuss briefly in §3.2. These labels are used by `reflect` and `genWithWeights` to track the choices that should be weighted higher or lower based on the examples. This means that, rather than building reflective generators with `oneof` or `frequency`, the programmer should use `labeled` or `pick`. Recall that the reflective generators library provides base generators that are labeled as well. However, there is some flexibility here: if the programmer would prefer some choices *not* be re-weighted based on examples, they can simply elide the labels. This is another way that the reflective generators approach generalizes IFH.

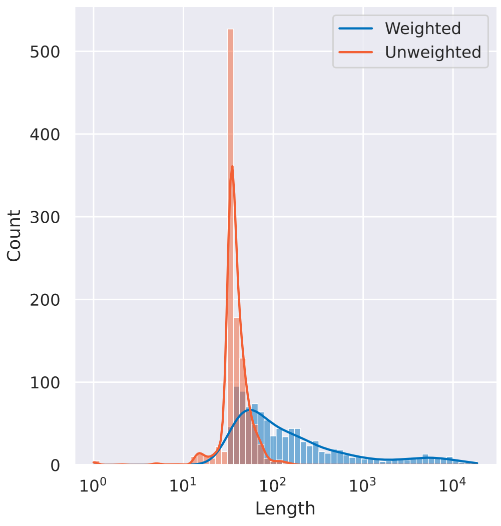
Example-Based Generation in Action. Soremekun et al. [2020] use the IFH tuning algorithm as part of a comprehensive testing regime; by contrast, we have found them to be most useful as a quick way to tune a generator to yield a reasonable distribution of sizes and shapes.

To see this in action, consider a generator that generates a JSON document (the payload) along with a short hash of the document that can be used as a checksum:

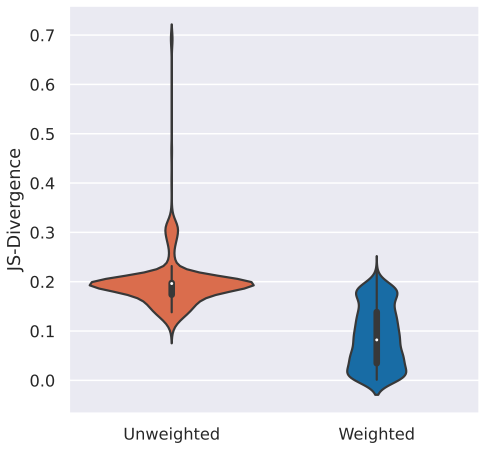
```
withHashcode :: Reflective String String
```

The full generator can be found in Appendix F. This is inspired by a generator for JSON documents from the IFH paper, the reflective version of which is shown in full in Appendix D. Note that, while the JSON part of the generator is equivalent to a context-free one, `withHashcode` is not (since it has to compute the hash during generation).

We sampled 10 JSON documents that were used in the IFH experiments, ranging from ~200-1,200 bytes long, and used them to weight `withHashcode` 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 6.



(a) Length distributions of unweighted and weighted.



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

Fig. 6. Analysis of `withChecksum` tuned by example in the style of *Inputs from Hell.*

Figure 6a 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 33. Inspecting the data revealed that the payloads of these inputs are all either `{}` or `[]`, both relatively uninteresting and certainly not worth generating hundreds of times! In contrast, the weighted generator has a varied length distribution. It generates very few trivial inputs, instead producing a wide distribution that covers more of the input space.

Figure 6b 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[6] [Lin 1991], between the example distribution and the character distribution of the sample and plotted those divergences in a violin plot. JS divergence measures the difference between two probability distributions, so it is a simple way of getting a sense of how similar or different the characters in the samples are from the ones in the examples. The plots show that the unweighted samples are farther from the the example distribution than the weighted samples.

Without this example-based tuning, the developer of `withHashcode` would need to think carefully about the distribution that they want and even harder about how to alter the generator weights to get there. With example-based tuning, they simply need to assemble 10 or so examples, compute weights from those, and then use those weights for generation instead.

6 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 (§6.1), show that reflective generators make these algorithms much more flexible (§6.2), and finally sketch the ways that these ideas also apply to test-case mutation (§6.3).

6.1 Test-Case Reduction in Hypothesis

Shrinking is the process of turning a large counterexample into a smaller one that still triggers a bug. Shrinking is critical in PBT because bugs are often tickled by very large inputs that are nearly impossible to use for debugging—shrinking makes it much easier to understand which specific bits of the input are actually necessary to trigger the bug.

In QuickCheck, users can either use GHC's Generics [Magalhães et al. 2010] to derive a shrinker automatically for a given type, or they can write a shrinker by hand. The former is effective in simple cases, as we will see below, but it is not very general—these automatic shrinkers only know about the type structure, so they cannot ensure that the shrunken values satisfy important preconditions nor adequately shrink less structured data like strings. The latter is totally general, but many users find writing shrinkers by hand confusing and error-prone.

This unsatisfying situation led MacIver and Donaldson [2020] to design Hypothesis's "internal test-case reduction," which gives the best of both worlds. It solves the generality issue without requiring user effort or understanding. The key insight is that the generator itself already has all of the information needed to produce precondition-satisfying inputs, so the generator should be used as part of the shrinking process. The accompanying clever trick is to *shrink the random choices* used to generate an input value, rather than shrinking the value itself.

Concretely, Hypothesis represents its input randomness as a bracketed string of bits. For example `(10(1(100)0))` produces the tree in Figure 7. 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 counterexample; shortlex order considers shorter strings to be less than longer strings, and follows lexicographic ordering otherwise (brackets are ignored for the purpose of ordering). In

[6]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.

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

`(10(1(100)0))`

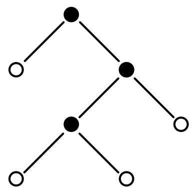


Fig. 7. A Hypothesis-inspired reflective generator and a tree that the generator might produce.

practice, shortlex order turns out to be an effective proxy for complexity of generated test cases: smaller bitstrings tend to produce smaller test cases.

The actual shrinking procedure uses a number of different passes, each of which attempts to shorten the choice string, swap 1s with 0s, or both, resulting in a shortlex-smaller choice string. The passes are described in the Hypothesis paper and available in the open source codebase[7].

6.2 Reflective Shrinking

The downside of the Hypothesis approach is that this style of shrinking 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 for which one does *not* have a corresponding bitstring. In particular, shrinking can be useful for understanding externally provided inputs that were not produced by the generator at all; for example, if a user submits a bug report containing a printout of some large input that caused a crash, it might 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, as might be desirable when doing fuzzing-style testing where test inputs are mutated to explore values in a particular region. Luckily, reflective generators can help.

Extracting Bracketed Choices Sequences. We implement *reflective shrinking* via yet another interpretation of reflective generators, with the following type:

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

The `Choices` type describes rose trees with two types of nodes: `Choice`, which represents a single-bit choice, and `Draw` which represents some nested sequence of choices;[8] this type 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 a list of all the choice sequences that result in generating that value.

The implementation of `choices` is similar to that of `reflect`. 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:

[7]https://github.com/HypothesisWorks/hypothesis

[8]In the Haskell artifact, we use a slightly more complicated type, caching 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].

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

```
(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. 8. Shrinking strategies.

Shrinking Strategies. With an appropriate bracketed choice sequence in hand, shrinking can begin. We implemented a representative subset of the shrinking passes described in the Hypothesis paper: one pass tries shrinking to every available child sequence of the original, a second replaces Draw nodes 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 8a, 8b, and 8c accordingly.

Replicating Hypothesis Evaluation. To check that we replicated Hypothesis shrinking correctly, we replicated one of the experiments from the Hypothesis paper. MacIver and Donaldson 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 counterexample and shrink that counterexample 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. These modifications are based on the observations from §4.3. Then, we ran each experiment 1,000 times and reported the average size of the resulting counterexamples 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.

We find that 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 counterexamples that they produce, likely because they only implement a subset of Hypothesis's shrinking strategies, but nevertheless their counterexamples 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.

A Realistic Example. As a final demonstration that reflective shrinkers are useful, we return to a modified version of the JSON example used in §5. 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 G), the developers can use their reflective shrinker to reduce it to a far simpler file. The new file (also in Appendix G) 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.

6.3 Reflective Mutation

HypoFuzz, a tool for coverage-guided fuzzing [Fioraldi et al. 2020] of PBT properties, is a newer and lesser-known part of the Hypothesis ecosystem. Like Crowbar [Dolan and Preston 2017], HypoFuzz uses a PBT generator to aid the fuzzer.

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. "Mutating well" can be challenging, since naïve mutations will rarely produce values that are valid inputs to the program; HypoFuzz gets around this 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 positive side, it is type agnostic, easy to use, and guarantees validity of the mutated values. On the other side, 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. but this does not work with Hypothesis-style mutation: the seeds needed for this style are not user-comprehensible inputs but choice sequences! Once again, reflective generators provide a compelling solution. We can simply extract a choice sequence from each seed using the `choices` interpretation.

7 IMPROVING THE TESTING WORKFLOW

So far we have seen reflective generators in the context of example-based generation, shrinking, and mutation. In this section, we explore several more useful interpretations, demonstrating reflective generators' power and flexibility.

7.1 Reflective Checkers

The "bigenerators" in the original work on PMPs [Xia et al. 2019] can be viewed as a special case of reflective generators. Rather than rather than reflect on a value and produce 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, by reflecting on the generator's choices and asking whether or not there exists a set of choices that results in the desired value.

Going further, a reflective generator can calculate the *probability* of generating a particular value with the `generate` interpretation. We implement this in our artifact as an interpretation, probabilityOf, which tracks the different ways of generating a particular value and the likelihood of choosing those different ways. Obviously this works best when the generator's overlap is low (see §4.2)—in cases where overlap is exponential or infinite this may be slow or fail to terminate.

7.2 Reflective Completers

A rather different use case for reflective generators is generation based on a *partial value*. For 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 (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 carefully targeted hacks. A partial value is represented as a Haskell value containing undefined:

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

Suppose, now, that this value were passed into a backward interpretation of bst from §1—where would things 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 can 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 exception handling mechanism:

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

As long as no exception occurs, the code works as before. If there is ever an exception, the current value is abandoned and the rest is generated via the generate interpretation. In other words, complete mixes both backward and forward styles of interpretation to achieve its goals.

This trick works best 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 may be able to work around this by making their predicates lazier. For example, one could imagining writing an optimistic type checking algorithm optimisticTypeOf that maximizes laziness by blindly trusting user-provided type annotations. The user could then use the reflective generator comap optimisticTypeOf type_ >>= expr to complete an incomplete term like App (Ann (Int :-> Int) undefined) (Ann Int undefined). The completer would successfully determine that the type of the whole expression is Int, and then it would have enough information to complete the undefineds with well-typed expressions.

7.3 Reflective Producers

Weighted random generation in the style of QuickCheck is not the only way to get test inputs: both enumeration [Braquehais 2017; Runciman et al. 2008] and guided generation [Fioraldi et al. 2020; Zalewski 2022] have been explored as alternatives. Indeed, much of the PBT literature has moved from talking about *generators* to talking about *producers* of test data, where the specific strategy does not matter [Paraskevopoulou et al. 2022; Soremekun et al. 2020]. We use the language of "generators" here because it is familiar and concrete, but reflective generators might better be considered as *reflective producers* because they can also be used in these other styles.

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

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

for "roughly size-based" enumeration, leaning heavily on the combinators and techniques found in LeanCheck [Braquehais 2017]. 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 experiments are promising. For example, `enumerate (bst (1, 10))` reaches size-4 BSTs before its 10th enumerated value and matches the size order of an idiomatic LeanCheck enumerator given in Appendix E.

While fuzzing is sometimes treated as a separate topic from PBT—focused on finding crash failures by generating inputs external to a system rather than finding more subtle errors in individual functions—a number of recent projects have attempted to bridge the gap, and reflective generators may offer a useful unifying framework for such efforts. We already saw that reflective mutators are helpful in the context of HypoFuzz-style mutation; 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++ [Fioraldi et al. 2020; Zalewski 2022]. Since Crowbar already uses a free-monad-like structure to represent its generators, we can imagine writing an equivalent reflective generator interpretation that works the same way. More generally, reflective generators can be used in any producer algorithm that uses a generator as a parser.

8 RELATED WORK

This work builds on the ideas of 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 better way to represent effectful code in pure languages. While our implementation remains faithful to the 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 other bidirectional abstractions.

The concrete realization of reflective generators is also related 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 a different type for bind that does not normalize as aggressively.

The idea of reflective generators was originally sparked by the tools developed in *Inputs from Hell* [Soremekun et al. 2020], and these tools in turn tie into the broader world of grammar-based generation [Aschermann et al. 2019; Godefroid et al. 2008, 2017; Holler et al. 2012; Srivastava and Payer 2021; Veggalam et al. 2016; Wang et al. 2019]. Grammar-based approaches are less expressive than monadic ones, since they can only generate strings from a context-free grammar, and therefore cannot generate complex data structures with internal dependencies.

Replicating example distributions is a classic problem in *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 ideas in the Hypothesis framework [MacIver and Donaldson 2020; MacIver et al. 2019], but Hypothesis-style internal test-case reduction is not the only approach to automated shrinking. As we mentioned in §6, QuickCheck provides `genericShrink`, which provides a competent shrinker for any Haskell type that implements `Generic`. While `genericShrink` is a decent starting point, it fails to shrink unstructured data (like strings) and values with complex preconditions. Another alternative is provided by Hedgehog [Stanley [n. d.]], a QuickCheck competitor in the Haskell ecosystem, which implements automatic shrinking by combining shrinking with generation. Generators contain information about how to shrink the values they generate, with composite generators relying on sub-generators to shrink sub-structures. This is related to reflective shrinking, but it requires a more programmer effort.

9 CONCLUSION AND FUTURE DIRECTIONS

Reflective generators are a powerful abstraction for producing and manipulating test inputs. We have developed their theory and demonstrated 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 build on reflective generators, automating their creation and improving their usability.

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 hope that, in a large variety of cases, the annotations can be *synthesized*.

We plan to work with Hoogle+ [James et al. 2020], using its type-based synthesis algorithm to obtain candidate programs for the annotations with no user intervention. This is an especially compelling opportunity because it is easy to tell whether annotations are correct: they must be sound and complete, as described in §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 changing an annotation increases the number of passing tests, it is more likely to be correct; if the change causes more tests to fail, it is likely wrong. If this idea works, it could make transitioning from QuickCheck generators to reflective generators almost entirely automatic.

Usability. We have taken care to design an API for reflective generators that aligns with existing QuickCheck functions and minimizes programmer effort. Our own experience writing reflective generators studies has been positive, and, except for the aforementioned limitations (§4.3), we ran into no issues upgrading existing generators. The automation techniques hypothesized above could make reflective generators even more usable. Still, we certainly do not constitute a representative sample of PBT users: the usability of reflective generators should be studied empirically.

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 on a thorough usability analysis of reflective generators. Inspired by prior work [Coblenz et al. 2021], we hope our analysis will be useful for both assessing and refining our design.

REFERENCES

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.

Jean-Yves Girard. 1986. The system F of variable types, fifteen years later. *Theoretical Computer Science* 45 (Jan. 1986), 159–192. https://doi.org/10.1016/0304-3975(86)90044-7

Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 206–215. https://doi.org/10.1145/1375581.1375607

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. 2012. Fuzzing with code fragments. In *Proceedings of the 21st USENIX conference on Security symposium (Security'12)*. USENIX Association, USA, 38.

John Hughes. 2019. How to Specify It!. In *20th International Symposium on Trends in Functional Programming*. https://doi.org/10.1007/978-3-030-47147-7_4

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. 2012. Typed Tagless Final Interpreters. In *Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, Jeremy Gibbons (Ed.). Springer, Berlin, Heidelberg, 130–174. https://doi.org/10.1007/978-3-642-32202-0_3

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.

Daan Leijen and Erik Meijer. 2001. Parsec: Direct Style Monadic Parser Combinators For The Real World. (2001), 22. http://www.cs.uu.nl/research/techreps/repo/CS-2001/2001-35.pdf

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

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. *ACM SIGPLAN Notices* 44, 2 (Sept. 2008), 37–48. https://doi.org/10.1145/1543134.1411292

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

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

APPENDIX

A POLYMORPHIC INTERPRETATION FUNCTION

```
interpret ::
  forall p d c.
  (PartialProf p, forall b. Monad (p b)) =>
  (forall b a. [(Weight, Choice, Reflective b a)] -> p b a) ->
  Reflective d c ->
  p d c
interpret p = interp
  where
    interp :: forall b a. Reflective b a -> p b a
    interp (Return a) = return a
    interp (Bind r f) = do
      a <- interpR r
      interpret p (f a)

    interpR :: forall b a. R b a -> p b a
    interpR (Pick xs) = p xs
    interpR (Lmap f r) = lmap f (interpR r)
    interpR (Prune r) = prune (interpR r)
```

B PROOFS OF LEMMA 4.1 (LAWS)

This appendix proves the equations from Lemma 4.1.

```
(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
```

Using the following relevant definitions:

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

data R b a where
  Pick :: [(Weight, Choice, Reflective b a)] -> R b a
  Lmap :: (c -> d) -> R d a -> R c a
  Prune :: R b a -> R (Maybe b) a
  ChooseInteger :: (Integer, Integer) -> R Integer Integer
  GetSize :: R b Int
  Resize :: Int -> R b a -> R b a

type Reflective b = Freer (R b)

instance Monad (Reflective b) where
  return = Return
  Return x >>= f = f x
  Bind u g >>= f = Bind u (g >=> f)

prune :: Reflective b a -> Reflective (Maybe b) a
prune (Return a) = Return a
prune (Bind x f) = Bind (Prune x) (prune . f)

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

dimap :: (c -> d) -> (a -> b) -> Reflective d a -> Reflective c b
dimap _ g (Return a) = Return (g a)
dimap f g (Bind x h) = Bind (Lmap f x) (dimap f g . h)
```

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

Proof of $(PMP3)$. By rewriting.

```
(lmap f . prune) (return y)
  = {- def. return -}
  (lmap f . prune) (Return y)
  = {- def. prune (Return case) -}
  lmap f (Return y)
  = {- def. lmap -}
  dimap f id (Return y)
```

```
      = {- def. dimap (Return case) -}
      Return y
      = {- def. return -}
      return y
```

Thus (*PMP*3) holds. □

Proof of (*PMP*4). By induction over the structure of x.

Case `x = Return a`:

```
        (lmap f . prune) (Return a >>= g)
          = {- def. >>= (Return case) -}
          (lmap f . prune) (g a)
          = {- re-bracket -}
          (lmap f . prune . g) a
          = {- def. >>= (Return case) -}
          Return a >>= (lmap f . prune . g)
          = {- def. return -}
          return a >>= (lmap f . prune . g)
          = {- PMP3 -}
          (lmap f . prune) (return a) >>= (lmap f . prune . g)
          = {- def. return -}
          (lmap f . prune) (Return a) >>= lmap f . prune . g
```

Case `x = Bind r h`:

```
        (lmap f . prune) (Bind r h >>= g)
          = {- def. >>= -}
          (lmap f . prune) (Bind r (h >=> g))
          = {- def. prune + lmap -}
          Bind (Lmap f (Prune r)) (lmap f . prune . (h >=> g))
          = {- IH -}
          Bind (Lmap f (Prune r)) (lmap f . prune . h >=> lmap f . prune . g)
          = {- def. >>= -}
          (Bind (Lmap f (Prune r)) (lmap f . prune . h)) >>= (lmap f . prune . g)
          = {- def. prune + lmap -}
          (lmap f . prune) (Bind r h) >>= lmap f . prune . g
```

Thus (*PMP*4) holds. □

C PROOF OF LEMMA 4.4 (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, where the definition of `reflect'` is as follows:

```
reflect' :: Reflective b a -> b -> [a]
reflect' = interp
  where
    interp :: Reflective b a -> b -> [a]
    interp (Return x) _ = return x
    interp (Bind r f) b = interpR r b >>= \x -> interp (f x) b

    interpR :: R b a -> b -> [a]
    interpR (Lmap f r') b = interpR r' (f b)
    interpR (Prune  r') b = maybeToList b >>= \b' -> interpR r' b'
    interpR (Pick   gs) b = gs >>= (\ (_, _, g) -> interp g b)
```

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`
`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 >>= \ x -> reflect' (f x) 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 >>= \ x -> generate (f x))`.
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 ∈ maybeToList b >>= \ b' -> interpR`$_{\texttt{reflect'}}$ `r' b'`.
Thus, ∃`b'` such that `a ∈ interpR`$_{\texttt{reflect'}}$ `r' b'`
By IH$_R$, `a ~ interpR`$_{\texttt{generate}}$ `r' b'`.
Thus, `a ~ interpR`$_{\texttt{generate}}$ `(Prune r')`.

Case `r = Pick gs`:
Assume `a ∈ interpR`$_{\texttt{reflect'}}$ `(Pick gs) b`.
Thus, `a ∈ (gs >>= \ (_, _, g) -> reflect' g b)`.

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

This completes the proof. □

D GENERATOR FOR JSON DOCUMENTS

Note: 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.

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

label :: String -> Reflective b ()
label s = labeled [(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 =
  labeled
    [ ("array", array),
      ("object", object)
    ]

-- object  =  "{"  "}"  |  "{"  members "}"  ;
object :: Reflective String String
object =
  labeled
    [ ("'{' '}'", 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 =
  labeled
    [ ("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 =
  labeled
    [ ("'[' ']'", 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 =
  labeled
    [ ("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 =
  labeled
    [ ("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 =
  labeled
    [ ("'\"' '\"'", 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 =
  labeled
    [ ("char_", (: []) <$> focus _head char_),
      ("char_ chars", (:) <$> focus _head char_ <*> focus _tail chars)
    ]

-- char_ = digit | unescapedspecial | letter | escapedspecial ;
char_ :: Reflective Char Char
char_ =
  labeled
    [ ("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 = labeled (map (\c -> ([c], exact c)) letters)

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

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

-- number = int_  frac  exp  |  int_  frac  |  int_  exp  |  int_  ;
number :: Reflective String String
number =
  labeled
    [ ("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_ =
  labeled
    [ ("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 =
  labeled
```

```
      [ ("digit", (: []) <$> focus _head digit),
        ("digit digits", (:) <$> focus _head digit <*> focus _tail digits)
      ]

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

  -- nonzerodigit  =  "3"  |  "4"  |  "7"  |  "8"  |  "1"  |  "9"  |  "5"  |  "6"  |  "2"  ;
  nonzerodigit :: Reflective Char Char
  nonzerodigit =
    labeled (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 =
    labeled
      [ ("'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+"))
      ]
```

E 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]]
```

F JSON WITH HASH CODE GENERATOR

```
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 >>- \_ ->
            return (a ++ payload ++ b ++ hashcode ++ c)
  where
    hash = foldl' (\h c -> 33 * h `xor` fromEnum c) 5381
    consume s = lmap (take (length s)) (exact s)
```

A reflective generator for JSON objects with a hashcode, not expressible with a grammar-based generator. The generator produces a payload, then computes its hash, and then assembles the larger JSON object containing both.

G 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"
  }
}
```

H REFLECTIVE GENERATOR FOR POLYMORPHIC LAMBDA CALCULUS TERMS