

Advancing Property-Based Testing in Theory and Practice

Dissertation Proposal

Harrison Goldstein

University of Pennsylvania

September 28, 2022

Advisor

Benjamin C. Pierce, Prof. University of Pennsylvania, USA

Thesis Committee

Stephanie Weirich, *Committee Chair*, Prof. University of Pennsylvania, USA

Mayur Naik, Prof. University of Pennsylvania, USA

Andrew Head, Asst. Prof. University of Pennsylvania, USA

Hila Peleg, *External Committee Member*, Asst. Prof. The Technion, Israel

Contents

1	Introduction and Background	4
1.1	Background and Related Work	6
1.1.1	Property-Based Testing in General	6
1.1.2	The Valid Generation Problem	7
1.1.3	Monadic Generators	9
2	Parsing Randomness	12
2.1	High-Level Story	14
2.2	Free Generators	15
2.2.1	Representing Free Generators	15
2.3	Derivatives of Free Generators	16
2.4	Free Generators	18
2.4.1	Background: Monadic Parsers	18
2.4.2	Representing Free Generators	19
2.4.3	Interpreting Free Generators	24
2.5	Derivatives of Free Generators	28
2.5.1	Background: Derivatives of Languages	28
2.5.2	The Free Generator Derivative	29
3	Reflecting on Random Generation	32
3.1	Background	34
3.1.1	Bigenerators	34
3.2	Reflective Generators	36

3.2.1	The Reflective Abstraction	37
3.2.2	Interpretations	38
3.2.3	Ramping Up	40
3.3	Example-Based Tuning	42
3.3.1	Inputs from Hell, Briefly	43
3.3.2	Tuning Reflective Generators	43
3.4	Test-Case Mutation	46
3.4.1	Value Preserving Mutation	47
3.4.2	Mutating with Reflective Generators	48
3.5	Limitations	52
4	Bringing Fuzzing into Focus (<i>Speculative</i>)	55
4.1	Proposed Project	56
4.1.1	An Aside about Rust	56
4.1.2	Part 1: Gradually Constrained Generation	57
4.1.3	Part 2: Fuzzing Properties	60
5	Some Problems with Properties	61
5.1	Preliminary Study	62
5.1.1	Study Population	62
5.1.2	Interview Design and Questions	62
5.1.3	Thematic Analysis	64
5.2	Planned Work: Full-Scale Study	68
5.2.1	Research Questions	68
5.2.2	Study Population	69
5.2.3	Interview Plan	70
6	The Promise of Properties (<i>Speculative</i>)	71
6.1	Follow-On Studies	71
6.2	User Centered Design of Developer Tools	72
6.3	Education	73

6.4 Solving a Testing Problem at Jane Street	74
7 Conclusion	75
7.1 Timeline	76

Chapter 1

Introduction and Background

In 2022, software is not just ubiquitous, it is inescapable. A person may interact with some kind of computer every minute from the moment they wake up to the moment they go to sleep. As software permeates our lives, so do the bugs the software contains, threatening our time, our money, our privacy, and even our health.

Luckily, techniques for ensuring the correctness of software have grown alongside the software industry. A modern programmer has a wealth of tools available to help them avoid potentially catastrophic errors. One such tool, property-based testing (PBT) combines formal software specification with random testing to achieve low-effort, high-impact bug-finding. It has seen enormous success, finding critical bugs in telecommunications software [1], replicated file-stores [21], cars [2], and a range of other programs [19]. Select companies make PBT part of their testing workflow and find it to be an invaluable tool for ensuring correctness [7].

But the PBT community must not rest on its laurels: there is still work to be done. The literature has identified a number of challenges that make PBT difficult or time consuming to apply in certain testing situations. One critical class of challenges concerns the random data generators that testers use as part of the PBT process. Successful PBT relies on random data that is both well distributed and “valid” for the system under test. I have spent the last few years exploring ways to make effective random generators more powerful and easier to write, and I will continue that work in my dissertation.

Additionally, it is clear that there are also “unknown unknowns”: challenges facing PBT, particularly those important to software developers, that the research community is not yet aware of. I

have begun a series of need-finding studies, consisting of semi-structured interviews with developers, to uncover and hopefully solve some of these hidden challenges. This work will form the second major theme of my dissertation.

In my dissertation I intend to understand and solve both known and unknown challenges in PBT, making it more usable and valuable for the software developers who work with it.

This document begins with a review of background on PBT and a discussion of relevant work in the area. After that I discuss the following research projects:

- **Parsing Randomness** I formalize the relationship between parsing and random generation using a new abstraction called *free generators*. While free generators can be used on their own as a tool for generation, this thesis is most interested in the way that free generators decouple generators from randomness.
- **Reflecting on Random Generation** I present *reflective generators*, which combine ideas from free generators with ones from the bidirectional programming literature. Reflective generators can be run both forward, to generate test inputs, and backward, to reflect on the *choices* the generator makes when generating a particular input. I show that reflective generators enable a variety of improvements to PBT, including novel approaches to validity-preserving mutation and shrinking.
- **Bringing Fuzzing into Focus (*Speculative*)** I present a plan to narrow the divide between PBT and fuzzing by unifying the approaches in a single framework. At the core of this idea is a new testing setup that combines an off-the-shelf fuzzer with reflective generators to achieve (hopefully) high-performance fuzzing and PBT.
- **Some Problems with Properties** I observe that while improving testing performance is important, there are likely other things that software developers need from PBT. I discuss a pair of semi-structured interview studies (inspired by work in the Human Computer Interaction) that aim to discover new ways for the research community to add value to PBT.
- **The Promise of Properties (*Speculative*)** I describe a speculative plan for work that follows from the the studies *Some Problems with Properties*. Of course, the whole point of the need-finding studies is to determine which future projects will be most valuable, so it is far

too early to commit to any one idea. That said, the preliminary study did generate a number of ideas that seem worth pursuing.

Finally, I conclude with a proposed timeline for this work and some closing thoughts.

Current Progress The first paper I published in my PhD, *Do Judge a Test by its Cover* [15], published at ESOP’21, does not appear on this list. If anyone on the committee would like to see it as part of the thesis, please let me know. Otherwise I do not plan on including it, as I do not feel it fits as well with the overall picture.

The following chapters each have progress bars to indicate the current state of each line of work. At the time of writing *Parsing Randomness* is accepted to OOPSLA’22 and I consider it to be essentially “complete” work. A completed draft of *Reflecting on Random Generation* was sent to, and unfortunately rejected from, ICFP’22 and Haskell’22; I plan to improve the evaluation section and send it to ICFP’23, hopefully with more success. *Some Problems with Properties* discusses ongoing work: we have completed a preliminary study, and we are currently conducting interviews for the full-scale study. A HATRA’22 paper announcing this project is currently under submission, and we eventually hope to publish the results of the study at a top-tier conference in either programming languages, software engineering, or human computer interaction. Finally, *Bringing Fuzzing into Focus* and *The Promise of Properties* are speculative: at least one of them chapters (and potentially both) will be written up and submitted to a prestigious conference. Additional details on the split between complete and incomplete work can be found in the following sections, and I present a high-level timeline near the end of this document.

1.1 Background and Related Work

In this section I present the background necessary to understand the contributions of my dissertation.

1.1.1 Property-Based Testing in General

Property-based testing (PBT) is a form of random testing [16] that was popularized by the QuickCheck [10] library in Haskell. In PBT, users write executable functions that act as partial specifications of a function under test. For example, a tester might write the following property to specify the `insert` function for an implementation of a binary search tree (BST):

```
prop_insertCorrect x t = isBST t ==> isBST (insert x t)
```

This property is an ordinary function that takes an integer and a tree as input and asserts that if
80 the original tree satisfies the BST invariant, then so does the tree after inserting the integer. There are a myriad of different kinds of properties that one might use to specify a system; a good source of examples is Hughes [20].

With a property in hand, a tester then passes a series of inputs to the property and checks that the property evaluates to **True** for each input. If some input causes the property to fail,
85 then it constitutes a *counterexample* to the property and potentially a bug in the program. If no counterexample is found after the tester’s time budget has elapsed, the tester will have gained confidence that the property holds in general.

In this dissertation I focus on the case of random PBT, where the series of inputs that check the property are generated by some random process, but there are alternatives. In particular,
90 *enumerative testing* advocates deterministically listing every possible input from the smallest to the largest, relying on the “small scope hypothesis” [22] that most bugs can be found with “small” inputs. There are a variety of systems that use this technique [37, 8]. Forms of model checking can also be considered alternatives to random PBT; their approaches vary but at their core model checkers are often tools for searching an input space for counterexamples [5].

Despite the success of the enumerative testing and model checking, random generation is still
95 a dominant player in the PBT space. Its success is often attributed to the fractal nature of “big” test cases—many structures are self-similar, so testing with one big structure also tests the code with the exponentially many smaller sub-structures. This effect is powerful, but it still takes a bit of work to get random generation “right.” Many have pointed out that the distribution that inputs
100 from drawn from is incredibly important for effective testing, so programmers are often forced to choose a distribution manually [10]. Even more importantly, testers need to ensure that the sampled inputs are *valid* to test with; I discuss this problem in the next section.

1.1.2 The Valid Generation Problem

Note that the property above has a *precondition* (which I also refer to equivalently as a *validity*
105 *condition* or *input constraint*): the property is vacuous if the input tree is not a valid BST. This

can be problematic, because many preconditions are difficult to satisfy randomly; in the case above we would need to randomly stumble on valid BSTs, which is very unlikely for trees that are more than a few nodes deep. I call this the *valid generation problem*. In the next section I will discuss the standard solution to this problem in PBT—hand-written *monadic generators*—but first here is
110 a short overview of other options.

The solutions to the valid generation problem fall on a spectrum from fully automatic to fully manual. The most automatic possible approach is to simply generate inputs as normal, ignoring preconditions entirely, and throw away any that do not satisfy the precondition. As I mentioned above, this is extremely inefficient when the precondition is *sparse*, or unlikely to be satisfied randomly.

115 Equally automatic, but often more effective, are coverage-guided approaches like those used by *fuzzers*. Fuzzers like AFL [45] measure the code coverage achieved by each input they try. If an input does not achieve coverage of new program branches, it is discarded, but if the input is deemed “interesting,” then it may be mutated and tried again in an attempt to explore even more new code paths. This process makes finding valid inputs more likely because optimizing for code
120 coverage encourages exploration of paths deeper in the program (i.e., past the point where the precondition is checked). Unfortunately, this only works so well, and again begins to fail with very sparse preconditions.

A different approach that is *almost* entirely automatic is TARGET [30]; it uses search strategies like hill climbing and simulated annealing to guide generation to more useful inputs. Löschner and
125 Sagonas’s approach works well when the notion of validity is in some way continuous (i.e., an input is not simply invalid, it is $X\%$ valid), but this is often not the case.

Some approaches use machine learning to automatically generate valid inputs. LEARN&FUZZ [14] generates valid data using a recurrent neural network. This solution seems to work best when a large corpus of inputs is already available and the validity condition is more structural than semantic. In
130 the same vein, RLCHECK [36] uses reinforcement learning to guide a generator to valid inputs.

For preconditions that are primarily structural, *grammar-based fuzzing* provides a compelling (if slightly more manual) solution. In *Grammar-based whitebox fuzzing* [13] a context-free grammar (CFG) is used to constrain the fuzzer’s output. Over the years, many versions of this paradigm have been developed, including ones that use pre-written fragments of the input language to give
135 the mutator interesting things to work with [17] and ones that use genetic programming to find

more interesting inputs [41]. The newest grammar-based fuzzing approaches use the grammar to do more structured mutation of values [42, 39]. These ideas have also been integrated into PBT in the FuzzChick library [28].

Moving into the range of “mostly manual” solutions, there are a variety of solver-aided languages for expressing generators for data with preconditions. Dewey proposed using constraint logic programming (CLP) to define generators for interesting structures like Rust programs [11]. The LUCK language [27] also uses a solver, but a bespoke one, to define generators and validity predicates at the same time. Steinhöfel and Zeller [40] use a grammar and SMT-expressible constraints on a structure to generate precondition-satisfying values.

Finally, *Generating Good Generators* [29] an outlier on the spectrum: technically it is a manual approach since it requires users to first express their validity predicates as inductive relations in Coq, but in Coq it is likely that the user already has an inductive relation available! *Generating Good Generators* is a great approach in the situations where it applies.

1.1.3 Monadic Generators

The most manual, but arguably the most flexible, solution to the valid generation problem is write *monadic* generators.

We represent generators using *monads* [33]. A monad is a type constructor (e.g., `List`, `Maybe`, etc.) `M` equipped with two operations,

```
return :: a -> M a
```

and

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

(pronounced “bind”). Conceptually, `return` is the simplest way to put some value into the monad, while `bind` gives a way to sequence operations that produce monadic values.

We can use these operations to define `genTree` like we would in `QuickCheck`:

```
genTree :: Int -> Gen Tree
genTree 0 = return Leaf
genTree h = do
  c <- frequency [(1,False), (3,True)]
  case c of
```

```

165     False -> return Leaf
        True -> do
            x <- genInt
            l <- genTree (h - 1)
            r <- genTree (h - 1)
170     return (Node l x r)

```

We use the monadic operations (along with **frequency**) to generate a random tree of integers. The expression **return Leaf** is a degenerate generator that always produces the value **Leaf**—this is what we mean by the “simplest way to put a value into the **Gen** monad.”

Rather than use (**>>=**) explicitly, we use **do**-notation, where

```

175     do
        a <- x
        f a

```

is syntactic sugar for **x >>= f**. In the context of the **Gen** type, this operation samples from a generator **x** to get a value **a** and then passes it to **f** for further processing—this is what we mean by “sequencing operations.”

Expressiveness Relative to Other Abstractions Monadic parsers are maximally expressive. They can generate values satisfying arbitrary computable constraints (e.g., it is possible to write a monadic generator for well-typed System F terms), subsuming less powerful representations like probabilistic context-free grammars.

185 For example, the following monadic generator generates (only) valid binary search trees:

```

genBST :: (Int, Int) -> Gen Tree
genBST (lo, hi) | lo > hi = return Leaf
genBST (lo, hi) = do
    c <- frequency [(1,False), (3,True)]
190    case c of
        False -> return Leaf
        True -> do
            x <- genRange (lo, hi)
            l <- genBST (lo, x - 1)
            r <- genBST (x + 1, hi)
195    return (Node l x r)

```

The generator maintains the BST invariant by keeping track of the minimum and maximum values available for a given sub-tree and ensuring that all values to the left of a value are less and that

all values to the right of a value are greater. This kind of generator is impossible to express as a
200 stochastic CFG, since there is dependence between the choice of value \mathbf{x} and the choices of sub-
trees. Our examples are mostly focused on simple (non-dependent) generators to streamline the
exposition, but our theory applies to the full class of monadic generators with finitely supported
distributions.

Chapter 2

205 Parsing Randomness

Progress Paper accepted to OOPSLA'22.



Collaborators Benjamin C. Pierce

This section contains excerpts from *Parsing Randomness*, appearing at OOPSLA'22. The Haskell development and writing was all my own work, with help and guidance from Benjamin.

In this chapter, I formalize the relationship between parsing and random generation using a new abstraction called *free generators*. While free generators can be used on their own as a tool for generation, this thesis is most interested in the way that free generators decouple generators from
210 randomness.

* * *

“A generator is a parser of randomness...” It’s one of those observations that’s totally puzzling right up to the moment it becomes totally obvious: a random generator—such as might be found in a property-based testing tool like QuickCheck [10]—is a transformer from a series of random
215 choices into a data structure, just as a parser is a transformer from a series of characters into a data structure.

While this connection may be obvious once it is pointed out, few actually think of generators this way. Indeed, to our knowledge the framing of random generators as parsers has never been explored formally. The relationship between these fundamental concepts deserves a deeper look!

220 We focus on generators written in the *monadic* style popularized by the QuickCheck library, which that build random data structures by making a sequence of random choices; those choices are the key. Traditionally, a generator makes decisions using a stored source of randomness (e.g., a seed) that it consults and updates whenever it must make a choice. Equivalently, if we like, we can pre-compute a list of choices and pass it in to the generator, which gradually walks down the list
225 whenever it needs to make random decisions. In this mode of operation, the generator is effectively parsing the sequence of choices into a data structure!

To connect generators and parsers, we introduce *free generators*, syntactic data structures that can be interpreted as *either* generators or parsers. Free generators have a rich theory; in particular, we can use them to prove that a large class of random generators can be factored into a parser and
230 a distribution over sequences of choices.

Besides clarifying folklore, free generators admit transformations that do not exist for standard generators and parsers. A particularly exciting one is a notion of *derivative* which modifies a generator by asking the question: “what does this generator look like after it makes choice *c*?” The derivative previews a particular choice to determine how likely it is to lead to useful values.

235 In Section 2.1 below, we introduce the ideas behind free generators and the operations that can be defined on them. We then present our main contributions:

- We formalize the folklore analogy between parsers and generators using *free generators*, a novel class of structures that make choices explicit and support syntactic transformations (Section 2.4). We use free generators to prove that any finitely supported *monadic generator*
240 can be factored into a parser and a distribution over strings.
- We exploit free generators to transport an idea from formal languages—the *Brzowski derivative*—to the context of generators (Section 2.5).

In the full paper, we also present and evaluate an algorithm that uses free generators and their derivatives to generate inputs that satisfy executable preconditions. We elide those details here for
245 brevity.

2.1 High-Level Story

To set the stage, let's clarify the specific formulations of generators and parsers that we plan to discuss. Consider the following programs:

```
genTree h =
  if h = 0 then
    return Leaf
  else
    c ← frequency [(1, False), (3, True)]
    if c == False then return Leaf
    if c == True then
      x ← genInt ()
      l ← genTree (h - 1)
      r ← genTree (h - 1)
      return Node l x r

parseTree h =
  if h = 0 then
    return Leaf
  else
    c ← consume ()
    if c == 'l' then return Leaf
    if c == 'n' then
      x ← parseInt ()
      l ← parseTree (h - 1)
      r ← parseTree (h - 1)
      return Node l x r
    else fail
```

250 The program on the left, **genTree**, generates random binary trees of integers like

Node Leaf 5 Leaf and Node Leaf 5 (Node Leaf 8 Leaf),

up to a given height h , guided by a series of weighted random Boolean choices made using **frequency**. Each time the program runs, it produces a random tree—i.e., the program denotes a distribution over trees. Generators like these can describe arbitrary finitely supported distributions of values.

255 The program on the right, **parseTree**, parses a string into a tree, turning

n5l1l into Node Leaf 5 Leaf and n5ln8l1l into Node Leaf 5 (Node Leaf 8 Leaf).

It consumes the input string character by character with **consume** and uses the characters to decide what to do next. This program is deterministic, but its execution (and thus the final tree it produces) is guided by a string of characters it is passed as input. Parsers like these can parse

260 arbitrary computable languages.

These two programs are nearly identical in structure, and both produce the same set of values. The main difference lies in how they make choices: in **genTree** branches are taken at random, whereas in **parseTree** they are controlled by the input string.

This is the key observation that links generators and parsers. To make it more concrete, let us imagine how to recover the distribution of **genTree** h from **parseTree** h . We can do this by

choosing a string at random and then parsing it—if we choose strings with the correct distribution, then the result of parsing those strings into values will be the same as if we had run `genTree` in the first place.

Here, we want the distribution over strings given to `parseTree` to satisfy the weighting of the Boolean choices in `genTree`. That is, `n` should appear three times more often than `l`, since `True` is chosen three times more often than `False`.

2.2 Free Generators

With these intuitions in hand, let’s connect parsing and generation formally. First, we unify random generation with parsing by abstracting both into a single data structure; then we show that a structure of this form can be viewed equivalently as a generator or as a parser and a source of randomness.

2.2.1 Representing Free Generators

Our unifying data structure is called a *free generator*. Free generators are syntactic structures that can be interpreted as programs that either generate or parse. For example:

```

280 fgenTree h =
    if h == 0 then
        return Leaf
    else
        c ← pick [(1, l, return False), (3, n, return True)]
285 if c == False then return Leaf
    if c == True then
        x ← fgenInt ()
        l ← fgenTree (h - 1)
290 r ← fgenTree (h - 1)
        return Node l x r

```

The structure of this program is again very similar to that of `genTree` and `parseTree`. The call to `pick` on line 5 combines ideas from both the generator (capturing the relative weights of `False` and `True`) and the parser (capturing the labels `l` and `n` corresponding to different paths in the parser code). However, the meaning of `fgenTree` is very different from that of either `genTree` or `parseTree`. The operators in `fgenTree` are entirely syntactic, and the result of running `fgenTree h`

300 is simply an abstract syntax tree (AST).

The syntactic nature of free generators means that they can simultaneously represent generators, parsers, and more. In Section 2.4 we give several ways to interpret free generators. We write $\mathcal{G}[\![\cdot]\!]$ for the *random generator interpretation* of a free generator and $\mathcal{P}[\![\cdot]\!]$ for the *parser interpretation*. In other words,

$$305 \quad \mathcal{G}[\![\text{fgenTree } h]\!] \approx \text{genTree } h \quad \text{and} \quad \mathcal{P}[\![\text{fgenTree } h]\!] \approx \text{parseTree } h.$$

The interpretation functions walk the AST produced by `fgenTree` to recover the behavior of the generator and parser programs.

These two interpretations can be related, formally, with the help of one final interpretation function, $\mathcal{R}[\![\cdot]\!]$, the *randomness interpretation* of the free generator. The randomness interpretation
 310 produces the distribution of sequences of choices that the random generator interpretation makes. Now, for any free generator g , we have

$$\mathcal{P}[\![g]\!] \langle \$ \rangle \mathcal{R}[\![g]\!] \approx \mathcal{G}[\![g]\!]$$

where $\langle \$ \rangle$ is a “mapping” operation that applies a function to samples from a distribution. Since a large class of generators (monadic generators with a finitely supported distribution) can also be written as free generators, another way to read this theorem is that such generators can be factored
 315 into two pieces: a distribution over choice sequences (given by $\mathcal{R}[\![\cdot]\!]$), and a parser of those sequences (given by $\mathcal{P}[\![\cdot]\!]$).

This precisely formalizes the intuition that “A generator is a parser of randomness.” But wait, there’s more to come!

2.3 Derivatives of Free Generators

320 Since a free generator defines a parser, it also defines a formal language: we write $\mathcal{L}[\![\cdot]\!]$ for this *language interpretation* of a free generator. The language of a free generator is the set of choice sequences that it can parse.

Viewing free generators this way suggests some interesting ways that free generators might be manipulated. In particular, formal languages come with a notion of *derivative*, due to Brzozowski [9].

325 Given a language L , the Brzozowski derivative of L with respect to a character c is

$$\delta_c^{\mathcal{L}} L = \{s \mid c \cdot s \in L\},$$

that is, the set of all strings in L that start with c , with the first c removed.

We can apply the same intuition to parsers by considering the derivative of a parser with respect to c to be whatever parser remains after c has been parsed. Each consecutive derivative fixes certain choices within the parser, simplifying the program:

330

<pre> parseTree 10 = c ← consume() if c == 1 then return Leaf if c == n then x ← parseInt() l ← parseTree 9 r ← parseTree 9 return Node l x r else fail </pre>	$\delta_n^{\mathcal{L}}(\text{parseTree } 10) \approx$ <pre> x ← parseInt() l ← parseTree 9 r ← parseTree 9 return Node l x r </pre>	$\delta_5^{\mathcal{L}} \delta_n^{\mathcal{L}}(\text{parseTree } 10) \approx$ <pre> l ← parseTree 9 r ← parseTree 9 return Node l 5 r </pre>
--	--	--

The first derivative fixes the character `n`, ensuring that the parser will produce a `Node`. The next fixes the character `5`, which determines the value `5` in the final `Node`.

Free generators have a closely related notion of *derivative*, illustrated by an almost identical set of transformations:

335

<pre> fgenTree 10 = c ← pick [...] if c == False then return Leaf if c == True then x ← fgenInt() l ← fgenTree 9 r ← fgenTree 9 return Node l x r else fail </pre>	$\delta_n^{\mathcal{L}}(\text{fgenTree } 10) \approx$ <pre> x ← fgenInt() l ← fgenTree 9 r ← fgenTree 9 return Node l x r </pre>	$\delta_5^{\mathcal{L}} \delta_n^{\mathcal{L}}(\text{fgenTree } 10) \approx$ <pre> l ← fgenTree 9 r ← fgenTree 9 return Node l 5 r </pre>
--	--	---

But there is a critical difference between this series of derivatives and the ones for `parseTree`. Whereas the parser derivatives we saw could be thought of *intuitively* as a program transformation on parsers, the analogous transformation on free generators is readily computable! Just as we can

compute the derivative of a regular expression or a context-free grammar, we can compute the
 340 derivative of a free generator via a simple and efficient syntactic transformation.

In Section 2.5 we define a procedure, $\delta_c^{\mathcal{L}}$, for computing the derivative of a free generator and prove it correct, in the sense that, for all free generators g ,

$$\delta_c^{\mathcal{L}} \llbracket g \rrbracket = \llbracket \delta_c g \rrbracket.$$

In other words, the derivative of the language of g is equal to the language of the derivative of g . (See Theorem 1.)

345 2.4 Free Generators

We now turn to developing the theory of free generators.

2.4.1 Background: Monadic Parsers

In the background above, I present an overview of monads and give an example of a generator as a monad. It turns out that parsers are monads as well. The following program looks just like `genTree`
 350 but it parses a tree rather than generating one randomly:

```

parseTree :: Int -> Parser Tree
parseTree 0 = return Leaf
parseTree h = do
  c <- consume
  355 case c of
    l -> return Leaf
    n -> do
      x <- parseInt
      l <- parseTree (h - 1)
      360 r <- parseTree (h - 1)
      return (Node l x r)
  _ -> fail

```

Here, `return a` means “parse nothing and produce `a`”, and `x >>= f` means “run the parser `x` to get a value `a` and then run the parser `f a`.” Under the hood, we have:

```

365 type Parser a = String -> Maybe (a, String)

```

A `Parser` can be applied to a string to obtain either `Nothing` or `Just (a, s)`, where `a` is the parse result and `s` contains any extra characters. The `consume` function pulls the first character off of the string for inspection.

2.4.2 Representing Free Generators

370 Recall the example of a monadic generator from the background section. With those definitions in mind, we give the formal definition of a free generator.¹

Type Definition The actual type of free generators is based on a structure called a *freer monad* [25]:

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

This type looks complicated, but it is essentially just a representation of a monadic syntax tree. The constructors of `Freer` align almost exactly with the monadic operations `return` and `(>>=)`, providing syntactic forms that can represent the building blocks of monadic programs.

An eagle-eyed reader might notice that the type of `Bind` here is not quite an instance of the type
380 of `(>>=)` above—one would have expected to see

```
Bind :: Freer f a -> (a -> Freer f b) -> Freer f b
```

with `Freer f a` as the first argument. The version we use is equally powerful, but more convenient. We will see in a moment that syntax trees in a freer monad are normalized by construction.

But what is going on with this `f` that appears throughout `Freer`? The type constructor `f` is
385 a type of *specialized operations* that are specific to a particular monadic program. For example, programs in the `Gen` monad do not just use `return` and `(>>=)`, they also use a `Gen`-specific operation, `frequency`. Similarly, representing a `Parser` as a syntax tree requires a way to represent a call to `consume`. In general, `f a` should be a syntactic representation of an operation returning `a`. Thus, we might have a type representing a parser operation that returns a character:

```
390 data Consume a where
  Consume :: Consume Char
```

¹For algebraists: Free generators are “free” in the sense that they admit unique structure-preserving maps to other “generator-like” structures. In particular, the $\mathcal{G}[\![\cdot]\!]$ and $\mathcal{P}[\![\cdot]\!]$ maps are canonical. For the sake of space, we do not explore these ideas further here.

Since **Freer** is polymorphic over **f**, it can capture any specialized operation necessary to represent the syntax tree of a monad.

For free generators specifically, the specialized operation we need is called **pick**—we saw it in Section 2.1. Intuitively, **pick** subsumes both **frequency** and **consume**. We define the **Pick** operation with a data type (since free generators are syntactic objects) simultaneously with our definition of **FGen**, the type of *free generators*:

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

By defining **FGen** as **Freer Pick**, we are really saying that “**FGen** is a monad with operation **Pick**.”

The **Pick** operation takes a list of triples. The first element of type **Weight** represents the weight given to a particular choice; weights are represented by signed integers for efficiency, but for theoretical purposes we treat them as strictly positive. The type **Choice** can theoretically be any type that admits equality, but for the purposes of this paper we take choices to be single characters. This makes the analogy with parsing clearer. Finally, **Freer Pick a** is actually just the type **FGen a**! Thus we should view the third element in the triple as a *nested* free generator that is run iff a specific choice is made.

Together the elements of these triples represent both kinds of choices that we have seen so far, subsuming both the weighted random choices of generators and the input-directed choices of parsers. Depending on our needs, we can interpret **Pick** as either kind of choice. In the rest of the paper, we sometimes speak of free generators “making” or “parsing” a choice, but remember that this is really just an analogy—a free generator is simply syntax, and the interpretation comes later.

Our First Free Generator The **FGen** structure achieves our goal of unifying monadic generation and parsing, so let’s try writing a free generator. Following the basic structure of **genTree** and **parseTree**, we can start to define **fgenTree**:

```
fgenTree :: Int -> FGen Tree
fgenTree 0 = Return Leaf
fgenTree h = Bind
  (Pick [(1, l, Return False), (3, n, Return True)])
  (\c -> case c of
```

```

False -> Return Leaf
True  -> ... )

```

425 The first few lines are relatively easy to translate. The height checks are all the same as before, but now in the $h = 0$ case we produce the syntactic object `Return Leaf` rather than `return Leaf`, whose behavior depends on a particular implementation of `return`. When $h > 0$, we use `Bind` and `Pick` to specify that the generator has two choices: `False` (with weight 1, marked by character `l`) and `True` (with weight 3, marked by `n`).

430 But things get a bit more complicated when we get into the anonymous function passed as the second argument to `Bind`. In the `False` case we `Return Leaf` again, but in the `True` case the next step should be a call to `fgenInt`. We *could* look at the definitions of `genInt` and `parseInt` to determine the next choice, and then we *could* create a `Bind` node to make that choice, but that would be fairly tedious to do for every choice that the generator might eventually make. In general, 435 while `FGen` is the right type to capture free generators, its constructors are a bit cumbersome to write down directly.

Recovering Monadic Syntax Luckily, we can use the same monadic machinery used by `genTree` and `parseTree` to make free generators much easier to write. We can define `return` and `(>>=)` for `FGen` as follows, allowing us to use `do`-notation to write free generators:

```

440 return :: a -> FGen a
    return = Return

    (>>=) :: FGen a -> (a -> FGen b) -> FGen b
    Return a >>= f = f a
445 Bind p g >>= f = Bind p (\a-> g a >>= f)

```

The `return` operator maps directly to a `Return` syntax node, but there is a bit more going on in the definition of `(>>=)`. Specifically, `(>>=)` normalizes the structure of the computation, ensuring that there is always an operation at the “front.” The advantage of this is that it is always $O(1)$ to check if a free generator has a choice to make. There is no need to dig through the syntax tree to 450 determine the next step.

Another convenient way to manipulate free generators is via an operation called “`fmap`,” written `f <$> x`. Like `return` and `(>>=)`, `(<$>)` is a syntactic transformation, but intuitively `f <$> x` means “apply the function `f` to the result of generating/parsing with `x`”. We define it as:

```

    (<$>) :: (a -> b) -> FGen a -> FGen b
455 f <$> Return a = Return (f a)
    g <$> (Bind p f) = Bind p ((g <$>) . f)

```

(Note that all monads have an analogous operation; this will come in handy later.)

Representing Failure For reasons that will become clear in Section 2.5, it is useful to be able to represent a free generator that can “fail.” We call the always-failing free generator `void`, and define it like this:

```

460 void :: FGen a
    void = Bind (Pick []) Return

```

Any reasonable interpretation of this free generator must fail (by either diverging or returning a signal value); with no choices in the `Pick` list, there is no way to get a value of type `a` to pass to the second argument of `Bind`. Additionally, the use of `Return` as the second argument to `Bind` is irrelevant, since any free generator with no choices available will fail. This suggests that we can check if a free generator is certainly `void` by matching on an empty list of choices! In Haskell this is easy to do with a pattern synonym:

```

470 pattern Void :: FGen a
    pattern Void <- Bind (Pick []) _

```

This declaration means that pattern-matching on `Void` is equivalent to matching a `Bind` with no choices to make and ignoring the second argument. It is simple to define a function that uses this new pattern to check if a particular free generator is `void`:

```

475 isVoid :: FGen a -> Bool
    isVoid Void = True
    isVoid _    = False

```

While `void` is useful as an error case for algorithms that build free generators, it would be incorrect for a user to use `void` in a hand-written free generator. To enforce this constraint, we define a wrapper around `Pick` (called `pick`) that does a few coherence checks to make sure that the generator is constructed properly:

```

    pick :: [(Weight, Choice, FGen a)] -> FGen a
    pick xs =
        case filter (\ (_, _, x) -> not (isVoid x)) xs of

```

```

ys | hasDuplicates (map snd ys) -> undefined
485 [] -> undefined
ys -> Bind (Pick ys) Return

```

This function is partial: it yields `undefined` if the list passed to `pick` is invalid. (This is analogous to raising an exception in a conventional imperative language.) The first line filters out any choices that are equivalent to `void`, since making those choices would lead to failure. The second line checks
490 that the user has not duplicated any of the choice labels; this would introduce a nondeterministic choice that would complicate the interpretation considerably (see Section 3.5). Finally, the third line ensures that the generator we construct is not itself `void`. In practice, these checks ensure that the various interpretations of free generators presented in the remainder of this section work as intended.

495 **Examples** Now that we have seen the building blocks of free generators, let's look at a couple of concrete examples. First, we can finally write down an ergonomic version of `fgenTree`:

```

fgenTree :: Int -> FGen Tree
fgenTree 0 = return Leaf
fgenTree h = do
500   c <- pick [(1, l, return False), (3, n, return True)]
   case c of
     False -> return Leaf
     True  -> do
       x <- fgenInt
       505   l <- fgenTree (h - 1)
       r <- fgenTree (h - 1)
       return (Node l x r)

```

Remember, the `do`-notation here is no longer sequencing generators or parsers. Instead, each line of a `do`-block builds a new `Bind` node in a syntax tree. Similarly, `return` has no semantics, it only
510 wraps a value in the inert `Return` constructor. In this way `fgenTree` looks like both `genTree` and `parseTree`, but it does not behave like either (yet).

Trees are nice as a running example, but they are by no means the most complicated thing that free generators can represent. Here is a free generator that produces random (possibly ill-typed) terms of a simply-typed lambda-calculus:


```

fgenExpr :: Int -> FGen Expr
fgenExpr 0 = pick [ (1, i, Lit <$> fgenInt), (1, v, Var <$> fgenVar) ]
fgenExpr h =
  pick [ (1, i, Lit <$> fgenInt),
        (1, p, do
          e1 <- fgenExpr (h - 1)
          e2 <- fgenExpr (h - 1)
          return (Plus e1 e2)),
        (1, l, do
          t <- fgenType
          e <- fgenExpr (h - 1)
          return (Lam t e)),
        (1, a, do
          e1 <- fgenExpr (h - 1)
          e2 <- fgenExpr (h - 1)
          return (App e1 e2)),
        (1, v, Var <$> fgenVar) ]

```

Structurally `fgenExpr` is similar to `fgenTree`; it just has more cases and more choices. One stylistic difference between `fgenExpr` and `fgenTree` is that `fgenExpr` does not `pick` a coin and use it to decide what should be generated next; instead, it picks among a list of free generators directly. These styles of writing free generators are equivalent.

This version of the lambda calculus uses de Bruijn indices for variables and has integers and functions as values. This is a useful example because, while syntactically valid terms in this language are easy to generate (as we just did), it is more difficult to generate only well-typed terms.

2.4.3 Interpreting Free Generators

A free generator does not do anything on its own—it is just a data structure. To actually use these structures, we next define the interpretation functions that we mentioned in Section 2.1 and prove a theorem linking those interpretations together.

Free Generators as Generators of Values The first and most natural way to interpret a free generator is as a QuickCheck generator—that is, as a distribution over data structures. Plain QuickCheck generators ignore failure cases like `void` (they throw an error if there are no valid choices to make), but to make things a bit more explicit for our theory we use a modified generator monad: `Gen⊥`.²

²Our Haskell development implements partial generators with `Gen (Maybe a)`. We elide these details to streamline the presentation.

We define the *random generator interpretation* of a free generator to be:

```

 $\mathcal{G}[\cdot] :: \text{FGen } a \rightarrow \text{Gen}_{\perp} a$ 
 $\mathcal{G}[\text{Void}] = \perp$ 
535  $\mathcal{G}[\text{Return } v] = \text{return } v$ 
 $\mathcal{G}[\text{Bind (Pick } xs) f] = \text{do}$ 
     $x \leftarrow \text{frequency (map } (\backslash (w, \_, x) \rightarrow (w, \text{return } x)) xs)$ 
     $a \leftarrow \mathcal{G}[x]$ 
 $\mathcal{G}[f a]$ 

```

540 Note that the operations on the right-hand side of this definition do *not* build a free generator; they are Gen_{\perp} operations. This translation turns the syntactic form **Return** *v* into the semantic action “always generate the value *v*” and the syntactic form **Bind** into an operation that chooses a random sub-generator (with appropriate weight), samples from it, and then continues with *f*.

Note that $\mathcal{G}[\text{fgenTree } h]$ has the same distribution as **genTree** *h*.

545 **Free Generators as Parsers of Random Sequences** The *parser interpretation* of a free generator views it as a parser of sequences of choices. The translation looks like this:

```

 $\mathcal{P}[\cdot] :: \text{FGen } a \rightarrow \text{Parser } a$ 
 $\mathcal{P}[\text{Void}] = \backslash s \rightarrow \text{Nothing}$ 
 $\mathcal{P}[\text{Return } a] = \text{return } a$ 
550  $\mathcal{P}[\text{Bind (Pick } xs) f] = \text{do}$ 
     $c \leftarrow \text{consume}$ 
     $x \leftarrow \text{case find } ((== c) . \text{snd}) xs \text{ of}$ 
         $\text{Just } (\_, \_, x) \rightarrow \text{return } x$ 
         $\text{Nothing} \rightarrow \text{fail}$ 
555  $a \leftarrow \mathcal{P}[x]$ 
 $\mathcal{P}[f a]$ 

```

This time the **do**-notation on the right hand side is interpreted using the **Parser** monad (as before, defined as **String** \rightarrow **Maybe** (*a*, **String**)). In the case for **Bind**, the parser consumes a character and attempts to make the corresponding choice from the list provided by **Pick**. If it succeeds, it 560 runs the corresponding sub-parser and continues with *f*. If it fails, the whole parser fails.

Note that $\mathcal{P}[\text{fgenTree } h]$ has the same parsing behavior as **parseTree** *h*.

Free Generators as Generators of Random Sequences Our final interpretation of free generators represents the distribution with which the generator makes choices, ignoring how those choices are used to produce values. In other words, it captures exactly the parts of the structure that the

565 parser interpretation discards. We define the *randomness interpretation* of a free generator to be:

```

 $\mathcal{R}[\cdot] :: \text{FGen } a \rightarrow \text{Gen}_{\perp} \text{ String}$ 
 $\mathcal{R}[\text{Void}] = \perp$ 
 $\mathcal{R}[\text{Return } a] = \text{return } \varepsilon$ 
 $\mathcal{R}[\text{Bind (Pick xs) f}] = \text{do}$ 
570   (c, x) <- frequency (map (\ (w, c, x) -> (w, return (c, x))) xs)
   s <-  $\mathcal{R}[x >=> f]$ 
   return (c : s)
```

Again, we use `Gen⊥` and `frequency` to capture randomness and potential failure.

Factoring Generators These different interpretations of free generators are closely related to one another; in particular, we can reconstruct $\mathcal{G}[\cdot]$ from $\mathcal{P}[\cdot]$ and $\mathcal{R}[\cdot]$. That is, a free generator's random generator interpretation can be factored into a distribution over choice sequences plus a parser of those sequences.

To make this more precise, we need a notion of equality for generators like the ones produced via $\mathcal{G}[\cdot]$. We say two QuickCheck generators are *equivalent*, written $g_1 \equiv g_2$, iff the generators 580 represent the same distribution over values. This is coarser notion than program equality, since two generators might produce the same distribution of values in different ways.

With this in mind, we can state and prove the relationship between different interpretations of free generators:

Theorem 1 (Factoring). *Every free generator can be factored into a parser and a distribution over 585 choice sequences that are, together, equivalent to its interpretation as a generator. In other words, for all free generators g ,*

$$\mathcal{P}[g] \langle \$ \rangle \mathcal{R}[g] \equiv (\lambda x \rightarrow (x, \varepsilon)) \langle \$ \rangle \mathcal{G}[g].$$

Proof sketch. By induction on the structure of g ; see the paper for the full proof. □

Corollary 1. *Any monadic generator, γ , written using `return`, `(>>=)`, and `frequency`, can be factored into a parser plus a distribution over choice sequences.*

590 *Proof.* Translate γ into a free generator, g , by replacing `return` and `(>>=)` with the equivalent free generator constructs, and `frequency` with `pick`. (This will require choosing labels for each choice,

but the specific choice of labels is irrelevant.)

By construction, $\gamma = \mathcal{G}[[g]]$.

Additionally, g can be factored into a parser and a source of randomness via Theorem 1. Thus,

$$(\lambda x \rightarrow (x, \varepsilon)) \langle \$ \rangle \gamma = (\lambda x \rightarrow (x, \varepsilon)) \langle \$ \rangle \mathcal{G}[[g]] \equiv \mathcal{P}[[g]] \langle \$ \rangle \mathcal{R}[[g]],$$

595 and γ can be factored as desired. □

This corollary is what we wanted to show all along. Monadic generators are parsers of randomness.

Free Generators as Formal Language Syntax One final interpretation will prove useful. The *language of a free generator* is the set of choice sequences that it can make or parse. It is defined recursively, by cases:

```

 $\mathcal{L}[[\cdot]] :: \text{FGen } a \rightarrow \text{Set String}$ 
 $\mathcal{L}[[\text{Void}]] = \emptyset$ 
 $\mathcal{L}[[\text{Return } a]] = \varepsilon$ 
 $\mathcal{L}[[\text{Bind (Pick xs) f}]] = [ c : s \mid (w, c, x) \leftarrow xs, s \leftarrow \mathcal{L}[[x \gg= f]] ]$ 

```

605 This definition uses Haskell's list comprehension syntax to iterate through the large space of choices sequences in the language of a free generator. To determine the language of a `Bind` node, we look at each possible choice and then at each possible string in the language $\mathcal{L}[[x \gg= f]]$ obtained by continuing with that choice. (This recursion is well-founded as long as the language of the free generator is finite; by monad identities `Bind (Pick xs) f = Bind (Pick xs) Return >>= f`,
610 and `x` is strictly smaller than `Bind (Pick xs) Return`.) For each of these strings, we attach the appropriate choice label to the front. The end result is a list of all of the sequences of choices that, if made in order, would result in a valid output.

We can think of the result of this interpretation as the support of the distribution given by $\mathcal{R}[[g]]$. The language of a free generator is exactly those choice sequences that the random generator
615 interpretation can make and the parser interpretation can parse.

2.5 Derivatives of Free Generators

Next, we review the notion of Brzowski derivative from formal language theory and show that a similar operation exists for free generators. The way these derivatives fall out from the structure of free generators highlights the advantages of taking the correspondence between generators and
 620 parsers seriously.

2.5.1 Background: Derivatives of Languages

The *Brzowski derivative* [9] of a formal language L with respect to some choice c is defined as

$$\delta_c^{\mathcal{L}} L = \{s \mid c \cdot s \in L\}.^3$$

In other words, it is the set of strings in L that begin with c , with the initial c removed. For example,

$$\delta_a^{\mathcal{L}} \{\mathbf{abc}, \mathbf{aaa}, \mathbf{bba}\} = \{\mathbf{bc}, \mathbf{aa}\}.$$

Many formalisms for defining languages support syntactic transformations that correspond to

625 Brzowski derivatives. For example, we can take the derivative of a regular expression like this:

$$\begin{array}{ll} \delta_c^{\mathcal{L}} \emptyset = \emptyset & \nu^{\mathcal{L}} \emptyset = \emptyset \\ \delta_c^{\mathcal{L}} \varepsilon = \emptyset & \nu^{\mathcal{L}} \varepsilon = \varepsilon \\ \delta_c^{\mathcal{L}} \mathbf{c} = \varepsilon \quad (c = \mathbf{c}) & \nu^{\mathcal{L}} \mathbf{c} = \emptyset \\ \delta_c^{\mathcal{L}} \mathbf{d} = \emptyset \quad (c \neq \mathbf{d}) & \nu^{\mathcal{L}} \mathbf{c} = \emptyset \\ \delta_c^{\mathcal{L}} (r_1 + r_2) = \delta_c^{\mathcal{L}} r_1 + \delta_c^{\mathcal{L}} r_2 & \nu^{\mathcal{L}} (r_1 + r_2) = \nu^{\mathcal{L}} r_1 + \nu^{\mathcal{L}} r_2 \\ \delta_c^{\mathcal{L}} (r_1 \cdot r_2) = \delta_c^{\mathcal{L}} r_1 \cdot r_2 + \nu^{\mathcal{L}} r_1 \cdot \delta_c^{\mathcal{L}} r_2 & \nu^{\mathcal{L}} (r_1 \cdot r_2) = \nu^{\mathcal{L}} r_1 \cdot \nu^{\mathcal{L}} r_2 \\ \delta_c^{\mathcal{L}} (r^*) = \delta_c^{\mathcal{L}} r \cdot r^* & \nu^{\mathcal{L}} (r^*) = \varepsilon \end{array}$$

The $\nu^{\mathcal{L}}$ operator, used in the “.” rule and defined on the right, determines the *nullability* of an expression—whether or not it accepts ε . If r accepts ε then $\nu^{\mathcal{L}} r = \varepsilon$, otherwise $\nu^{\mathcal{L}} r = \emptyset$.

As one would hope, if r has language L , it is always the case that $\delta_c^{\mathcal{L}} r$ has language $\delta_c^{\mathcal{L}} L$.

³The superscript \mathcal{L} highlights that is the *language* derivative, distinguishing it from the generator derivative to be defined momentarily.

2.5.2 The Free Generator Derivative

630 To define derivatives of free generators, we first need a definition of *nullability* for free generators:

```

ν :: FGen a -> Set a
ν(Return v) = {v}
νg          = ∅      (g ≠ Return v)

```

Note that this behaves a bit differently than the $\nu^{\mathcal{L}}$ operation on regular expressions. For a regular
 635 expression r , the expression $\nu^{\mathcal{L}}r$ is either \emptyset or ε . Here, the null check returns either \emptyset or the
 singleton set containing the value in the **Return** node. That is, ν for free generators extracts a value
 that can be obtained by making no further choices. Another difference is that, for free generators,
 “can accept the empty string” and “accepts only the empty string” are equivalent statements; this
 greatly simplifies the definition of ν .

640 To see what the derivative operation might look like, we can write down some equations that it
 should satisfy, based on the equations satisfied by regular expressions:

$$\delta_c \text{void} \equiv \text{void} \quad (2.1)$$

$$\delta_c(\text{return } v) \equiv \text{void} \quad (2.2)$$

$$\delta_c(\text{pick } xs) \equiv x \quad \text{if } (c, x) \in xs \quad (2.3)$$

$$\delta_c(\text{pick } xs) \equiv \text{void} \quad \text{if } (c, x) \notin xs$$

$$\delta_c(x \gg= f) \equiv \delta_c(f \ a) \quad \text{if } \nu x = \{a\} \quad (2.4)$$

$$\delta_c(x \gg= f) \equiv \delta_c x \gg= f \quad \text{if } \nu x = \emptyset$$

The derivative of an empty generator, or of one that immediately returns a value without looking
 at any input, should be **void**. The derivative of **pick** depends on whether or not c is present in the
 list of possible choices—if it is, we simply make the choice; if not, the result is **void**. Finally, the
 645 equations for ($\gg=$) are based on the equation for concatenation of regular expressions, using ν to
 check to see if the left hand side of the expression is out of choices to make.

Of course, these equations are not definitions. In fact, the actual definition of the *derivative* for
 a free generator g is much simpler:

```

δ :: Char -> FGen a -> FGen a
δ_c(Return v)          = void
δ_c(Bind (Pick xs) f) =
  case find ((== c) . snd) xs of

```

```

Just (_, _, x) -> x >>= f
Nothing -> void

```

655 Since freer monads are pre-normalized, there is no need to check nullability explicitly in this definition. It is always apparent from the top-level constructor (**Return** or **Bind**) whether or not there is a choice available to be made. The definition is not even recursive!

We can use the earlier equations to give us confidence that this definition is correct.

Lemma 1. δ_c satisfies equations (2.1), (2.2), (2.3), and (2.4). In other words, the free generator
 660 derivative behaves similarly to the regular expression derivative.

Proof sketch. See the full paper for the proofs. Most are immediate. □

Another way to ensure that the derivative operation acts as expected is to see how it behaves in relation to the free generator’s language interpretation. The following theorem makes this concrete:

Theorem 2. The derivative of a free generator’s language is the same as the language of its deriva-
 665 tive. That is, for all free generators g and choices c ,

$$\delta_c^{\mathcal{L}} \llbracket g \rrbracket = \llbracket \delta_c g \rrbracket.$$

Proof sketch. Straightforward induction (see the paper). □

Since derivatives behave as expected, we can use them to simulate the behavior of a free generator. Just as we can check if a regular expression matches a string by taking derivatives with respect to each character in the string, we can simulate a free generator’s parser interpretation by taking
 670 repeated derivatives. Each derivative fixes a particular choice, so a sequence of derivatives fixes a choice sequence.

* * *

The high-level take-away here is that free generators give a way to separate monadic generators from their randomness. This enables useful re-interpretations of standard operations like **pick** and **>=**, for
 675 example as parser operations. The immediate consequence that I explored in *Parsing Randomness* was the availability of new operations like a Brzozowski derivative, but separating generators from

randomness actually turns out to have even deeper consequences. In particular, in the next section I show that generators can be re-interpreted in *two different directions*.

Chapter 3

Reflecting on Random Generation

Progress Draft on arXiv. Needs more thorough evaluation.



Collaborators Samantha Frohlich, Benjamin C. Pierce, Meng Wang

This section contains excerpts from *Reflecting on Random Generation*, which is currently unpublished. The Haskell development was equally divided between myself and Samantha. An early draft of the paper was written with equal contribution, but the latest version (quoted here) was rewritten entirely by me; Benjamin and Meng provided editing and guidance.

In this chapter, I present *reflective generators*, which combine ideas from free generators with ones from the bidirectional programming literature. Reflective generators can be run both forward, to generate test inputs, and backward, to reflect on the *choices* the generator makes when generating a particular input. I show that reflective generators enable a variety of improvements to PBT, including novel approaches to validity-preserving mutation and shrinking.

* * *

The testing literature is full of different techniques for producing random example inputs. For instance, *example-based tuning* [38] changes a generator’s distribution to mimic a set of interesting examples, while *test-case mutation* [45, 28] is used in a variety of testing approaches to explore “the space around” a particular value. But these techniques can be expensive when applied to the complex domains that QuickCheck testers are used to, because they generally involve *rejection sampling*—producing both valid and invalid values and throwing away the invalid ones—which can be costly if valid inputs are rare compared to invalid ones.

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

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

With this intuition in mind, we present a new abstraction for writing generators that “reflect” on the choices made when generating one value to guide the generation of other values. Like
710 *lenses* [12], our *reflective generators* can be run in two directions: forward, as normal generators that make choices and produce values, and also backward, taking values and analysing the choices required to produce those values. The backward and forward modes of reflective generators can be used in tandem to implement both of the generation techniques mentioned above. In both cases, the reflective generator is first run backward to reflect on the choices that produce some old values,
715 then forward to produce new values based on those choices.

After a bit of background (Section 3.1), we offer the following contributions:

- We propose *reflective generators*, a variant of standard QuickCheck generators that can be run both forward, to generate random values, and backward, to reflect on the choices that lead to given values (Section 3.2).
- We apply reflective generators to *example-based tuning*, simplifying a recently published technique [38] and extending its domain from applicative to monadic generators (Section 3.3).
- We apply reflective generators to *test-case mutation*, presenting a novel algorithm for validity-preserving mutation that does not require type-specific heuristics (Section 3.4).

Ultimately, we hope that reflective generators will provide a foundation on which to build a new generation of smarter PBT tools.

3.1 Background

3.1.1 Bigenerators

Generators encode knowledge about what it means to satisfy a validity condition. Our goal with reflective generators is to exploit that knowledge to guide interesting testing strategies. We build on *Composing Bidirectional Programs Monadically* [44], which makes QuickCheck-style generators *bidirectional*: they can run forward as generators, and they can also run backward as *checkers* for the validity condition that the generator direction enforces. This is less powerful than the “reflecting on choices” paradigm that we will present in Section 3.2, but the underlying machinery is the same.

(Fair warning: the abstractions discussed in this section are fairly technical, but they need not be understood in depth to follow the rest of the paper. Readers not familiar with profunctors may find it useful to refer to [44] for further examples.)

Generating and Checking It turns out that, to enable backward “checking”, a generator should be more than just a monad—it needs to be a *profunctor* as well. Profunctors are two-argument functors that are contravariant in their first parameter and covariant in the second. In other words, the second parameter acts like a normal functor, while the first flips the arrows. The following Haskell type-class shows how to map over structures like this:

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

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

Since we do not want to give up the monadic composition used to build up QuickCheck gener-

ators, we require that the profunctors we work with be monads in their second argument. This is expressed via the `Profmonad` class, which has no operations of its own:

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

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

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

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

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

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

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

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

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

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

Programming with Monadic Profunctors Coding with monadic profunctors is like standard monadic programming, but we need to (1) add `dimap` annotations to make the “backward” direction

of the computation explicit and (2) use

780 `internaliseMaybe` when the backward direction might fail. Consider this example, which can be read forward as producing the pair (4, 5) or backward as rejecting all values except that pair:

```

1  profmEx :: (Profmonad p, PartialProfunctor p) => p (Int, Int) (Int, Int)
2  profmEx = do
3      x <- dimap (exact 4) id . internaliseMaybe . dimap fst id $ return 4
785 4      y <- dimap (exact 5) id . internaliseMaybe . dimap fst id $ return 5
5      return (x, y)
6
7  exact :: Int -> Int -> Maybe Int
8  exact m n | m == n = Just m
790 9  exact _ _         = Nothing

```

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

<pre> genBST :: (Int, Int) -> Gen Tree genBST (lo, hi) lo > hi = return Leaf genBST (lo, hi) = frequency [(1, return Leaf), (5, do x <- genInt (lo, hi) l <- genBST (lo, x-1) r <- genBST (x+1, hi) return (Node l x r))] </pre>	<pre> refBST :: forall g. Reflective g => (Int, Int) -> g Tree Tree refBST (lo, hi) lo > hi = return Leaf 'at' _Leaf refBST (lo, hi) = pick [("leaf", return Leaf 'at' _Leaf), ("node", do x <- refInt (lo, hi) 'at' (_Node._2) l <- refBST (lo, x-1) 'at' (_Node._1) r <- refBST (x+1, hi) 'at' (_Node._3) return (Node l x r))] </pre>
---	---

Figure 3.1: Converting a QuickCheck generator to a reflective one.

3.2 Reflective Generators

The bigenerators presented by Xia et al. only say whether or not *there exist* generator choices that result in a given value. What we really want to know is *which* choices! In this section, we describe

800 reflective generators and show how they are able to “reflect” on the choices that produce a particular value and expose them for external use.

3.2.1 The Reflective Abstraction

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

```
type Choice = String

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

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

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

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

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

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

820 Figure 3.1 shows a reflective generator for binary search trees (BSTs) that we will use throughout the paper. The recipe for converting such a QuickCheck generator (shown on the left) into a reflective generator (shown on the right) is fairly straightforward (colors connect each step to the parts of the generator it modifies):

1. Replace `Gen` with `Reflective` and `frequency` with `pick`.
- 825 2. Add a descriptive `tag` at each choice point.

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

3. Add **backward annotations** to guide the behaviour of the backward reflection.

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

This example eschews the verbose `dimap` annotations from the previous section and instead uses a much more convenient `at` combinator that we provide. The `at` combinator uses *lenses* and *prisms*² that can be conveniently derived using Template Haskell³. The `_Leaf` prism simply checks that the value is, in fact, a `Leaf`, and the `_Node` prism extracts arguments from the `Node` constructor. The
835 `_1`, `_2`, and `_3` lenses extract the first, second, and third argument respectively. We also provide combinators that let users write backward annotations with normal functions, rather than lenses, but we find the lens presentation quite natural.

This annotation scheme is not difficult to apply to real-world code. For example, we converted all of the generators in `xmonad`⁴—an industrial-strength, dynamically tiling X11 window manager
840 that is famed for being thoroughly tested in QuickCheck—to reflective generators. The whole exercise took just a few hours, much of it spent understanding the rather complex domain types and testing goals.

3.2.2 Interpretations

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

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

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

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

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

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

²<https://hackage.haskell.org/package/lens>

³<https://hackage.haskell.org/package/template-haskell>

⁴<https://xmonad.org/>

Using polymorphism for interpretation like this gives us what is often called a *tagless-final embedding* [24]; besides labelling choices, this is the main place that our reflective generators diverge from bigenerators. Tagless-final style takes advantage of Haskell type classes in a “classy” embedding
855 that easily supports adding both new interpretations (new type-class instances) and new constructs (new type-class methods).⁵

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

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

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

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

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

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

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

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

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

```
> sample (gen (refBST (-10,10)))
```

```
Leaf
```

```
> sample (gen (refBST (-10,10)))
```

```
880 Node Leaf 4 (Node Leaf 10 Leaf)
```

⁵Note that this is an entirely different interpretation mechanism than the one used by free generators, but the intuition is the same. The salient difference is that free generators admit operations like derivatives, whereas tagless final embeddings use type-class magic to infer interpretations.

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

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

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

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

890 This goes through the listed generators (again ignoring tags with `snd`) and runs each on the `b` value that is being analyzed. As long as one of the generators in `xs` can produce `b`, the result of `pick` will be a `Just`, signalling that the generator can in fact produce the desired value. (Morally, the `Just` value simply means `True`, but we need a `Maybe` here for technical reasons.)

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

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

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

905 3.2.3 Ramping Up

Our abstraction recovers the behaviors of both standard QuickCheck generators and “classic” bi-generators, but this is just the first step. The beauty of reflective generators is that they can go *far beyond* simple test generation and validity checking; other interpretations are just a few type-class instances away. In particular, there is rich information in the `pick` tags that neither of the
910 above interpretations capitalize on. Section 3.3 and Section 3.4 present two relatively sophisticated interpretations; in the rest of this section, we explore some smaller examples to make sure that the mental model is clear.

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

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

```
newtype Enum b a = Enum { run :: [a] }
instance Pick EnumGen where
  pick = Enum . concatMap (run . snd)
```

Interpreting `refBST (-10, 10)` with this type produces a list of all possible BST with node values between -10 and 10. The implementation of `pick` is dead simple: map `run` over the sub-generators and then concatenate the results. We also have proof-of-concept interpretations that use better enumeration monads like `Logic` [26] and `SmallCheck`'s `Serial` [37], which are more efficient with regard to depth and fairness, albeit with restricted control over enumeration order.

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

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

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

```
newtype Prob b a = Prob { run :: (b, Choice -> Weight) -> (a, Rational) }
instance Pick Prob where
  pick xs = Prob $ \(b, w) ->
    let totalWeight = sum [w t | (t, _) <- xs]
        prob = sum
            [ p * q | (t, g) <- xs
```

```

945         , let p = w t / totalWeight
          , (_, q) <- run g (b, w) ]
in (b, prob)

```

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

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

3.3 Example-Based Tuning

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

Luckily, at least in certain domains, there are more automatic tuning methods that are conceptually simple and work quite well—for example, “example-based” tuning. In this paradigm, testers write down a list of examples that they use as a template for generating more inputs to the program under test. One instance of this approach, *Inputs from Hell* [38], recommends writing down a list of “typical” inputs that the application might commonly encounter and using those to teach a generator how to produce further inputs that are either rather similar to or rather different from the given ones.

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

3.3.1 Inputs from Hell, Briefly

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

```
(2 * 3)
975 2 + 2 + 1 * (1) + 2
((3 * 3))
...
```

These generated inputs look similar to the example, in the sense that they make the same choices with the same frequencies. The expressions use numbers (1, 2, and 3), operations (* and +),
980 and parentheses with weights proportional to their frequency in the original example. The full expression language has many more constructors, but the example constrains the generator to a subset of those options (e.g., subtraction and division never appear). Additionally, by *inverting* the constructor frequencies, the *Inputs from Hell* method can tune the generator to produce inputs that are very different from the original example:

```
985 +5 / -5 / 7 - +0 / 6 / 6 - 6 / 8 - 5 - 4
-4 / +7 / 5 - 4 / 7 / 4 - 6 / 0 - 5 - 0
+5 / ++4 / 4 - 8 / 8 - 4 / 8 / 7 - 8 - 9
```

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

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

3.3.2 Tuning Reflective Generators

For consistency with the rest of this paper, we demonstrate the example-based tuning process for
995 reflective generators using binary search trees instead of expressions.

The key to implementing example-based tuning is reflection. Specifically, we want to take an example like `Node Leaf 5 Leaf`, understand what choices would lead the BST generator to produce that tree, and then tune the generator to make further choices with weights derived from those past choices.

1000 Reflecting on [Choice] We want a function that takes a `Tree` and returns a `[Choice]` that represents the choices made when producing a given tree. We can get one by instantiating `refBST` with an interpretation called `ReflectList`:

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

This type makes most sense when aligned; then it is a function that takes a value of type `a` and
1005 attempts to re-create it using the generator. The generator follows the structure of the input and it records its choices in the `[Choice]`. The result is a list of pairs, since a given value might be produced by the generator in many different (including possibly zero) ways.

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

```
reflectList :: (forall g. Reflective g => g a a) -> a -> [[Choice]]  
1010 reflectList x a = snd <$> run x a
```

Calling

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

gives:

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

1015 That is: to build this tree, the generator first chooses to construct a node, then chooses 5 for the value, and then chooses leaves for both the left and right subtrees.

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

```
instance Pick ReflectList where  
1020   pick xs = ReflectList $ \b -> do  
       (t, g) <- xs  
       (x, ts) <- run g b  
       return (x, t : ts)
```

This runs every available choice and, for those that might lead to the given value, records the choice
1025 label in the output list.

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

Weighted Choices Next, suppose we have obtained a big bag of choices by running `reflectList` on a suite of examples. How do we go about generating new values with a similar distribution?

1030 Following the lead of *Inputs from Hell*, we start by aggregating the bag of choices into a map of type `Map Choice Int` that tracks the count of each choice.⁶ Then we use another `Reflective` interpretation to randomly generate new values, given a weighting function:

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

1035 weightedGen :: Ord t =>
    (forall g. Reflective g => g a a) -> Map Choice Int -> QuickCheck.Gen a
    weightedGen g = run g . (!)
```

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

1040 the choices made to produce a suite of examples.

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

```
instance Pick WeightedGen where
    pick xs = WeightedGen $ \w ->
        frequency' [(w t, run x w) | (t, x) <- xs]
1045     where
        frequency' ys
            | all ((== 0) . fst) ys =
                oneof (map snd ys)
            | otherwise =
1050         frequency ys
```

Each choice is made with frequency `w t`, where `w` associates a natural number weight to each tag `t`. This form of weighted choice is similar to QuickCheck's `frequency` combinator, except that the weights are provided externally rather than being built into the generator. (The `frequency'` combinator acts like `frequency`, except when all weights are zero; `frequency` fails in that case, 1055 while `frequency'` simply picks uniformly.)

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

⁶One could imagine remembering the *order* of choices that lead to the given outputs and trying to replicate that too; for simplicity, we follow the lead of [Soremekun et al.](#) and pay attention only to the distribution of individual choices.

```

tunedLike :: (forall g. Reflective g => g a a) -> [a] -> Gen a
1060 tunedLike g =
    weightedGen g .
    Map.fromListWith (+) .
    map (,1) .
    concatMap (head . reflectList g)

```

1065 We can call it like this:

```

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

```

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

This setup does more than just replicate the results of [Soremekun et al.](#): it generalizes them. The
1070 generators in *Inputs from Hell* are written in a quite constrained format—probabilistic context-free grammars—that is much less expressive than our monadic generator language. This works fine for some classes of values, but it cannot express the kinds of dependencies that monadic generators allow. In the case of the expression example, reflective generators make it possible to encode constraints (e.g., expressions must contain division by zero) that are impossible to encode in a context-free way.
1075 And, of course, to generate BSTs, we need the ranges for the subtrees of each node to depend on its label.

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

1080 3.4 Test-Case Mutation

The standard QuickCheck approach to testing is a bit wasteful. Suppose a generator goes to the trouble of producing a particularly interesting value. Maybe that value exercises code paths that most values do not, or maybe it has a particular shape that is interesting. Regardless, QuickCheck will use that value once, then discard it and generate the next input from whole cloth. But what
1085 if, instead, we could *mutate* values that seem interesting? Then we could “squeeze more juice out of them” by exploring other similar values.

This is superficially similar to example-based tuning, in the sense that old values are used to produce new ones, but it is operationally quite different in a couple of ways. First, new mutated

values are expected to be *structurally similar* to the old values; the *Inputs from Hell* approach does not attempt to do this. Second, whereas examples in example-based tuning are provided by the tester, interesting values that warrant mutation here would typically be discovered with a feedback-driven process—e.g., choosing examples that exercise parts of the system under test that have not been explored by previous tests.

Test-case mutation is used by *coverage guided fuzzing (CGF)* and related techniques [45, 18, 28], and it can be extremely effective at exercising a system under test. Unfortunately, the mutation strategies in existing tools do not respect validity conditions; rather, they simply discard mutants that turn out to be invalid. This can be costly if mutations frequently produce invalid values.

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

3.4.1 Value Preserving Mutation

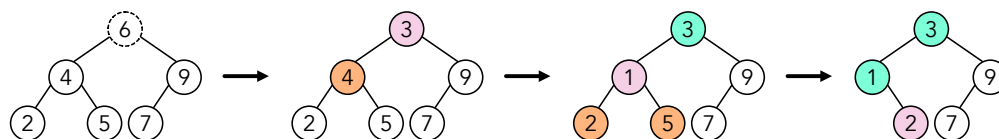


Figure 3.2: A validity-preserving mutation on a BST.

In CGF, the “fuzzer” keeps a pool of “seed” values that it deems interesting and mutates those values to try to find other interesting values that are “similar”. While fuzzers like FuzzChick [28] have been designed with property-based testing (PBT) in mind, incorporating mutations for complex structured data.

To see why this is a problem, consider trying to mutate a BST like the one shown in Figure 3.2. If, for example, we change the root node’s value from 6 to 3, we leave the tree in an invalid state, with a 4 to the left of the 3. If we stopped here, we’d need to throw away this mutant tree and try again.

But we do not have to stop: we can go a bit farther and try to fix up the tree to again satisfy the validity condition. This is the idea of *validity-preserving mutation*: take a value that satisfies a validity condition and turn it into a “nearby” value that also satisfies the condition. For BSTs, the process is illustrated in Figure 3.2: it starts the same way, mutating 6 to 3, then propagates the

validity condition down the tree, updating more values until the invariant is restored.

1115 It might seem like such mutators would be a pain to implement, requiring bespoke code for each type of value and validity condition; surprisingly, they can be derived available automatically for any reflective generator!

3.4.2 Mutating with Reflective Generators

The idea behind mutating with reflective generators is to mutate the *choices* that are made when a value is generated. As with example-based tuning, this requires two coordinated interpretations of 1120 the same reflective generator:

1. The backward interpretation gives us the choices that led to a value.
2. We randomly mutate these choices.
3. We then use the forward interpretation to construct a new value using the mutated choices, being careful to recover gracefully if the mutated choices do not actually correspond to choices 1125 the generator can make.

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

1130 **Refining the Choice Structure** In Section 3.3 we captured choices in a list. This representation turns out to be inadequate here, since it loses some information about where the choices were made. Consider this list of choices and the term it produces:

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

1135 It is hard to see which choices correspond to which subtree. It also makes it hard to preserve the tree’s structure once we start mutating the choices. For example, suppose we randomly change the third element from “node” to “leaf”. The left subtree of the root obviously changes from (Node Leaf 2 Leaf) to Leaf. When we come to generating the right subtree, the next available choice

is "2", which makes no sense in this context, so we discard it and use the next choice, which is

1140 "leaf", to complete the tree:

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

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

1145 We can do better by storing choices in a data type that encodes the branching structure of the generator:

```
data Choices c  
  = None  
  | Mark c (Choices c)  
1150  | Split (Choices c) (Choices c)
```

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

```
Mark "node"  
  (Split (Mark "5" None)  
1155    (Split (Mark "node"  
              (Split (Mark "7" None)  
                    (Split (Mark "leaf" None)  
                          (Mark "leaf" None))))  
    (Mark "node"  
1160    (Split (Mark "2" None)  
          (Split (Mark "leaf" None)  
                (Mark "leaf" None))))))
```

We can change the backward interpretation of the reflective generator from Section 3.3 to use this representation instead of lists of choices:

```
1165 newtype Reflect b a = Reflect { run :: b -> [(a, Choices Choice)] }  
  
reflect :: (forall g. Reflective g => g a a) -> a -> [Choices Choice]  
reflect x a = snd <$> run x a
```

The class instances are almost identical to the ones above for `ReflectList`; we just modify the way
1170 the choice structure is built. We can run `reflect` on our example tree to get the tree structure above.

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

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

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

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

Here are two more useful mutators:

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

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

These mutations are designed to mimic FuzzChick mutations, which themselves mimic the industry standard fuzzing tool AFL [45]. Together, they form a comprehensive picture of the ways one might want to mutate a structured value. Still, we do not claim that these mutations are the best possible ones, and we expect careful thought will need to be put into exactly which mutations are applied when. For now, our goal is simply to demonstrate that a wide variety of mutations are possible, and that those mutations can be designed without specialising to a particular type of value. Any mutator written over trees of choices can be used to mutate any value using a reflective

generator.

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

```
1205 newtype MutGen b a = MutGen { run :: Choices MayReroll -> QuickCheck.Gen a }
```

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

```
1 instance Pick MutGen where
1210 2 pick gs = MutGen $ \case
3     Mark (Keep c) rest ->
4         case find ((== c) . fst) gs of
5             Nothing -> arb gs rest
6             Just (_, g) -> run g rest
1215 7     Mark (Reroll c) rest ->
8         case filter ((/= c) . fst) gs of
9             [] -> arb gs rest
10            gs' -> arb gs' rest
11     _ -> run (snd (head gs)) None
1220 12 where
13     arb gens rest = do
14         g <- elements (snd <$> gens)
15         run g rest
```

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

This greedy choice made by `pick` rests on one strong assumption about the way the reflective generator is written: it assumes that the first choice in any given `pick` is not recursive. This is a somewhat unfortunate requirement, but it greatly improves behaviour of mutation. To see why, consider a mutation that changes a `Leaf` to a `Node` in a BST. If we want to keep the mutation

1235 “small”, we want new subtree to be as small as possible; specifically, we would like to have `Leaf` on both sides of the new `Node`. The greedy choice assumption makes this happen without a fuss. Happily, if the generator terminates at all, it can be rewritten to satisfy this assumption, so this does not pose a significant problem.

To complete the picture, we can write a function that takes a reflective generator and a value
1240 and yields a mutated value:

```
mutate :: Ord t => (forall g. Reflective g => g a a) -> a -> QuickCheck.Gen a
mutate g x =
  elements (reflect g x) >>=
    (manipulate >=> mutGen g)
1245 where
  mutGen = run
  manipulate c =
    oneof
      [ rerollMut c,
1250       (Keep <$>) <$> swapMut c,
        (Keep <$>) <$> shrinkMut c ]
```

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

1255 3.5 Limitations

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

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

```
1260 sized :: (Int -> Gen a) -> Gen a
```

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

Second, the API that we chose for `pick` does not directly support manually weighted generators.
1265 The `WeightedGen` defined in Section 3.3 covers most of the functionality of QuickCheck’s `frequency`

by assigning weights to choice labels, but there is one pattern that `WeightedGen` fails to capture:

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

There is currently no way to choose `n` within the weighting function provided to `weightedGen`,
1270 because the information needed exists only dynamically during generation. We believe there are
workarounds within the framework as we present it, but a more robust solution would be to provide
a slightly less elegant version of `pick` that takes weight information alongside tags. Again, this can
be done fairly easily. bidirectionalization: some generators simply cannot be made bidirectional
for fundamental computational reasons. For example, a generator might do things like compute a
1275 hash that is computationally hard to invert or generate data and throw it away entirely (making it
impossible to replicate those choices backward). This is an unavoidable limitation.

However, many functions that are commonly thought of as “non-invertible” work perfectly well as
reflective generators. Functions with partial inverses are handled automatically by `PartialProfunctor`,
and non-injective functions can be made to work by choosing a canonical inverse. In general, we
1280 expect that only generators with very complex control flow (e.g., ones that require higher-order
functions) and pathological generators like the ones mentioned above will pose problems in prac-
tice. Indeed, we have found that even fairly complex generators (e.g., for well-typed System F
terms) are usually straightforward to bidirectionalize.

* * *

1285 I am quite happy with the current state of the reflective generators work. The applications to
example-based tuning and validity-preserving mutation have thoroughly convinced me that these
ideas are worth pursuing. Still, some reviewers were not convinced. When we submitted these ideas
to ICFP and the Haskell Symposium, reviewers asked for more empirical evaluation.

I think the most direct path towards an impressive evaluation would be to use reflective gen-
1290 erators’ value-preserving mutation to improve testing performance in a realistic setting. This will
require two nontrivial implementation tasks: (1) combining a reflective generator mutator with a
coverage-guided fuzzing algorithm, and (2) finding a good benchmark to demonstrate the power of
reflective generators.

The first implementation task, finding a way to connect my validity-preserving mutation scheme
1295 to a coverage guided fuzzer, could be accomplished in a number of ways. Perhaps most straightforward (but potentially ill-advised) would be to simply mock up a rudimentary system myself. This would be time-consuming, but it has the benefit that I can build exactly what I need (and no more). Another option would be to find a pre-built (or partially built) fuzzer for Haskell and piggy-back off of that implementation. I happen to know of a colleague who has been working on something like
1300 this, but as of now that work is not published, so it is unclear if I would be able to build on that work. Finally, I could abandon Haskell altogether and re-build the reflective generator infrastructure in another language. One compelling option is actually to implement reflective generators in Coq, since its type system is powerful enough to support monadic profunctors and since FuzzChick [28] is a compelling point of comparison. I am open to feedback about exactly which of these paths I
1305 should take.

The other implementation task is finding a good benchmark. Luckily, I am currently collaborating on work (separate from my thesis) that aims to address the current lack of PBT benchmarks. This project brings together a number of benchmarks in both Coq and Haskell that aim to span from standard examples used in papers to more realistic examples from the real world. I will use
1310 the ideas already floating around in this project to help narrow down the kinds of programs that value-preserving mutation might be helpful for testing.

Ultimately, I hope to carry out these implementation efforts—potentially with the help of other students—and have a new version of *Reflecting on Random Generation* ready for the next ICFP.

Chapter 4

Bringing Fuzzing into Focus (*Speculative*)

Progress None yet.

This section is speculative, so ideas and suggestions are especially welcome.

In this chapter, I present a plan to narrow the divide between PBT and fuzzing by unifying the approaches in a single framework. At the core of this idea is a new testing setup that combines an off-the-shelf fuzzer with reflective generators to achieve (hopefully) high-performance fuzzing and PBT.

As this work is speculative, I need all of the feedback I can get. In particular, since this work exists in an (admittedly awkward) gap between fuzzing and PBT, I would appreciate feedback about places where my analysis of existing fuzzing tools is incorrect, or where my assumptions may depart from assumptions in the literature.

A fuzz tester or *fuzzer* is a tool that executes a program over and over with randomized inputs in the hopes of triggering an exception or crash. Fuzzers like AFL [45] have been extremely successful at finding security bugs in a variety of real-world systems. Since fuzzing and PBT are both methods of randomized testing, one might expect that they share a significant amount of literature, but in reality they do not. There are some examples of papers at the intersection of PBT and fuzzing (e.g., [35, 28, 36] to name a few) but often the communities seem to “talk past” one another.

I propose that the main thing separating the PBT and fuzzing communities is simply a difference of *focus*. The fuzzing literature mostly talks about on fast and automatic ways to find critical (security) vulnerabilities in programs. In contrast, PBT researchers want effective ways to test semi-
1335 formal logical specifications of their programs. Both areas of focus are important: fuzzing captures the “80%” of cases catching high-profile bugs with minimal programmer effort, and PBT gives a level of thoroughness that fuzzing does not claim to match. But there is something unsatisfying when things are laid out this way. In particular, while fuzzing and PBT focus on different testing problems, they face many of the same technological hurdles. Both PBT and fuzzing need fast and
1340 effective ways to generate random inputs that are valid for the systems that they are testing, and neither community has truly settled on the “right” way to get there.

The project I have in mind will provide a common ecosystem in which to chip away at the problem of constrained generation to the benefit of both the fuzzing and PBT communities. I will use use reflective generators along with off-the-shelf fuzzers to create a framework of generators
1345 that can be improved gradually and as-needed. I will also provide infrastructure to use this fuzzing scheme to generate inputs to functional properties, ensuring that as fuzzing technology improves so does PBT.

4.1 Proposed Project

In this section, I describe a high-level plan that will bridge some of the gaps between fuzzing and
1350 PBT.

4.1.1 An Aside about Rust

In a departure from the rest of the work in this document, I want to design and develop this tool in Rust (not Haskell). There are a few reasons this makes sense:

- Rust is a good target for fuzzing. Fuzzing tends to be most effective and popular in low-level
1355 languages that manage their own memory and are therefore prone to security vulnerabilities; while safe Rust does indeed defend against many of these potential issues, sometimes unsafe code is necessary for efficiency reasons.
- Rust has a powerful type system and a very powerful macro system. While reflective generators

are theoretically expressible in any programming language, they are more natural to work with in languages that have a type system that can ensure that backward-direction annotations are inserted correctly. I am relatively confident that, with the help of a few macros to get around Rust’s lack of higher-kinded types, Rust’s type system can prevent the same kinds of generator bugs that Haskell’s can.

- Rust is trendy! I don’t think it hurts to work in a language that folks already have their eyes on for other reasons.

I could be convinced to implement this tool in a different language, but these reasons have been enough to convince me that Rust is the right choice.

4.1.2 Part 1: Gradually Constrained Generation

The main conceptual component of this project combines reflective generators with an off-the-shelf fuzzer to enable *gradually constrained* generation. My plan is shown in Figure 4.1. We start with a system under test; just like in standard fuzzing, the goal is to make the system crash by passing in a variety of semi-random inputs. Normally, inputs are fed into the system’s parser, which does the job of checking that the input values are valid. To avoid generating lots of inputs that fail to parse (which would be far too slow), fuzzers often use a grammar (e.g., a CFG) to constrain the inputs that are passed to the parser.

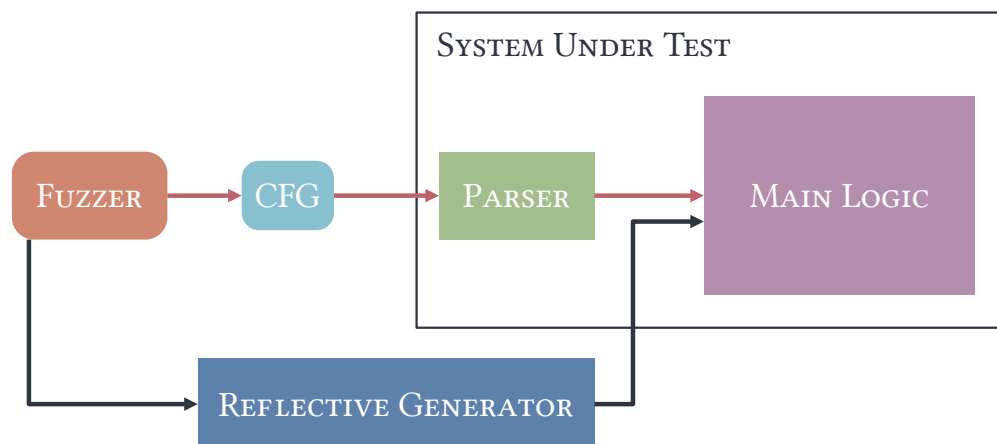


Figure 4.1: An overview of gradually constrained fuzzing.

In our workflow, we skip the parser phase of the system entirely. We generate the data structure

that the parser produces directly, using a reflective generator.¹ However, unlike in standard PBT, we control the generation of the generator using an off-the-shelf fuzzer. The fuzzer generates the choices that the reflective generator makes, biasing it towards choices that may improve code coverage (which comes for free with most fuzzers). These choices are then used to construct the data structure that is used to exercise the main logic of the system under test.

There are a number of features that make this a compelling approach.

Feature: Gradually Enforced Constraints The flagship feature of this approach is that a reflective generator is far more powerful and flexible than a CFG when it comes to constraining the generated inputs. At its simplest, a reflective generator can enforce the same invariants as a simple CFG, ensuring that the input is well-formed (and, since the reflective generator lives in the same language as the system under test, well-typed). But we need not stop there. If the tester finds that the fuzzer still fails to produce useful inputs, the user is free to add constraints to the reflective generator. These constraints can be simple, like narrowing the range of an integer field, or extremely complex, like ensuring that a program is well-typed. The beauty is that the tester can decide exactly how hard to work on the generator, finding the the right balance between effort and testing performance.

Feature: All the Benefits of Fuzzing Of course, if a tester was only concerned with satisfying input constraints, they might just opt to write a standard PBT generator. Another benefit of this approach is that it keeps an off-the-shelf fuzzer in the loop, using its highly-tuned process for coverage-guided mutation to help make choices intelligently. In theory, this may mean that the user can stop optimizing the generator early, allowing coverage information to guide the generator the rest of the way.

One interesting note is that there is flexibility in exactly how we tell the generator to process choices. We could choose to interpret the reflective generator as a strict parser (albeit of a much simpler language) and reject invalid choice sequences. Alternatively, we could interpret the reflective generator in a way that more robust to invalid choice sequences (e.g., using **MutGen** from Section 3.4.1). It's possible that each could be better in different circumstances, so I am eager to

¹The parser itself can be tested in any number of ways, separate from the system as a whole.

experiment to see which makes the most sense.

1405 Feature: Seeding the Fuzzer At this point, one might wonder why we bothered to use a reflective generator (rather than a free generator). One compelling reason is that the backward interpretation of the reflective generator can be used to help seed the fuzzer. Most fuzzers ask for a number of *seeds*, input examples that the fuzzer can start from, in order to ensure that the fuzzer does not spend ages exploring inputs that have no hope of working out. Normally these seeds are
1410 easy enough for the user to write down, since they are simply program inputs, but now that we are asking the fuzzer to generate sequences of choices it becomes much more tedious (and error prone) to produce seeds by hand.

This is one great use for a backward interpretation. The user can write down their seeds (either as values in the program, or as text that can be parsed by the program’s parser), and then the
1415 reflective generator can reflect on the choices that produce those seeds.

Feature: Validity-Preserving Mutation Another potential use of the reflective part of reflective generators is to do validity-preserving mutation in the style of Section 3.4.1. In theory, one could replace or supplement the fuzzer’s mutation schedule with mutations that are obtained by manipulating choice *trees* (rather than choice sequences). This would require some help from the
1420 fuzzer itself, since many fuzzers handle mutation internally. Still, it seems like a worthwhile path to explore.

Limitations There are a couple of limitations with this approach that should be discussed. First and foremost, the system under test must be decomposable into a parser and “main logic” in order for this scheme to work. Hopefully this is not too much to ask—after all, many applications’ parsers
1425 are generated via tools like Yacc [23] so they are separated out by default. Even for applications with a handwritten parser, most would argue that this architecture is the “correct” one to use. Still, this requirement is worth keeping in mind.

Another issue is that, this scheme requires more effort than many fuzzing setups. The effort involved in writing and gradually upgrading a reflective generator should not be ignored. On the
1430 bright side, it is quite likely that we can mitigate part of this problem by automatically inferring the initial reflective generator, thereby requiring much less initial effort from the user. This can be

done in a variety of ways, but the simplest would be to derive the generator from type information using macros.

4.1.3 Part 2: Fuzzing Properties


1435 Bringing PBT ideas to fuzzing, as I propose in Part 1, has the advantage that advancements to PBT can directly improve fuzzing. It would be great if the reverse was true as well! The FuzzChick library in Coq has already shown that using fuzzing techniques to test properties is worthwhile [28], so what would it look like to bring properties into the system that I outlined in Part 1?

The answer is actually very simple: fuzz a binary that checks the property and crashes if it 1440 fails. This can be done relatively easy with macros or build scripts, so the programmer hardly needs to know that there is fuzzing happening at all. Ultimately this part of the project is not groundbreaking, but I think the contribution lies in the unification of the ecosystems.

I hope that juxtaposing the ideas from Parts 1 and 2 gives a picture of a world in which, despite their differences in focus, fuzzing and PBT use and improve the same technologies.

Chapter 5

Some Problems with Properties

Progress Pilot study complete, full study in progress. 

Collaborators Joseph W. Cutler, Adam Stein, Andrew Head, Benjamin C. Pierce

This section contains excerpts from *Some Problems with Properties*, under submission to HATRA'22. The writing and research were done with primary (and roughly equal) contribution from Joseph and myself, with significant help from Andrew and Benjamin. Adam was involved in interviewing and initial analysis of the pilot study.

In this chapter, I observe that while improving testing performance is important, there are likely other things that software developers need from PBT. I discuss a pair of semi-structured interview studies (inspired by work in the Human Computer Interaction) that aim to discover new ways for the research community to add value to PBT.

* * *

We describe the first steps in a research program that aims to find and address challenges facing PBT's adoption. Our immediate focus is a pair of need-finding studies that, when complete, will provide the community with a much more robust model of available opportunities to improve PBT. We offer the following contributions.

- We describe a small ($N = 7$) preliminary study using semi-structured interviews to begin to understand opportunities for PBT in the software industry (Section 5.1). The themes from this study offer an initial model of the challenges that PBT faces.

- We outline plans for a future study with a larger population ($N \approx 30$) and a focus refined by the conceptual model of the preliminary study. The study population will tentatively be drawn from the developers at Jane Street, LLC, a well-respected quantitative trading firm (Section 5.2).

The conclusions from this paper are discussed in Chapter 6.

5.1 Preliminary Study

To begin getting a handle on the challenges facing users of PBT, we designed and ran a preliminary study in the Spring of 2022. The core of the study was a set of short semi-structured interviews with developers who had used PBT in the past. The goal in this small-scale study was not to arrive at statistically significant conclusions; rather, we hoped to clarify some of the issues impeding the use of PBT in practice and sharpen our ideas for a later, larger study.

5.1.1 Study Population

To help obtain a cohesive signal from the interviews, we focused our recruiting primarily on professional developers with experience using Hypothesis, the best-known PBT library for an imperative language (Python) [31]. While all of our participants had used Hypothesis, some also described experiences with other PBT libraries in other languages.

We recruited participants using our professional networks, social media, and by cold-emailing authors of public articles and blog posts about PBT. All told, we recruited and interviewed seven participants; we refer to them below as P1–P7. All identified as male, and all were between the ages of 27 and 42. Six of the seven had advanced degrees, with three holding doctorates. They had between 4 and 28 (median 14) years of experience writing code and between 2 and 15 years doing so professionally. The participants reported actively using PBT tools for between 0.5 and 8 years, total.

5.1.2 Interview Design and Questions

Our interview procedure was designed to feel like a casual conversation, to get the participants comfortable talking about successes and failures using PBT. Each interview was thirty minutes

long, conducted over Zoom. These Zoom calls were recorded, then automatically transcribed using Otter.ai for further analysis. (While the transcription was not perfect, it was accurate enough for our open and axial coding in Section 5.1.3. The quotes in this paper are all manually re-transcribed from the recorded audio.) Three team members participated in each interview, with one doing the
1490 actual interviewing, one taking notes on the participant’s answers, and one taking meta-level notes about the interview itself to improve procedures.

The interviewer followed a loose script comprising three main prompts, each intended to spur about ten minutes of discussion, along with some optional prompts that the interviewer could use if time allowed. For all of the prompts, we strove to avoid leading phrasing while still focusing
1495 participants’ attention enough to extract useful information from their answers.

The three main prompts were:

1. Tell us about your most memorable experience applying PBT.
2. How did you come up with the properties that you tested?
3. Did you need to write custom generators? Or do you have an example of a project where you
1500 did?

Prompt (1) got the participant thinking about a particular experience, preventing general, high-level pontification. Prompt (2) was intended to get the participant talking about one of the main two components of PBT, the properties. We hoped to hear about problems that users had experienced in expressing properties for their code. Finally, prompt (3) asked about the other aspect
1505 of PBT—the random generators for test inputs. Testers write a range of generators, from very simple ones that require only a basic use of the library’s combinators to complex ones that require substantial programming, so we were sure to focus the conversation on generators that enforced complex preconditions.

Finally, we attempted to use snowball sampling to find more study participants. Although we
1510 did not successfully recruit via this method, the ensuing discussions made it clear that PBT was often not widely used or understood in our participants’ companies; we discuss this further below.

5.1.3 Thematic Analysis

We analyzed the interview transcripts following the *thematic analysis* methodology described by Blandford et al. [6]. The analysis began with *open coding* of 3 of the 7 transcripts. Three authors (“coders”) independently coded the transcripts, determining an initial set of descriptive codes and categories of those codes, focusing primarily on the obstacles informants encountered. The codes were refined through discussion among the three coders. Then the remaining four transcripts were independently analyzed by the coders with the refined codes, with each author further revising the codes as needed. A preliminary set of codes was then distilled through additional discussion among the three coders.

A comprehensive axial coding was then performed by one of the authors using the agreed-upon set of codes. Another author reviewed the analysis, providing suggestions and pointing out inconsistencies. The analysis was revised to incorporate this feedback, and then validated by another author once again.

What emerged were five categories of obstacles that developers encountered, relating to the design of properties, the definition of generators, learning PBT, integrating PBT into team workflows, and achieving critical mass in the use of the tools. Below, we detail these challenges, referring to particular participants by pseudonyms P1–7. Quotes from participants are lightly edited for clarity, removing filler words and backtracking. We note that, given the small scale of this preliminary study, some of the obstacles are reflective of singular experiences, and the set of obstacles may not be complete. This highlights the need for expanded studies of the sort we describe in Section 5.2 to confirm and deepen our understanding of the usability of PBT tools.

Challenges Designing Properties

One common challenge for developers was to identify, design, and articulate properties that would meaningfully test their code.

To begin with, developers seemed to have trouble identifying use-cases for PBT that took advantage of the unique benefits of PBT over alternative approaches. While participants did write properties, often times they either under-specified the behavior of the system (for instance, simply testing that a program does not crash), or over-specifying it (for instance, comparing to a behav-

1540 iorally complete *model* of the program under test, which might need to be separately implemented) (P1, P3–5). These participants were not testing properties as is conceptualized in conventional PBT, which either undermined the power of their tests or led to tests that could be brittle or time-consuming to implement.

In other cases, developer simply did not know how to express the notion of correctness of their 1545 program as a property, which we call a failure to “imagine” a property. P1 described this as the problem of “formulating the right property in the first place,” a difficulty they had experienced despite prior successes using PBT according to an over-specified model-based approach. P2 described circumstances in which this challenge to imagine properties might arise: they were testing a complex numerical computation involved in financial risk modeling. While they anticipated that properties 1550 could help them test the validity of the computation better than their existing suite of unit tests, they emphasized that determining such properties was not at all straightforward.

Once a developer can imagine a property, the next challenge is to implement it. Implementation challenges were of several sorts (P4–6). One is common to software design generally—that module boundaries must be present around the functionality to be tested. P6 described this as the challenge 1555 of finding “compose points where you can integrate [PBT] with the system.” Refactoring code to be testable can be tedious at best and prohibitively time-consuming at worst. P6 reported that, for them, often the costs of restructuring code outstripped the benefits, leading them not to use PBT where it might otherwise have been useful.

Challenges Writing Random Data Generators

1560 Developers also faced obstacles writing data generators for their property-based tests. Some developers reported that they lacked support for generating the kinds of distributions of input data that they felt were necessary for their tests.

P7 recounted testing an application where the inputs were network access control lists (ACLs) with many different validity conditions, each given by a different hardware vendor. In this situa- 1565 tion, P7 found that the *rejection sampling* approach to generation—generating arbitrary ACLs and throwing out invalid ones—was too slow to be useful. The standard solution would be to write a generator that enforces preconditions by construction. Indeed, the participant did note that “it might be possible to make the generator smarter.” However, they continued that it “may or may

not be worth it,” due to the complexity of generating access control lists that are valid for *all* of the vendors.

In another expected finding, some participants described difficulties with controlling the distribution over test cases that their generators produce. As P6 recounted, poor test-case distributions can lead to long testing times and missed bugs as the generator repeatedly tries similar inputs and misses importantly different ones.

“ I just relied on the built in [generators]. But...[there were] some tests that were running for quite a long time. Sometimes they were missing some useful use cases, just because the data distribution targeted use cases were missed by the default generators. ”

Besides these technical issues, participants encountered usability problems working with generator tools. P3 struggled to write a generator in Hypothesis’ generator DSL. They explained that the (purely-functional style) abstraction did not match their expectations, given that Python is an imperative language, and they could not “hold it in [their] head very easily.” This highlights the importance of *idiomatic* generator abstractions.

Challenges Integrating PBT into Programmer Workflows

One unanticipated finding was that some participants struggled to incorporate PBT into their team’s development process. One issue that came up was exactly where properties should be checked in the development pipeline. P4 explained that they generally only check their properties in their continuous integration (CI) system, because they are too slow to run locally. However, they also pointed out that, because property-based testing finds bugs at random, running PBT in CI is “a bit like Russian roulette, because sometimes the code breaks while you’re working on something totally unrelated.” Indeed, P1 agreed that they preferred to not have properties in CI for that reason.

Challenges Learning PBT

For some developers, the process of learning PBT proved a challenge in and of itself. While developers learned about PBT concepts from presentation materials, library documentation, and blog posts, they sometimes found these sources inaccessible or otherwise insufficient for learning how to apply PBT in practice. For example, P4 described finding it challenging to understand the circumstances in which to use PBT after having watched a presentation about it, and P1 expressed a desire for

the library documentation to include a comprehensive corpus of examples to use as starting points for their own tests:

“ *The hard part is knowing when you have a property that you can test and how to write that test.*

1600 *What would make it easier? I almost feel like more examples, you know, like... just examples upon examples upon examples, so that you can just read through them and eventually hit one that, you know, hit a couple that click, and then those are clicked in your mind. And if you see that again, you can apply [them]? ”*

Developers also recounted trepidation at the idea of explaining PBT to their organization’s
1605 newcomers, fearing that it represented yet another unfamiliar tool to learn in their already complex stack.

“ *It’s a bit scary, because every time I have to show the code base to a newcomer, there is plenty of complicated things in the code base... and when you talk about all those things, which are complicated or interconnected, you then go on, ‘By the way, there is this Hypothesis thing, which is really a bit more*

1610 *complicated’... so, *laughs*... I don’t know if when I talk about that people are too lost. ”*

The Critical Mass Problem

Finally, some of our participants’ responses underlined a far more basic and somewhat circular impediment to PBT adoption: for PBT to become widely adopted, it must first become popular. This was best illustrated by P5, when asked if they thought that PBT was missing any features
1615 that could help drive adoption.

“ *Well, I think the problem is really more one of popularity, more than a technical problem. If lots of people use property based testing, then more people will be adept at using it. And people will be more willing to write code that might support such a methodology of testing. ”*

P6 also implied a very similar problem when describing how their managers reacted to a property-
1620 based test that found a potentially severe bug. Despite their excitement around the success story, P6’s managers were unwilling to further invest in PBT due to its relative unpopularity.

“ *They realize that [PBT is] very important and very useful, but can they find somebody to write those tests and maintain those tests? That’s another question. ”*

This bootstrapping problem is not specific to PBT, but this is a helpful reminder that solving
1625 both the technical problems above and the as-of-yet hidden ones is necessary, but likely not sufficient, to ensure broad adoption of PBT. To truly see PBT used across the software industry, researchers

need to convince developers, managers, and community leaders, to invest in PBT without guarantees that it will take off.

The preliminary study identified a number of important growth opportunities for PBT. In Chapter 6 we discuss our plans for follow-on work which will aim to capitalize on these opportunities. But before doing so, we must confirm and refine the findings of the preliminary study, as well as identify barriers to adoption that we may have missed: this requires a larger scale study.

5.2 Planned Work: Full-Scale Study

In this section we describe an ongoing study that builds on the preliminary study from Section 5.1.

We plan to partner with Jane Street, LLC to gain deeper insights into the challenges facing PBT in industrial settings and to begin to explore potential solutions.

5.2.1 Research Questions

The research questions for the full-scale study will again address our central question: How can the research community make PBT more valuable for software developers?

The preliminary study highlighted five sorts of challenges that PBT faces: what might be called *property*, *generator*, *workflow*, *learning*, and *social* challenges. We expect that learning challenges are best explored in a classroom setting (see Chapter 6), and that social challenges are too intertwined with challenges of other sorts to study directly. Accordingly, the research questions for the full-scale study will focus on property, generator, and workflow challenges.

RQ1. What support do developers need to help them imagine properties?

RQ2. What kinds of generators do testers need to exercise their properties effectively? Do they have specific precondition and/or distributional requirements?

RQ3. What aspects of the developer workflow around PBT need improvement?

RQ4. What concrete changes could be made to modern PBT systems to improve effectiveness and usability?

RQ1, **RQ2**, and **RQ3** relate to property, generator, and workflow challenges, respectively. **RQ4** more directly asks how existing systems might be improved. This final question will help to keep us focused, reminding us that participants likely have the context and knowledge not only to identify challenges but also to suggest solutions!

1655 5.2.2 Study Population

As in the preliminary study, our primary tool will be interviews with developers. We will tentatively partner with Jane Street, LLC, a large financial technology firm, interviewing around 30 Jane Street employees about their experiences using PBT.

Jane Street has a number of qualities that makes it an ideal place for this study. To start, Jane
1660 Street developers use a variety of testing tools, including PBT. This gives us a place to start when asking questions, since participants will likely have seen PBT before, and it means that will be score for exploring trade-offs between PBT and other forms of testing. Additionally, Jane Street developers famously build almost all of their systems in OCaml, a mostly functional programming language with strong support for static typing and modularity. This unified ecosystem will allow
1665 us to control for a number of potentially confounding factors: all of the developers we talk to will have access to the same PBT tools and the same libraries, language-level programming abstractions, house coding rules, etc. (which might make testing easier or harder).

Naturally, carrying out a study at a single firm has potential drawbacks as well. The most obvious is that our results may not generalize: We cannot guarantee that our findings will apply
1670 outside of the OCaml ecosystem (and other ecosystems like it). However, Jane Street is home to a diverse array of software systems, including trading systems, quantitative algorithms, networked systems, and hardware description code [32]. We hope that the breadth of these programming tasks will mean that the software developers that we talk to will come to the discussion with a wide variety of experiences.

1675 **Stakeholders** Our initial discussions with Jane Street leadership made it clear that there are two distinct populations at Jane Street that we can learn from. The group we had initially planned to talk to was developers who have used PBT in their work at the firm. These can tell us about how PBT helped them, what challenges they faced, and what techniques they use instead of PBT to

check that their code is correct. But we we can also learn a ton by talking to PBT *stakeholders* at Jane Street—i.e., the folks who build and maintain the PBT systems themselves. Since talking with stakeholders may also help us to refine our developer interview script, we will interview them first.

5.2.3 Interview Plan

Each interview will be allotted a one hour slot, to account for a more detailed interview script than the one in the preliminary study. As mentioned above, we will start by interviewing stakeholders. Our prompts will primarily set the stage for our developer interviews and establish background that will help us to interpret our results, but stakeholders may also be able to help us answer **RQ4** (and to a lesser extent other research questions) directly. We will pick the stakeholder’s brains about opportunities to support users of PBT, phrasing our prompts to encourage creative thinking.

After talking to stakeholders, we will begin the developer interviews. The script for these interviews will depend on our conversations with stakeholders, but at a high level we will aim to answer our research questions as directly as possible. Much like the preliminary study, we will have our participants tell us about a specific experience with PBT and ask them about the properties and generators that they used (hopefully giving us insights about **RQ1** and **RQ2**). We will also ask about whether or not they are using PBT on their *current* project (and if not, why not). Finally, we will ask questions addressing **RQ3** and **RQ4** directly.

* * *

I am excited to carry out the large scale study with Jane Street, and to learn more about the experiences real developers have with PBT. Currently the study is under IRB review, but we expect to get started soon. In the mean time, I can spend some time thinking about the themes from the preliminary study and the implications those themes have on potential future projects.

Chapter 6

The Promise of Properties (*Speculative*)

Progress None yet.

This section is speculative, so ideas and suggestions are especially welcome.

1705 In this chapter, I describe a speculative plan for work that follows from the the studies in Chapter 5. Of course, the whole point of the need-finding studies is to determine which future projects will be most valuable, so it is far too early to commit to any one idea. That said, the preliminary study did generate a number of ideas that seem worth pursuing.

6.1 Follow-On Studies

1710 If we are lucky in our conversations with Jane Street developers, we will find as many new questions as we find answers to our original ones. If this is the case, I want to make sure to devote at least some effort to exploring the most interesting of the questions we generate. There are a number of studies and study designs that we may employ to explore these questions, but in this section I describe a couple of options that seem likely to be helpful.

1715 **Surveys for Quantitative Data** One question that the Jane Street study is almost guaranteed to generate is: Do these results generalize? In the previous chapter I give my argument for why I believe that the results *will* generalize, at least to some extent, but empirical evidence supporting that argument may be necessary to convince others.

One relatively straightforward way obtain such evidence is to design a survey that checks the validity of the original findings. The survey would consist of questions that are similar to the prompts we use in the original study, but rather than ask open-ended questions we instead ask questions that are based on the developer answers we received. If we are able to get the survey to enough developers, in diverse enough environments, we could check that other developers agree with what we heard from our Jane Street informants.

Behavioral Studies The data we collect from interviews should reflect developers' *impressions* of their work, but it may miss important details that were simply not memorable. Accordingly, we hope to follow up with one or more *behavioral studies*, where we observe developers completing PBT-related tasks to understand their struggles "in the moment."

Such a study might have a number of different designs. Perhaps the most reasonable approach is to observe developers completing a pre-determined task that we set for them. This would be straightforward to carry out and to recruit for, but our results will be biased based on the task we choose. A more ambitious option would be to carry out contextual inquiries [4], observing real-world users of PBT in their real working environment. We would be limited in our recruiting, since developers often intersperse testing and programming in an unpredictable way, but observing testing in context could provide some deep insights.

6.2 User Centered Design of Developer Tools

Another potential outcome of the full-scale study is an idea for a concrete developer tool. If that is the case, it may be worth spending the time to actually build the tool and evaluate its impact on the practice of PBT.

The preliminary study suggests that one area that new developer tooling may be needed is in managing the interplay between PBT and continuous integration (CI) systems. Recall that developers we spoke with pointed out frustration with the fact that PBT is both nondeterministic and long-running. This combination left them unsure of exactly where and when to test their properties: testing properties locally slowed down their workflow, but testing them in CI occasionally led them to find bugs at inconvenient times.

It is likely that PBT tools could play a role in improving this state of affairs. For example,

one could take inspiration from some theorem provers [3] and create a system in which properties are checked locally but in the background, as the programmer works on other things. This avoids waiting time while potentially being less frustrating than running in CI, since bugs would likely be found while the programmer still had the code “paged in.” Alternatively, one might design a PBT system with CI in mind, providing automated features for deferring property failure notifications until a specified time or turning failing properties into unit tests that can be saved for future testing.

If the full-scale study indicates that this would be a useful line of work, we will refine these ideas via user-centered design. Rather than build a system and hope users like it, we will build minimal prototypes and iterate on the design by observing testers using it. Ultimately we hope this will guide us to a tool that will meaningfully improve the experience of PBT.

6.3 Education

The preliminary study also reminded us that PBT builds on concepts that are not always comfortable for developers, and we expect that we will learn more about the specifics of that discomfort in the large-scale study. Prior work has explored ways to close this knowledge gap [43, 34], but we expect there are further education challenges that are worth exploring.

I hope to work with the course staff of CIS 1210 to incorporate PBT into the curriculum. The ideal scenario would be to add a PBT thread throughout the course, giving students the tools to specify and test their code as they go. I plan to follow the lead of others who have done similar things before (e.g., the PL folks at Brown University) to give students the best chance at incorporating PBT into their tool-set.

There are a few important challenges that need to be considered in order for this to work out. The curriculum is already quite full, so adding PBT likely means removing something else. I will need to work with the professors currently teaching the course to find room, but I expect that this process will be fairly difficult. It is also possible that adding PBT will actually make parts of the course *easier*, especially for students with some knowledge of logic and/or less well-developed unit testing instincts. Honestly I think this is a good thing, as it gives students more ways to succeed and it may re-enforce the value of PBT, but some may find this problematic.

6.4 Solving a Testing Problem at Jane Street

1775 In the course of the large-scale study we may also come across interesting testing scenarios that are worth tackling. Doing so has the potential to teach us about testing problems and solutions that we previously had not been exposed to.

Chapter 7

Conclusion

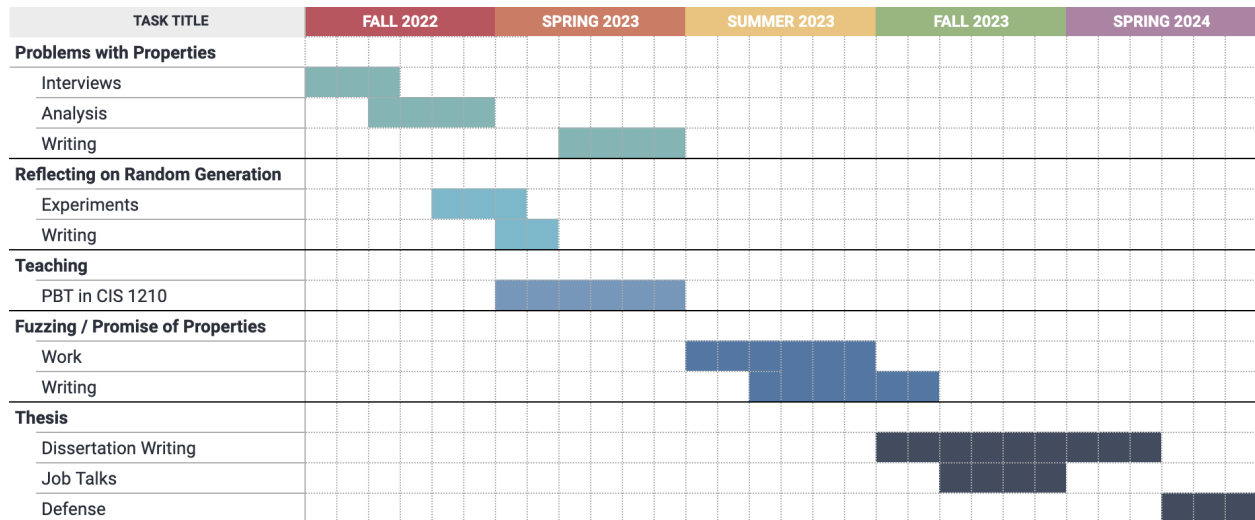
1780 My dissertation seeks to understand and capitalize on major opportunities to grow PBT into a widely used technology. Starting from open problems in the literature, I explore new technologies for random data generators that enable new and powerful generation techniques. Free generators demonstrate a clean way to separate a generator into a source of randomness and a parser, enabling interesting generator transformations and establishing a theoretical basis for other “inter-
1785 pretable” generator abstractions. Reflective generators take this idea a step further, allowing a kind of backward interpretation of generators that enables useful tuning and mutation strategies. These abstractions form the foundation of exciting future work, including an effort that may bring the fuzzing and PBT communities closer together.

Of course, there also are plenty of ways that PBT can be made more valuable that have nothing to
1790 do with generator abstractions. To that end, I am also in the process of carrying out significant user research to learn about the problems that inhibit PBT’s adoption and that limit PBT’s usefulness for software developers. I hope that the interview studies that I am in the process of carrying out lead to useful recommendations for future PBT research, some of which may become part of this thesis.

1795 I believe that property-based testing is worth investing in. It is a intellectually interesting tool that gives everyday programmers the chance to think about their code in terms of abstract specifications, and it is also demonstrably useful for finding bugs in real software. I love programming languages research because we refuse to compromise on beauty in our search for practical tools, and that ethos shines in the beautiful and practical world of property-based testing.

1800 7.1 Timeline

The following Gantt chart outlines a potential timeline for my dissertation:



- **Fall 2022**

- Interviews and analysis for *Some Problems with Properties*.
- Begin experiments for *Reflecting on Random Generation*.

- 1805 • **Spring 2023**

- Finish experiments and writing for *Reflecting on Random Generation*.
- Write the paper for *Some Problems with Properties*.
- Help add PBT to the CIS 1210 curriculum.

- **Summer 2023**

- 1810 – Execute and write up either *Bringing Fuzzing into Focus* or *The Promise of Properties* (if both seem within reach, I may do that and extend the work farther into Fall 2023).

- **Fall 2023**

- Finish up whichever speculative chapter I decide to write.
- Begin writing up dissertation.
- 1815 – Potentially start interviewing for jobs.

- **Spring 2024**

- Finish writing dissertation.
- Defend and graduate.

Bibliography

- 1820 [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] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. 2015. Testing AUTOSAR software with QuickCheck. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 1–4.
- 1825 [3] Stefan Berghofer and Tobias Nipkow. 2004. Random Testing in Isabelle/HOL.. In *SEFM*, Vol. 4. Citeseer, 230–239.
- [4] Hugh Beyer and Karen Holtzblatt. 1999. Contextual design. *interactions* 6, 1 (1999), 32–42.
- [5] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. 2009. Bounded model checking. *Handbook of satisfiability* 185, 99 (2009), 457–481.
- 1830 [6] Ann Blandford, Dominic Furniss, and Stephann Makri. 2016. Qualitative HCI research: Going behind the scenes. *Synthesis lectures on human-centered informatics* 9, 1 (2016), 1–115.
- [7] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *SOSP 2021*. <https://www.amazon.science/publications/using-lightweight-formal-methods-to-validate-a-key-value-storage-node-in-amazon-s3>
- 1835 [8] Rudy Matela Braquehais. 2017. Tools for Discovery, Refinement and Generalization of Func-

tional Properties by Enumerative Testing. (October 2017). <http://etheses.whiterose.ac.uk/19178/>

- [9] Janusz A Brzozowski. 1964. Derivatives of regular expressions. *Journal of the ACM (JACM)* 11, 4 (1964), 481–494.
- [10] 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>
- [11] Kyle Thomas Dewey. 2017. *Automated Black Box Generation of Structured Inputs for Use in Software Testing*. University of California, Santa Barbara.
- [12] John Nathan Foster. 2009. *Bidirectional programming languages*. Ph.D. Dissertation. University of Pennsylvania.
- [13] 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*. 206–215.
- [14] 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.
- [15] 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
- [16] Richard Hamlet. 1994. Random testing. *Encyclopedia of software Engineering* 2 (1994), 971–978.

- [17] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. 445–458.
- [18] Honggfuzz. 2022. Honggfuzz. <https://honggfuzz.dev/>
- [19] John Hughes. 2016. Experiences with QuickCheck: testing the hard stuff and staying sane. In *A List of Successes That Can Change the World*. Springer, 169–186.
- 1870 [20] John Hughes. 2019. How to Specify It! *20th International Symposium on Trends in Functional Programming* (2019).
- [21] John Hughes, Benjamin Pierce, Thomas Arts, and Ulf Norell. 2014. Mysteries of Dropbox. (2014).
- 1875 [22] Daniel Jackson and Craig A Damon. 1996. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on software engineering* 22, 7 (1996), 484–495.
- [23] Stephen C Johnson et al. 1975. *Yacc: Yet another compiler-compiler*. Vol. 32. Bell Laboratories Murray Hill, NJ.
- 1880 [24] Oleg Kiselyov. 2012. Typed tagless final interpreters. In *Generic and Indexed Programming*. Springer, 130–174.
- [25] Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. *ACM SIGPLAN Notices* 50, 12 (2015), 94–105.
- [26] Oleg Kiselyov, Chung-chieh Shan, Daniel P Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). *ACM SIGPLAN Notices* 40, 9 (2005), 192–203.
- 1885 [27] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner’s Luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 114–129. <http://dl.acm.org/citation.cfm?id=3009868>
- 1890

- [28] 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>
- 1895 [29] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. 2017. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.
- [30] Andreas Löschner 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>
- 1900 [31] David R MacIver, Zac Hatfield-Dodds, et al. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891.
- [32] Yaron Minsky. 2022. Signals and Threads. Web. <https://signalsandthreads.com/>
- 1905 [33] Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.
- [34] Tim Nelson, Elijah Rivera, Sam Soucie, Thomas Del Vecchio, John Wrenn, and Shriram Krishnamurthi. 2021. Automated, Targeted Testing of Property-Based Testing Predicates. *arXiv preprint arXiv:2111.10414* (2021).
- 1910 [35] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-Guided Property-Based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) (*ISSTA 2019*). Association for Computing Machinery, New York, NY, USA, 398–401. <https://doi.org/10.1145/3293882.3339002>
- 1915 [36] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2020. Quickly generating diverse valid test inputs with reinforcement learning. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1410–1421. <https://doi.org/10.1145/3377811.3380399>

- [37] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy small-check: 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>
- [38] 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).
- [39] 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*. 244–256.
- [40] Dominic Steinhöfel and Andreas Zeller. 2022. Input Invariants. *ESEC/FSE 2022* (2022).
- [41] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*. Springer, 581–601.
- [42] 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)*. IEEE, 724–735.
- [43] John Wrenn, Tim Nelson, and Shriram Krishnamurthi. 2021. Using Relational Problems to Teach Property-Based Testing. *The art science and engineering of programming* 5, 2 (2021).
- [44] Li-yao Xia, Dominic Orchard, and Meng Wang. 2019. Composing bidirectional programs monadically. In *European Symposium on Programming*. Springer, 147–175.
- [45] Michal Zalewski. 2018. AFL quick start guide. <http://lcamtuf.coredump.cx/afl/QuickStartGuide.txt>.