Bigenerators, Exposed!

SAMANTHA FROHLICH, University of Bristol, UK
HARRISON GOLDSTEIN, University of Pennsylvania, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA
MENG WANG, University of Bristol, UK

Random data generators like those used with QuickCheck [2] can do more than just sample from probability distributions over test inputs. For example, in their paper on *Composing Bidirectional Programs Monadically*, Xia *et al.* showed how to turn a generator into a *bigenerator*—a generator that can also be "run backward" to determine if a value is in its range [19]. Bigenerators that enforce nontrivial preconditions on the values that they generate can be used as checkers of those preconditions. But this idea of running generators backward can do even more.

We propose *exposed bigenerators*, a generalization of Xia et al.'s abstraction that are enriched with expressive annotations to expose the generator's choice points for analysis and control. Exposed bigenerators can be interpreted in a myriad of ways to address key challenges in property-based testing.

We propose solutions to three familiar problem areas from the property-based testing literature—generator validation, test suite analysis, and dynamic testing strategies—all of which build on the same abstraction. A single exposed bigenerator program can (1) validate its own soundness, completeness, and distributional properties; (2) analyse properties of existing test suites, including constructor frequencies and *combinatorial coverage* in the style of *Do Judge a Test by its Cover* [4]; and (3) Implement the core of dynamic testing strategies, including replicating and expanding on the example-based tuning presented in *Inputs from Hell* [18] and the adaptive fuzzing of FuzzChick [9]. All of this can be done with a few modest changes to the generators that QuickCheck users are already used to writing.

[Note: The writing in this draft is still a bit rough. When reading, feel free to focus on higher-level things and skip more granular issues for now.]

1 INTRODUCTION

Random data generators are a staple of property-based testing (PBT). They provide a vast array of examples that can validate properties of a system much more thoroughly than the few examples a programmer might select while unit-testing. In QuickCheck [2], generators are created using a monadic domain specific language (DSL). Users write compositional programs in the `Gen` monad to generate complex random data structures that satisfy any required preconditions by construction. This is an elegant solution that has proven itself extremely effective [1, 5], but challenges remain.

This paper focuses on three of those challenges: generator validation, test suite analysis, and dynamic testing strategies. While they seem completely unrelated, we will show that all three of these issues can be addressed by the same abstraction: a generalisation of the QuickCheck generator interface.

Generator Validation. Conventional wisdom says that testing code should be as simple as possible to reduce the likelihood of bugs in the test harness. Unfortunately, generators for values with non-trivial preconditions are often quite complicated programs that are not obviously correct at a glance. A generator might produce values that are invalid, or, more insidiously, it might miss important parts of the valid space of inputs. Relatedly, random generators induce a distribution over test values, and it can be frustratingly difficult to achieve the right one. Our framework addresses these concerns and gives programmers new ways to understand the behaviour of their generators.

ICFP'22, September 11–16, 2022, Ljubljana, Slovenia
2022.

Test Suite Analysis. Many potential metrics exist for understanding testing efficacy, and no single one gives a complete picture. Thus, it is best to use a variety of metrics to analyse a test suite. One compelling class of metrics is black-box *combinatorial* analyses. The simplest metric of this class is constructor frequency (how frequently constructor of a sum type appears in a test suite), which gives a high-level signal of potential gaps in the test distribution. More involved metrics like the combinatorial coverage metric presented in *Do Judge a Test by its Cover* give deeper way to understand how well a test suite covers the space of possible inputs [4]. These metrics are often computed in an ad-hoc way for each data type, but our framework provides these analyses no extra code.

Dynamic Testing Strategies. While vanilla QuickCheck works for many properties, it is sometimes necessary to use more guided or adaptive strategies to find bug-provoking inputs. Many fantastic approaches to dynamic property-based testing exist [10, 15, 17], including *Inputs from Hell*, which allows for guided testing with a distribution that is based on a programmer-supplied suite of interesting inputs [18], and FuzzChick, which brings ideas from adaptive fuzzing to the world of QuickCheck-style generators [9]. These kinds of ideas are usually considered separately, but our framework is able to replicate (and expand on) the core ideas behind both of these techniques, *again* without any extra programmer effort.

How could one framework possibly provide a unified solution to such disparate problems? The first key step is to make QuickCheck generators *bidirectional*. *Composing bidirectional programs monadically*, shows that, with the help of some ideas from bidirectional programming, generators can be run backward [19]. For Xia et al., running a generator backward means giving it a value and asking if it could have generated that value. This technique allows generators can be used as checkers of the properties that they enforce. For example, a generator for binary search trees (BSTs) can also be used to check if a given tree satisfies the BST invariants. This convenient feature only requires a few lightweight annotations in the generator that witness how each generation step was done. Incredibly clever, but it is only the tip of the iceberg.

We add to the bigenerator framework with one more small set of annotations. These annotations mark the choices that the generator might make, and allow those choices to be retrieved when the generator is run backward. *Exposing* a generator's choices like this gives unprecedented access to information about how generators produce a given value, and it unlocks solutions a myriad of problems including those listed above. This exploration and expansion of bigenerators is important groundwork for a new revolution in property-based testing, and we hope that this paper will encourage the development of even more applications.

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

- We expand the analytic power of bigenerators further by annotating them with semantic choice labels, creating *exposed bigenerators*. (§3)
- We present novel ways to use bigenerators for introspection: bigenerators can be used to validate their own correctness and distributional properties, giving insights that would otherwise require tedious manual calculations. (§4)
- We show that our new exposed bigenerators can replicate coverage measurements like those from *Do Judge a Test by its Cover* [4]. (§5)
- We use exposed bigenerators backward and forward, implementing core ideas behind two existing testing frameworks. We replicate and expand on the example-based tuning presented in *Inputs from Hell* [18], and the adaptive fuzzing of FuzzChick [9]. (§6)

All in all, we give seven interpretations of exposed bigenerators, each of which provides value to a tester for a low cost that is paid only once. We conclude with related work (§7) and ideas for future research (§8).

2 BACKGROUND

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

2.1 QuickCheck Generators

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

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

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

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

During testing, QuickCheck generates hundreds or thousands of integers and trees, checking that `prop_insertValid x t` is true for all of them. However, the likelihood of the tree generator randomly stumbling on enough valid BSTs to pass the `isBST t` check is low, making the property vacuously true, and the testing insincere. Luckily, libraries like QuickCheck provide means for creating better generators that can, for example, enforce the `isBST t` predicate by construction:

```
genBST :: (Int, Int) -> Gen Tree
genBST (lo, hi) | lo >= hi = return Leaf
genBST (lo, hi) = frequency
                  [ ( 1, return Leaf    ),
                    ( 5, genNode (lo, hi) ) ]


genNode (lo, hi) = do
  x <- chooseInt (lo, hi)
  l <- genBST (lo, x - 1)
  r <- genBST (x + 1, hi)
  return (Node l x r)


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

This example QuickCheck generator is written in monadic style using QuickCheck combinators. Its definition revolves around the desired invariant, using an argument to ensure that only integers within the correct range are used. The base case uses the monadic **`return`** to construct a degenerate generator that always produces a `Leaf`. Then, the inductive case uses the `frequency` combinator to make a weighted choice between the `Leaf` and `Node` constructors, where the nodes are generated using do notation to naturally compose together the recursive call for child trees, and the combinator `chooseInt` for selecting its value.

2.2 Bigenerators

Generators like `genBST`, which are carefully designed to only produce values that satisfy a given precondition, bake in a lot of logic about exactly how that precondition is (and is not) satisfied. Ideally, that information could be used for other purposes. This is exactly the idea behind one of the examples in *Composing Bidirectional Programs Monadically* [19]. Xia et al. make QuickCheck-style generators *bidirectional*: they can run forward as generators, and they can also run backward as a checker of the precondition that the generator enforces.

To enable backward interpretation, the monad is combined with a profunctor as the `Profmonad` class, which just insists that something is both a **`Monad`** and a `Profunctor`. Profunctors are bifunctors that are contravariant in their first parameter, and covariant in the second. In other words, the second parameter acts like a normal functor, but the first flips the arrows, applying functions "backward", making them intuitively the perfect structure for encapsulating bidirectionally.

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

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

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

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

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

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

```
biGenNode (lo, hi) = do
  x <- (dimap nodeValue id  . toFailureP) (biChooseInt (lo, hi))
  l <- (dimap nodeLeft  id  . toFailureP) (biGenBST (lo, x - 1))
  r <- (dimap nodeRight id  . toFailureP) (biGenBST (x + 1, hi))
  return (Node l x r)
```

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

3 EXPOSED BIGENERATORS

With a bidirectional generator that has been composed monadically, this is where we take the baton from Xia et al., as we believe there is more to be explored here. Our key insight is to extract the idea of "generator choices" into a type class:

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

This type class represents a structure with choice points and marks the way these choices can be made. Each choice in the list is paired with a semantic label or *tag*, `t`, and there is much to be gained by exposing this information. This is the extra expressive power that we arm bigenerators with to create *exposed bigenerators*:

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

The `ExpBiGen` class packages the aforementioned classes together giving us monadic profunctors (`Profmonad`) that can internalise failure (`ProfunctorPartial`) and have a notion of choice (`Pick`). A unified interface like `ExpBiGen` allows exposed bigenerators to be written using a "classy" (or tagless final [7]) embedding. Exposed bigenerators written in our framework do not have a concrete type like `Gen`, instead they have a universally quantified type like `forall g. ExpBiGen g t b a`, meaning that exposed bigenerators can be interpreted in many ways, examples of which will be explored soon.

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

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

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

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

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

[1]For those interested in optics [3], the function is the `match` of the prism corresponding to the data type being generated.

The `at` combinator provides this function from automatically derived prisms (such as `_Node` in the example).

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

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

Since this type is a wrapper around `Gen`, it is a monad, and it is trivially a profunctor (since contravariant parameter is ignored). This means that the only nontrivial instance needed to get an `ExpBiGen` is `Pick`:

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

The implementation of `pick` ignores the tags using **`snd`**, then picks between the list of exposed bigenerators using `oneof`.

With these instances, the universally quantified type of `genBST` can be specialised to `QC.Gen` using `run` and used as a generator:

```
     genBST (-10,10)  :: forall g. ExpBiGen g => g      String Tree Tree
run (genBST (-10,10)) ::                          QC.Gen String Tree Tree

> QC.sample (run (genBST (-10,10)))
Leaf
> QC.sample (run (genBST (-10,10)))
Node Leaf (-4) (Node Leaf 10 Leaf)
```

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

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

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

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

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

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

Interpreting `genBST (-10, 10)` using wrapper function `unGenCheck` works as intended:

```
unGenCheck :: (forall g. ExpBiGen g => g t a a) -> a -> Bool
unGenCheck x a = isJust (run x a)
```

```
> unGenCheck (genBST (-10,10)) Leaf
True
> unGenCheck (genBST (-10,10)) (Node Leaf (-4) (Node Leaf 10 Leaf))
True
> unGenCheck (genBST (-10,10)) (Node Leaf 13 Leaf)
False
```

The beauty of this approach is that there is no need to stop here. The existing bigenerator interpretation is an important proof of concept, but now other interpretations are just a few type class instances and annotations away. Even better, exposed bigenerators have rich information stored in their tags that neither of the above interpretations capitalize on. There is much more to do.

4 GENERATOR VALIDATION

While conventional testing wisdom recommends that testing code be as simple as possible, it is hard to avoid the fact that generators are complex programs. As such, it is often useful to be able to validate that a generator behaves as expected. This section describes two interpretations of bigenerators that allow for *introspection*: they give the tester the ability to query properties of its own distribution.

4.1 Testing Generators

For the sake of this subsection, assume that generator code is untrusted—it may have bugs or subtle mistakes. Also assume that there exists a trusted function for checking the precondition that the generator's values are supposed to satisfy. This scenario is the opposite of the one assumed in *Composing bidirectional programs monadically*, where there the generator is trusted and a checking function does not yet exist. Now the `UnGenCheck` instance checks whether a generator could have produced a value, not whether that value upholds the invariant that the generator enforces. In practice, both scenarios are common, but since many precondition functions can be written to be far simpler than the associated generator (since the generator needs to be written with distributional properties in mind [2, 6]), the scenario with a trusted checker and an untrusted generator might actually be more realistic.

Validating a generator with respect to a property can be broken down into two parts: soundness and completeness. Soundness promises that every value generated by the generator is valid. Completeness asserts that all valid values are generated.

Checking whether a generator is sound with respect a precondition is easy, and does not actually require bidirectionality. To gain confidence that `genBST` is sound, the tester can write a property that checks that any tree produced by `genBST` satisfies `isBST`:

```
prop_genBSTSound = forAll (gen genBST) (\t -> isBST t)
```

This is recommended for all QuickCheck generators, and the bigenerator formalism does not get in the way.

Completeness is hard to check for normal generators, because there is no automatic way to know if a generator can produce a particular value. With bigenerators, this is not a problem! In order to check that `isBST` is complete, a tester can do the following:

(1) Use an automated tool [13, 14] to derive a basic generator, `genTree`, for values of type `Tree`. (Remember that `genTree` would not be a great choice for testing program properties, since many of the values it produces will not satisfy `isBST`.)
(2) Check the property:

```
prop_genBSTComplete = forAll genTree (\t -> isBST t ==> unGenCheck genBST t)
```

> This ensures that for every value `t` of type `Tree`, if `isBST t` then `genBST` can generate `t`. (Note that this testing is not vacuous (§3) because the `forAll` combinator is used.)

This works for the same reason that `unGenCheck` works for Xia et al.: running the generator as a checker is really just asking if the value is in the generator's range.

Of course, step (2) is quite slow: it requires that `genTree` stumble upon valid values without any programmer guidance. But remember, this testing can be done once-and-for-all. When the tester is happy with the number of checked examples, there is no need to re-check, and future testing can continue with the confidence that `genBST` does not miss important areas of the input space.

These techniques are an important first-pass when evaluating the quality of a generator. They ensure that the generator can produce the expected range of values. But these testing schemes do not confirm that the generator has the right *distribution*. We address that next.

4.2 Probabilistic Analysis

John Hughes, one of the creators of QuickCheck, has noted (2007) that "it is meaningless to talk about random testing without discussing the distribution of test data" Without distributional knowledge, a tester might happily rest on their laurels, thinking that their program is thoroughly tested, when in truth some important part of the input space is so unlikely to be covered that it is essentially untested.

To this end, QuickCheck provides tools for tracking and reporting the kinds of inputs that are used to check a given property. These features are invaluable for testers, but they are not a complete solution. For one thing, they require per-property analysis, when distributional properties are often a problem that can be solved once and for all at the level of a generator. In addition, it can be difficult to know things about particular inputs without running a massive number of tests. Ideally, a generator could report some properties of its own distribution. Upgrading to bigenerators enables generators to do just that.

When bigenerators are run backward, they can do more than just check whether a value could have been generated: they can also report the *probability* that a value was generated. This is what `UnGenProb` does:

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

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

This information is accessed using the function:

```
unGenProb :: (forall g. ExpBiGen g => g t a a) -> (t -> Weight) -> a -> Probability
unGenProb x w a =
  case run x (a, w) of
    Nothing -> 0
    Just (_, p) -> p
```

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

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

5 TEST SUITE ANALYSIS

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

5.1 Understanding Generator Choices

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

5.1.1 Listing Choices. The annotations in an exposed bigenerator like `genBST` mark the different choices that the generator might make. For example `"node"` marks the choice to make a new `Node`, and `"5"` marks the choice of the value `5` in a `Node`. Thus, when generating the value `Node Leaf 5 Leaf`, the generator makes choices:

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

This process of turning an unstructured string of choices into into structured data looks a lot like *parsing*, and it is well known bidirectionalising a parser yields a *pretty-printer* [3, 11, 16, 19]. What does this mean for exposed bigenerators?

The following function takes an exposed bigenerator and a value and "pretty-prints" the list of choices that the generator makes when producing the value:

```
unGenList :: (forall g. ExpBiGen g => g t a a) -> a -> [[t]]
unGenList x a = snd <$> run x a
```

Note that the return type is `[[t]]`, because there might be multiple ways of getting to a particular value. This is a design choice—it is also possible make `unGenList` return `Maybe [t]`, taking the "leftmost" choice sequence, or even `Maybe ([t], Probability)`, taking the most likely choice sequence. We choose the list-of-lists presentation for simplicity.

As with previous examples, `unGenList` is implemented using a data structure that is an instance of `ExpBiGen`:

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

The `b` parameter represents the value that is run backward through the generator, and the `[(a, [t])]` is a list of potential outputs and the choices that lead to them.

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

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

The end result works as desired, with:

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

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

The **`Monad`** and `Pick` instances above use three different operations to assemble choice lists: `[]`, `(++)`, and `(:)`. The first two are required for **`Monad`**, and the last for `Pick`. A suitable generalization needs to capture all three of these operations.

Since list is the free monoid, it is natural to capture `[]` and `(++)` with the `Monoid` operations `mempty` and `mempty`. Unfortunately, `Monoid` does not give an alternative to `(:)`, a bit more structure is needed:

```
class (forall a. Ord a => Monoid (f a)) => MonoidInject f where
  inject :: Ord a => a -> f a -> f a

instance MonoidInject [] where
  inject = (:)
```

This class abstracts the notion of `(:)` to other types. (Ignore the **`Ord`** constraints for now, those are important for the next section.) With this new type class, it is possible to generalise `UnGenList` to `UnGenChoices`:

```
newtype UnGenChoices c t b a =
  UnGenChoices {run :: (Ord t, MonoidInject c) => b -> [(a, c t)]}

unGenChoices :: (Ord t, MonoidInject c) => (forall g. ExpBiGen g => g t b a) -> b -> [c t]
unGenChoices x b = fmap snd . ($ b) . run $ x
```

The type `[t]` has been replaced by `c t` for any container `c` that is `MonoidInject`. The instances are updated accordingly, replacing `(++)` with `mempty`, `[]` with `mempty`, and `(:)` with `inject`—everything else stays the same.

This more general interface is just as easy to use, and specialises to the type of `unGenList` from before:

```
unGenList' :: Ord t => (forall g. ExpBiGen g => g t b a) -> b -> [[t]]
unGenList' = unGenChoices
```

Next, we show how to use `unGenChoices` to analyze a suite of tests.

5.2 Evaluating Test Suite Coverage

Among the many ways that testers estimate how well a given test suite might catch bugs, *black-box* methods are some of the most versatile: they can be applied even if there is no way to access or instrument its internals of the system under test (e.g., because it is compiled, closed-source, etc.), and they rely only on properties of the test suite itself. This section describes two different black-box metrics and shows how exposed bigenerators can help make those metrics easier to compute.

5.2.1 Choice Frequencies. One simple, effective way to understand a test suite is measuring the frequencies of each data constructor of a type. For BSTs, this means counting the relative frequencies of nodes and leaves, as well as measuring the distribution of values across a set of trees. When data types are very simple, there are ways to automatically tune a generator to ensure a uniform distribution over constructors [14], but in general it not easy to know if a generator produces a reasonable distribution of constructor frequencies. In this area, we make two contributions: we refine constructor frequency to *choice* frequency, and we give an automatic way to compute choice frequencies for a test suite.

Moving from constructor frequencies to choice frequencies is a subtle change, but a useful one. Consider a generator for de Bruijn-indexed simply-typed lambda calculus (STLC) terms, which needs to select between available free variables:

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

The labels on the variable indices are all the same, `""`. There is no reason to prefer one free variable over another, so it would be counter-productive to distinguish such granular choices. As a result, a choice frequency measurement correctly ignores such details (simply giving a frequency for the empty tag, which can be ignored), whereas a naïve constructor frequency measurement does not. Critically, this is how the programmer would likely have written their exposed bigenerator anyway, so these considerations are handled without extra effort.

Automatically computing choice frequencies is easy, given the machinery from the previous section. The type `Frequencies` represents a map from choices frequency of that choice, and is defined as:

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

instance Monoid (Frequencies a) where
  mempty = Frequencies Map.empty
  mappend (Frequencies x) (Frequencies y) = Frequencies (Map.unionWith (+) x y)

instance MonoidInject Frequencies where
  inject x (Frequencies xs) = Frequencies (Map.insertWith (+) x 1 xs)

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

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

5.2.2 Combinatorial Coverage. Moving beyond choice counting, exposed bigenerators can also measure more nuanced test suite metrics like *combinatorial coverage*. Combinatorial coverage comes from the systems literature [8]. Originally, it was used to improve testing for systems with large numbers of simple configuration parameters, asking the question "how well does a particular set of configurations exercise the system"? The key insight is that a single test case, say `config = [True, False, True False]` for a list of four Boolean configuration parameters, simultaneously tests a combinatorial explosion of "smaller" test-cases. Testing with `config` tests the case where

```
  params[0] == True && params[1] == False
```

and the case where

```
  params[2] == False && params[3] == True
```

and all other two-way combinations. We say that `config` "covers" those two-way "descriptions". Measuring combinatorial coverage is useful, because knowing that all t-way descriptions have been covered means that no bug depends on fewer than $t + 1$ parameters simultaneously—the tester can categorically rule out certain shapes of bugs.

This simple notion of t-way description does not work well for algebraic data types because it does not take structure into account, but *Do Judge a Test by its Cover* presents a more general definition [4]. Instead of looking at relationships between configuration values, Goldstein et al.'s notion of coverage measures relationships between the data constructors in the type. In order to compute this generalised combinatorial coverage, values must be converted into a tree of constructors of type:

```
data Term a = Term a (Cons a)

newtype Cons a = Cons [Term a]
  deriving (Semigroup, Monoid)
```

This is conceptually simple to do, but it is somewhat tedious and requires some non-trivial choices about which constructors should "count" towards coverage. This hassle can be avoided with the help of exposed bigenerators and a few lines of code:

```
instance MonoidInject Cons where
  inject x xs = Cons [Term x xs]
```

The `MonoidInject` instance for `Term` does the obvious thing: every `pick` creates a new root of the constructor tree, and every `(>>=)` concatenates constructors at the same level of the tree. As a simple example:

```
  > unGenChoices (genBST (-10, 10)) (Node Leaf 5 Leaf) :: [Term String]
  [Term "node" (Cons [Term "5" (Cons []), Term "leaf" (Cons []), Term "leaf" (Cons [])])]
```

This all *just works*. The type of `unGenChoices` specialises to the desired type, and the tree is computed automatically. Even better, this is a tree of choices, not a tree of constructors, so it has the same advantages that choice frequencies have over constructor frequencies. In particular, the natural tagging scheme for exposed bigenerators appropriately ignores the impact of uninteresting constructors (like variable indices) in the tree.

To wrap up the story, a `Term` can be passed to a function that computes the list of descriptions covered by that `Term`:

```
covers :: Ord a => Int -> Term a -> [Description a]
```

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

6 DYNAMIC TESTING STRATEGIES

Forward and backward interpretations do not need to be used in isolation. Combining the two can lead to interesting new applications such as the replication and expansion of *Inputs from Hell* [18] and *FuzzChick* [9].

6.1 Inputs from Hell

Inputs from Hell describes a method for generating both common and uncommon inputs from examples, which may do a better job at tickling bugs than purely random inputs, at least in some circumstances [18]. While the method described by Soremekun et al. focuses on probabilistic grammars, and is therefore not directly applicable to programmatic generators, there is still hope. Exposed bigenerators can replicate the setup from *Inputs from Hell* and, in doing so, generalise the approach to work with monadic generators that enforce preconditions.

6.1.1 Replicating Inputs from Hell. The process of generating common and uncommon inputs from examples works like this:

(1) Extract choice weights from a set of examples.
(2) (Optionally) if "uncommon" examples are desired, invert the weights.
(3) Use the weights to guide the generation of new test values.

Implementing this process concretely requires a generator that can generate based on an assignments of weights to choice tags. A "weight map" biases values that are generated:

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

weightedGen :: (forall g. ExpBiGen g => g t a a) -> (t -> Int) -> Gen a
weightedGen = run
```

The weight map is represented by a function from the semantic tags of the generator to weights. For example, `genBST` can be biased to always make leaves:

```
alwaysLeaves = weightedGen (genBST (-10,10)) (\t -> if t == "leaf" then 1 else 0)

> QC.sample alwaysLeafs
Leaf
Leaf
...
```

The backward counterpart of `WeightGen` has already come up—it is `UnGenFrequencies` from §5. The `unGenWeights` function is just like `UnGenFrequencies`, but it only cares about the first set of weights extracted from a given value (the **`listToMaybe`** does this a lovely type safe manner, wrapping the result in a **`Maybe`**):

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

Using `Frequencies` here means that the weight map is now represented as a key-value map. This means that the weights can be adjusted programmatically, making inversion possible when generating uncommon examples. Using functions in the other direction was a design choice that improves user experience because, with functions, users have access to wildcard pattern matches and functions like **`const`**. The key-value weight maps can be converted to function weight maps for generation via the following function:

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

Together, `weightedGen` and `unGenWeights` can replicate the example from *Inputs from Hell* where expressions of the following type are generated:

```
data Expr = Term Term | Plus Expr Term | Minus Expr Term
data Term = Factor Factor | Times Term Factor | Div Term Factor
data Factor = Digits Digits | Pos Factor | Neg Factor | Parens Expr
data Digits = Digit Char | More Char Digits
```

An exposed bigenerator is needed for each of these types. As an example, here is the one for terms:

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

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

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

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

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

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

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

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

As expected, the weight map features representative occurrences of 1, 2, and 3, but no other numbers. Values generated from this weight map using `weightedGen` can be expected to only use these numbers, following the lead of `i`:

```
iReplicate :: Gen Expr
iReplicate = weightedGen (genExpr 4) (weightMap . fromJust $ iWeights)

> QC.sample (fmap print iReplicate)
"1+3"
"1"
"3*1+(1)*(2+1)"
```

These common inputs are clearly derived from `i` and match those made by the probabilistic grammars of *Inputs from Hell*. Success! Uncommon inputs are also achievable by inverting the weights in `iWeights`.

The weights of multiple examples can be amalgamated using the following function:

```
mine :: Ord t => (forall g. ExpBiGen g => g t a a) -> [a] -> Frequencies t
mine g = foldr ((<>) . fromJust . unGenWeights g) mempty
```

It uses the semigroup operation to sum together the weights of tags from the different examples.

6.1.2 Expanding Inputs from Hell. Since exposed bigenerators are monadic, they can tackle more complicated examples. For example, the `Expr` generators can be upgraded to omit divisions by zero. This new and improved version of `genTerm` does just that:

```
genTerm :: (ExpBiGen g) => Int -> g String Term Term
genTerm 0 = _Factor <$$> genFactor 0
genTerm n =
    pick
      [ ("factor", _Factor <$$> genFactor (n - 1)),
        ("times", _Times <$$> genTerm (n - 1) <**> genFactor (n - 1)),
        ( "div",
          do
            x <- genTerm (n - 1) `at` (_Div . _1)
            y <- genFactor (n - 1) `at` (_Div . _2)
            pure $ case evalFactor y of
              Just v | v /= 0 -> Div x y
              _ -> x
        )
      ]
```

In the `Div` case, the applicative style has been dropped in lieu of the more expressive monadic do notation. Now the generation of `Div` terms deals with each argument to the division separately, and uses an evaluator of `Expr` terms to ensure that the divisor is never zero.

6.2 FuzzChick

Another dynamic testing strategy that exposed bigenerators can help with is adaptive fuzzing. Adaptive fuzzers find and exploit interesing inputs via random mutation. Rather than generate new test inputs over and over again, the fuzzer keeps a pool of "seed" values that it deems interesting and mutates those values to try to find other interesting values that are conceptually "near-by".

Traditional fuzzers mutate values by flipping or swapping random bits—they generally operate on unstructured binary data—but FuzzChick brings adaptive fuzzing to functional programming with the help of structured mutation [9]. FuzzChick mutates its seed values at a much higher level than bit flips: it might cut off a branch of a tree, replace a node, or swap children in a term. The code that does these mutations is generally synthesised from type information, which is fantastic for many use-cases, but what if the random values need to satisfy a precondition? Unsurprisingly, exposed bigenerators can help.

6.2.1 Mutating with Trees. As with the previous example, doing precondition-preserving mutation requires two interpretations of the same bigenerator, one forward and one backward. At a high level, the process goes like this:

(1) Take a seed value and run it backward through the generator to get choices.
(2) Mutate the choices randomly.
(3) Run the choices forward through the generator to get a new value.

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

Unfortunately, the naïve implementation of this process does not quite work as intended. For example, suppose choices are extracted using `unGenList`, and a mutation consists of simply replacing one choice in the list for another. The term `Node (Node Leaf 2 Leaf) 5 (Node Leaf 7 Leaf)` run backward through `genBST` yields choices:

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

And those choices might be mutated to:

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

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

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

and removes *both* children to produce `Node Leaf 5 Leaf`. Ultimately, working with lists of choices is simply too unstructured.

The solution is one final instance of the `MonoidInject` class:

```
data FMonoidInject a
  = MEmpty
  | MAppend (FMonoidInject a) (FMonoidInject a)
  | Inject a (FMonoidInject a)
```

This structure is a bit different from previous instances: instead of implementing some semantically interesting version of each required operation, it simply acts as a syntax tree of `MonoidInject` operations. As the name suggests, it is essentially the "free" `MonoidInject`. The instances reinforce this intuition, translating abstract operations to concrete syntax:[2]

```
instance Monoid (FMonoidInject a) where
  mempty = MEmpty
  mappend = MAppend

instance MonoidInject FMonoidInject where
  inject = Inject
```

Calling `unGenChoices` with the appropriate type hints yields `FMonoidInject` trees that mimic the structure of the generator that produced them. These are perfect for mutation, since the tree-like structure enables targeted mutations that only affect the part of the tree that they intend to. This infrastructure can be used to implement a number of useful mutation strategies.

[Note: The rest of this section will be expanded significantly in the next week or so. We're still ironing out exactly which kinds of mutations are possible, and how best to achieve them in a generic way.]

6.2.2 Manipulating Choices. Step (2) above requires a way to manipulate a tree of choices. There are a number of mutations that might be reasonable, depending on the specific type being mutated. One option is to simply "drop" some node in the tree, forcing that choice to be re-computed. The function `dropMut` does this uniformly at random throughout the tree:

```
dropMut :: FMonoidInject a -> Gen (FMonoidInject (Maybe a))
```

The type of tag changes from `a` to **`Maybe`** `a` to accommodate a dropped node somewhere in the middle of the tree; any consumer of this mutated choice tree must handle a **`Nothing`** choice in some reasonable way.

There are also more advanced approaches that might work better or worse in different domains. The following function swap two compatible subtrees that start with the same choice:

```
swapMut :: FMonoidInject a -> Gen (FMonoidInject (Maybe a))
```

Even more advanced swapping would be possible, so long as the programmer provides bit more information about which choices are interchangeable. [Note: We will expand on this before submission.]

[2]Note that technically these instances are *unlawful*. Monoid properties like associativity of mappend do not hold, because no simplification is done. As this data structure is internal to our library, we are not concerned with this detail.

6.2.3 Regeneration Strategies. Once the choices have been mutated, the final step is to regenerate a value from the new choices. For the most part, this operation is quite simple; it just walks over a tree of choices and makes those choices at each choice point. However, the mutations above are not guaranteed to produce valid trees of choices: `dropMut` explicitly makes choices invalid by setting them to `Nothing`, and `swapMut` may put choices in the "wrong place" (e.g., any swap of `Node` constructors in a BST results in an invalid tree). This means that any `ExpBiGen` interpretation needs some strategy for dealing with incorrect or missing choices.

Making the Smallest Choice. One sensible way to deal with an invalid choice is to simply make the smallest possible available choice instead. In the case of a range of numbers that might mean the lowest, and in the case of a list of constructors, the one with the fewest arguments. This can be achieved with a type like `SmallReGen`:

```
newtype SmallReGen t b a =
  SmallReGen {run :: FMonoidInject (Maybe t) -> a}

smallReGen :: Eq t => SmallReGen t a a -> FMonoidInject (Maybe t) -> a
smallReGen = run
```

The implementation of the `ExpBiGen` instances are actually quite simple. The "smallest" choice is determined by the structure of the generator: the implementer must take care to list the smallest choices first in every call to `pick`. This requirement is not ideal (it could lead to termination issues if care is not taken), but in our experience this is the most natural way to write generators anyway. If the programmer handles this detail, then `SmallReGen` can just greedily make the first choice available whenever no good choice is supplied by the input.

As apparently myopic as `SmallReGen` is, it actually does produce some interesting and useful mutations. In the case of a binary search tree, it can chop off branches (which may help to expose bugs that depend on certain shapes of trees) and it can also *minimise* portions of a tree (which may trigger bugs in certain value-dependent corner cases.)

Making a Random Choice. Another approach, when faced with an impossible choice, is just to pick an option at random. The choices can be made uniformly, or, in the case of `MutReGen`, with the help of a weighting function:

```
newtype WeightedReGen t b a =
  WeightedReGen {run :: (t -> Int, FMonoidInject (Maybe t)) -> QC.Gen a}

weightedReGen :: Eq t => WeightedReGen t a a -> (t -> Int) -> FMonoidInject (Maybe t) -> Gen a
weightedReGen g w b = run g (w, b)
```

This regeneration strategy can do some really cool things. For example, consider mutating a value in a BST. Depending on what new value is chosen, it is likely that some of that node's children will now be invalid, but that is not an issue—the mutator can make random valid choices for the children's values as well! In this way, `MutReGen` can propagate a mutation down a BST in a way that preserves validity. Even better, this complex mutation scheme did not require any user code other than the original exposed bigenerator; it fell out for free. This approach also supercedes the previous one, since it could always randomly choose the smallest choice.

There are likely other useful mutators and strategies as well. One could imagine using heuristics like the ones in the Mutagen framework to come up with new strategies that work well for different classes of data-type [12]. Ultimately, this framework is a blank slate that can be adapted and tuned depending on an adaptive fuzzer's needs.

7 RELATED WORK

[Note: Work in progress.]

8 CONCLUSIONS AND FUTURE WORK

Exposed bigenerators lay the foundation for a new age in property-based testing. They provide a unified solution to challenges in generator validation, test suite analysis, and dynamic testing strategies, without requiring significant extra programmer effort. Using our framework, testers can annotate their QuickCheck-style generators with the appropriate backward maps and semantic tags to gain access to a wealth of powerful testing tools that go far beyond just random generation.

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

Automating Shrinking. There are a couple of interesting applications in shrinking that have not yet been explored. First, it seems likely that tools like `SmallReGen` in §6 could be useful for automatically shrinking certain data types. It would go almost like mutation, extract choice, shrink those choices, and then regenerate, but extra care would need to be taken to ensure that the resulting values are really shrinks and not unrelated values. This seems quite easily within reach.

Building on this idea, exposed bigenerators might also improve shrinking outside of Haskell. The Hypothesis library [20] in Python already does shrinking automatically by remembering the generator choices that produced a value and shrinking those choices. But what happens if a value is generated, manipulated in some way, and then used as a test? The choices get lost and there is no good way to shrink. Exposed bigenerators would give a way to retrieve choices from any value in the range of the generator, regardless of how disconnected that value is from the original generation process.

Learning from Choices. The *Inputs from Hell* example illustrates one way that exposed bigenerators can be used for a kind of "learning". In that case, example inputs are treated as training data to learn a simple model of choice weights. But that learning is quite simple—is it possible to combine exposed bigenerators with more sophisticated learning algorithms?

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

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

The main advantage this has over the approach in §6 is expressivity. Choice weights cannot express relationships between different parts of a data structure, and thus may not capture certain aspects of the examples. In contrast, an appropriate langauge model could learn to generate values that really capture essence of the examples.

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

REFERENCES

[1] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*. 2–10.

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

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

[4] Harrison Goldstein, John Hughes, Leonidas Lampropoulos, and Benjamin C. Pierce. 2021. Do Judge a Test by its Cover. In *Programming Languages and Systems: 30th European Symposium on Programming, ESOP 2021, Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021 (Lecture Notes in Computer Science, Vol. 12648)*. 264–291. https://link.springer.com/chapter/10.1007%2F978-3-030-72019-3_10

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

[6] John Hughes. 2019. How to Specify It! *20th International Symposium on Trends in Functional Programming* (2019).

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

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

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

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

[11] Kazutaka Matsuda and Meng Wang. 2018. FliPpr: A System for Deriving Parsers from Pretty-Printers. *New Generation Computing* 36, 3 (2018), 173–202. https://doi.org/10.1007/s00354-018-0033-7

[12] Agustín Mista. 2021. MUTAGEN: Faster Mutation-Based Random Testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 120–122.

[13] Agustín Mista and Alejandro Russo. 2019. Deriving compositional random generators. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*. 1–12.

[14] Agustín Mista, Alejandro Russo, and John Hughes. 2018. Branching processes for QuickCheck generators. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, Nicolas Wu (Ed.). ACM, 1–13. https://doi.org/10.1145/3242744.3242747

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

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

[17] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, Andy Gill (Ed.). ACM, 37–48. https://doi.org/10.1145/1411286.1411292

[18] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. 2020. Inputs from Hell Learning Input Distributions for Grammar-Based Test Generation. *IEEE Transactions on Software Engineering* (2020).

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

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