PROPERTY-BASED TESTING FOR THE PEOPLE

Harrison Goldstein

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2024

Supervisor of Dissertation
Benjamin C. Pierce, Henry Salvatori Professor of Computer and Information Science

Graduate Group Chairperson
Mayur H. Naik, Professor of Computer and Information Science

Dissertation Committee
Stephanie C. Weirich, ENIAC President's Distinguished Professor of Computer and Information Science
Andrew M. Head, Assistant Professor of Computer and Information Science
Mayur H. Naik, Professor of Computer and Information Science
Hila Peleg, Professor of Computer Science, Technion

PROPERTY-BASED TESTING FOR THE PEOPLE

COPYRIGHT

2024

Harrison Joseph Goldstein

ACKNOWLEDGEMENT

It is hard to describe what a privilege it has been to be able to complete this dissertation. Documents like this always look like the effort of a single person—there is one person's name prominently written on the title page and one person who is responsible for getting the words in the right order—but this document would not have been possible without the truly incredible amount of support that I have received throughout my life, and especially over the last five years.

To my academic mentors: Benjamin Pierce, Andrew Head, Adrian Sampson, Leo Lampropoulos, Hila Peleg, and many others. I have relied on you for everything from technical feedback to broad personal advice, and no matter what I needed you have always been honest and supportive. My trajectory has been shaped by your wisdom and insight. Great mentors are hard to come by, and I consider myself incredibly lucky to have so many.

To my academic peers and Ph.D. collaborators: Cassia Torczon, Jessica Shi, Jeff Tao, Joe Cutler, Sam Frohlich, and others. Doing research alone is no fun; my best Ph.D. work only started when I began to collaborate with other students. For those of you who contributed to projects in my main line of work: I am forever grateful for it. And for those of you who allowed me to help on your projects: I hope I did more good than harm. I would love to be able to continue collaborating with all of you.

To my friends: too numerous to list. Every one of you was a critical piece of this process. Thank you to everyone who listened to my complaining; everyone who helped me decompress with dinners and hangouts; everyone who let me stay at their place when I decided to escape for a weekend; everyone who asked how the thesis was going and everyone who figured they'd better not. I have the best friends.

To my family: Helen, Steven, and Noah Goldstein. You have always provided me with the perfect balance of guidance and independence. You give me advice when I ask for it and suggest solutions when I have problems, but you also give me room to be my own person and live my own life. This Ph.D. was certainly not your idea, but once I chose it you were there to support it. You also keep me grounded, with things like crossword puzzle meetings which kept me energized over the last few months of writing when the days blended together. Thank you, and I love you.

As I said, this dissertation is nominally my work—and I did write it—but I am a product of the dozens of people who have supported me throughout my life, so this dissertation is for them as well.

ABSTRACT

PROPERTY-BASED TESTING FOR THE PEOPLE

Harrison Goldstein

Benjamin C. Pierce

Software errors are expensive and dangerous. Best practices around testing help to improve software quality, but not all testing tools are created equal. In recent years, automated testing, which helps to save time and avoid developers' blind-spots, has begun to improve this state of affairs.

In particular, *property-based testing* is a powerful automated testing technique that gives developers some of the power of formal methods without the high cost. PBT is incredibly effective at finding important bugs in real systems, and it has been established as a go-to technique for testing in some localized developer communities.

To bring the power of PBT to a larger demographic, I shift focus to PBT *users*. My work is motivated by conversations with real PBT users, accentuating the benefits that they get from PBT and reducing the drawbacks. My work begins with *Property-Based Testing in Practice*. This in-depth user study establishes a rich set of observations about PBT's use in practice, along with a wide array of ideas for future research that are motivated by the needs of real PBT users. The rest of the work in the dissertation is informed by these results. One critical observation is that PBT users struggle with the random data generation that is a key aspect of PBT's operation. To address this problem, I establish a new foundation for random data generators in *Parsing Randomness* and extend that foundation to be more flexible and powerful in *Reflecting on Randomness*. These projects contribute new algorithms that increase developers' leverage during testing while decreasing developer effort. I also observed that PBT users are not always good at evaluating whether their testing was effective. In *Understanding Randomness*, I establish a new PBT paradigm that gives developers critical insights into their tests' performance and enables new ways of interacting with a PBT system. The TYCHE interface developed in that project is now an open source tool with real-world users.

By blending tools from programming languages, human-computer interaction, and software engineering, my work increases the reach and impact of PBT and builds a foundation for a future with better software assurance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

Introduction

Modern software is full of bugs, and they're *expensive*. A 2002 study [149] estimated that the total cost of software errors was almost $60 billion per year in the United States, and the problem has only gotten worse, with a 2022 study [98] estimating that the cost of poor software quality is now over $2 trillion per year. Software bugs also have a human cost: they erode trust and cause frustration in the best cases, and they lead to serious physical injury [109] in the worst.

Of course, software developers have a wealth of techniques that help them find and fix incorrect code, and chief among them is *testing*. The same 2002 study suggested that better testing infrastructure might help to recoup over a third of the estimated cost of software bugs. But what does better testing infrastructure look like?

At the moment, the most common form of testing is manual, example-based unit testing. Developers validate a program by checking that it conforms to input-output examples that capture desired behaviors. Unfortunately, this kind of testing is both expensive and inconsistent. Each example requires time to develop and express, and the examples are naturally biased towards situations that developers already had in mind, rather than towards blind-spots. In the face of these shortcomings, automated approaches to testing are growing in popularity: fuzz testing, metamorphic testing, model-based testing, and parameterized unit testing are just a few of the many approaches that developers can use to supplement their manually written tests.

In this dissertation, I focus on *property-based testing* [20]. Like many other approaches to automated testing, PBT avoids the limitations of manual testing with the help of randomness: by randomly choosing inputs to the program under test, PBT side-steps both the expense and inconsistency of developer-written examples. *Unlike* other approaches to automated testing, PBT is informed by formal methods [23]; PBT is centered around logical properties that formally capture the way a system under test should behave. This formal grounding makes PBT incredibly powerful, both as a testing technique and as a stepping stone to even stronger assurance [99].

And PBT is not just good in theory. The literature is full of accounts of PBT successes, e.g., finding critical bugs in telecommunications software [4], replicated file [82] and key-value [11] stores, automotive software [5], and other complex systems [80]. PBT libraries are available in most major programming languages, and some now have significant user communities—e.g., Python's Hypothesis framework [113] had an estimated 500K users in 2021 according to a JetBrains survey [85].

Still, there is plenty of room for growth. PBT's following is currently localized to relatively narrow communities of software developers. Half a million Hypothesis users represent only 4% of the total Python user base, whereas the Hypothesis maintainers estimate [30] that the "addressable market" is at least 25%. If PBT is going to pay down the trillions-of-dollar bill that software bugs rack up, it needs to reach a larger audience of developers.

In this dissertation, I argue that we can increase PBT's reach and impact by incorporating ideas and experiences from real-world users. My thesis is that combining techniques from programming languages and human-computer interaction leads to significant, actionable insights for the development of better software engineering tools. I demonstrate this through user-centered research on property-based testing that blends theory and practice to achieve tangible results.

1.1 PROPERTY-BASED TESTING IN PRACTICE

The story begins with some formative research. My collaborators and I had worked on PBT for a while, motivated primarily by theoretical concerns and our own intuition, but we were not sure that that work was going to have *impact*—to improve the PBT experience for real developers. Impact is, of course, never guaranteed, but there are ways to improve one's odds. In particular, the human-computer interaction community has shown time and time again that one can achieve significant impact by (1) understanding how real users interact with software tools and (2) motivating work based on that understanding and the stated needs of the users. Thus, we set out to answer two research questions:

RQ1 What are the characteristics of a successful and mature PBT culture at a software company?

RQ2 What are the opportunities for future work in PBT, motivated by the needs of real developers?

Answering the first question establishes a baseline understanding of realistic PBT use and paints a picture of the current best-case scenario for PBT; answering the second prepares us for future research that has a high chance of impact.

In Chapter 2 of this dissertation, I describe an in-depth user study that proposed answers to these questions. Our team—led by me and including Joseph Cutler, Dani Dickstein, Benjamin Pierce, and Andrew Head—interviewed developers about their experience with PBT, collecting qualitative data that addressed our research questions. We chose to focus on developers at Jane Street, a financial technology company that uses PBT extensively, and interviewed a total of 30 participants for around an hour each. We analyzed this extensive dataset via thematic analysis, extracting representative quotes and overall trends that emerged over the course of the study.

Our findings were split into *observations* that comment on important or surprising aspects of PBT's use and *research opportunities* that set up specific directions for future work. The full set of observations and research opportunities appear in Figure 2.2; here I will summarize a few high-level takeaways that are relevant to the rest of the dissertation.

One important observation is simply that PBT was overwhelmingly seen as a useful tool, especially when applied in certain *high-leverage scenarios*. Participants felt that PBT increased their confidence in their code over and above traditional approaches to unit testing, pointing out that it helped them find bugs in "the cases you didn't think of." But participants did not always choose to use PBT for testing. Instead, they tended to use PBT when it was both easy to apply and particularly effective. These scenarios included "differential" or "model-based" testing, where PBT was used to compare two programs that were supposed to behave the same way, and "round-trip" testing, where PBT helped to determine if one function was the inverse of another. One participant said that PBT was most useful for testing "a really good abstraction with a complicated implementation." By focusing PBT on these high-leverage scenarios, participants often saw significant value for manageable cost.

We also made important observations about *when* PBT is used in the testing process. Participants reported that they used PBT during development as a tool for unit testing, rather than as a tool for integration testing. (They used fuzzers like AFL for random integration testing.) This mode of use puts tight time-bounds on the PBT process: some participants included properties with their example-based unit tests in suites that ran every time they saved their file, which meant that they expected properties to run in less than a second! This observation emphasizes the need for performance in PBT tools, and suggests that slower data generation strategies may not be the best choice for some PBT workflows.

Speaking of data generation, one key research opportunity we found was to improve automation around random data generation. PBT requires lots of high-quality random data to be effective, and that data can be hard to come by. In the study, we saw properties that needed valid XML documents, well-typed programs, and compliant financial exchange messages in order to exercise some complex program under test. Participants did not want to have to figure out how to generate these kinds of data manually, and they asked for better automatic solutions. Participants also wanted better automation around debugging workflows, for example test-case "shrinking."

Finally, we identified an opportunity to improve tools for evaluating PBT effectiveness. While participants worked hard on their property-based tests, over a third reported not thinking carefully about whether those tests were actually effective. This caused problems: at least one developer reported a bug making it into production because their property was set up incorrectly and not testing what they thought it was testing. We posited that improved user interfaces could help with this issue.

These observations and research opportunities (in total 6 of the former and 7 of the latter) are intended to guide PBT research beyond just our research group, but we have made exciting progress in addressing some of the identified challenges. The rest of this dissertation addresses generator and shrinker automation as well as PBT evaluation.

1.2 PARSING RANDOMNESS

In the previous section, I described the richly structured random data that developers in our study sometimes needed for testing their properties. This rich structure poses difficulties. Take valid XML documents, for example: if a developer naïvely generates random strings, they are unlikely to produce anything remotely resembling XML. Even if they took some care to generate XML tags, the generator may produce strings like `<foo></bar>` that do not actually represent valid documents. The core problem is that properties usually have *preconditions* that say what it means for a value to be a valid input to the property, and if the generator does not produce enough valid inputs then testing performance degrades.

In Chapter 3 of this dissertation, I describe foundational work to explore this problem and improve the state of the art for users. Benjamin Pierce and I developed *free generators*, a formalization of random data generators that contributes both a better understanding of what generators are and a new algorithm for improving generator performance.

Our key insight is that generators are closely related to *parsers*. This insight had been understood practically for quite some time, but it had never been formalized in the context of QuickCheck-style "monadic" generators. To see how a generator is like a parser, we need to focus on choices. Consider this simple generator that produces pairs:

```
x <- choose (1, 10)
y <- choose (1, 10)
return (x, y)
```

This generator has two choices to make—one to pick `x` and one to pick `y`—and it randomly makes those choices on-the-fly. But what if the choices were made ahead of time? If we had a list of choices `[Pick 1, Pick 2]`, one could imagine the generator taking those as input and arriving at the result `(1, 2)`. From this perspective, a generator acts like a parser of a string of choices. Chapter 3 makes this intuition formal, including proofs of theorems that connect the behavior of parsers and generators.

This shift in perspective is not just interesting, it is also useful for improving the state of PBT for real developers. When viewed as parsers, generators admit a kind of "derivative" operation that can be used in generation algorithms to alter the generated distribution. In particular, we developed *choice gradient sampling*, an algorithm that guides a naïve generator to produce a higher proportion of inputs that satisfy a property's precondition.

The CGS algorithm addresses some of the concerns from the formative study by giving developers a better on-ramp to PBT. We evaluated the algorithm on a variety of complex preconditions, including ones that ensured an input program was well-typed, and we found that CGS significantly increased the proportion of valid values and improved on the variety of those values that were ultimately generated. This means that if developers want to get started with PBT, but they are not yet ready to write a high-quality generator, CGS can help.

Free generators also build a foundation for follow-on work that addresses both generation and shrinker automation; I describe that next.

1.3 REFLECTING ON RANDOMNESS

When viewing generators as parsers, as we do in Chapter 3, we can make modifications to the random choices between generating and parsing them. Concretely, we can take `[Pick 1, Pick 2]` from above, modify it to be `[Pick 1, Pick 1]`, and then parse the new choices to get a new tuple `(1, 1)`. Existing algorithms [112] take advantage of this idea for things like test-case shrinking: given a failing test input, we can attempt to shrink the randomness that produced that input to obtain a smaller example that still causes the property to fail. Since the counterexamples that PBT finds can be quite large, this is a very important step to make debugging possible. In the previous example, if we consider `(1, 1)` to be "smaller" than `(1, 2)`, then we have effectively shrunk the value by shrinking the randomness.

This approach to shrinking is incredibly flexible: the shrinking algorithm does not need to be reimplemented for each data type, it can be implemented once and for all. And developers in our formative study did discuss that hand-writing shrinkers (still a common practice, especially in libraries that do not implement a more generic approach to shrinking) was a significant distraction. But there is one large issue: there is no way to shrink values that were not recently generated. If an end-user were to submit a bug report containing an input that caused the program to fail, or if an input was generated and stored somewhere (e.g., for regression testing), those inputs could not be shrunk. Since shrinking can be critical for the debugging process, this is a significant downside.

In Chapter 4 of this dissertation, I describe work that I led with collaborators Samantha Frohlich, Meng Wang, and Benjamin Pierce, extending free generators to *reflective generators* that can be used to implement an automatic and fully-general approach to test-case shrinking. The key insight comes from the bidirectional programming literature [40]: reflective generators can be run "backward" to extract information from a generated value.

What does it mean to run a generator backward? The parsing view says that generators are functions from lists of choices `[Pick 1, Pick 2]` to values `(1, 2)`; thus, running the generator backward means taking a value `(1, 2)` and inferring the choices `[Pick 1, Pick 2]`. The details of how this is done, and how it relates to free generators, can be found in Chapter 4—for now, the important part is that it *can* be done, and that it requires only modest developer effort to achieve. With a reflective generator in hand, a developer can take an input from a user's bug report, run the generator backward to obtain a list of choices, shrink those choices, and then run the generator forward again to get a smaller counterexample.

Reflective generators also enable new algorithms for generation. The reflective generators chapter describes an algorithm for "example-based tuning," which allows users to modify an existing generator's distribution to mimic a set of provided example inputs. Our approach generalizes one from the literature [155], allowing it to work on a larger class of generators.

Reflective generators continue to push the boundaries of PBT generator technology, providing more flexibility, utility, and automation to developers. While we have not yet achieved the fully-automated generation for complex constrained data that users hoped for in our study, we have laid a solid theoretical and practical foundation. Free and reflective generators clarify the theoretical properties of PBT generators while providing useful algorithms for generation and shrinking that address real-world users' concerns and make PBT easier to adopt.

1.4 UNDERSTANDING RANDOMNESS

Pivoting away from generator and shrinker advances, I will now address another research opportunity from the formative study: testing evaluation. Recall that many participants reported that they did not carefully evaluate the effectiveness of their tests, leading to missed bugs in a few cases. The problem, again, has to do with generation of complex data—as noted above, generating things like valid XML documents is hard to do with naïve generation strategies, meaning that the test will not get enough good data to test with. But there is actually a subtler problem too: even if the generator produced a reasonable number of XML documents, they may not all be *interesting*. For example, if the generator often produces the empty document or a single node document, `<a></a>`, the test may still miss important bugs.

To address this problem, and motivated by suggestions from study participants, we began designing a user interface that includes visualizations to help developers understand the distribution of data that was used to test their properties. Chapter 5 describes both the user interface itself, called Tyche, and our process for refining the design of the interface with the help of PBT experts. I led the design, development, and evaluation of the tool, but I had critical input along the way from Jeffrey Tao, Zao Hatfield-Dodds, Benjamin Pierce, and Andrew Head.

The final Tyche interface shows developers a rich set of visualizations that help them to understand the effectiveness of their property-based tests. It is built with five design principles in mind:

- *Visual Feedback.* Developers should have visual representations of distributions that they can understand at a glance, rather than command-line output which hides important signals.
- *Workflow Integration.* No matter what the developer's PBT setup looks like, they should be able to fit this tool into their workflow. This means embedding the interface in IDEs but also making it available as a standalone application.
- *Customizability.* Each testing domain is different, and developers have different signals that indicate the quality of the inputs used for testing. The interface should be customizable to match that complexity.
- *Details on Demand.* Developers should be able to move fluidly between high-level summaries of the data and low-level details. The high-level is necessary for making quick decisions, and the low-level encourages curiosity that may lead to a better understanding of the system under test.
- *Standardization.* The interface should be usable with as many PBT frameworks and programming languages as possible—broader adoption means safer software.

The interface is described in detail in Chapter 5 alongside screenshots and workflow diagrams.

While we were confident that the final design of Tyche embodied our design principles and matched the expectations of the experts we spoke with, we were not yet confident that we had chosen the right visualizations to really help developers understand their generators. Thus, we evaluated Tyche in a controlled study, primarily answering the question: Does Tyche help developers to predict the bug-finding power of a given test suite? The study concluded that, for 3 of the 4 tasks we had participants complete, Tyche significantly reduced the number of mistakes users made when ranking generators.

The Tyche interface once again demonstrates the power of marrying PL and HCI techniques to achieve impact in software engineering—in this case, not only was Tyche motivated by human-centered research, but it was also co-designed with users' help and evaluated with real developers. The hope is that designing tools this way results in software that is better tailored to users' needs than one designed in a vacuum.

1.5 RELATED AND FUTURE WORK

Since the chapter topics vary significantly, ranging from practical PBT usage to theoretical PBT abstractions, references to related work are distributed throughout this document. Related work sections in each chapter provide background that is specific to the content of that chapter.

The work in this dissertation establishes a strong foundation for PBT that is widely usable and widely used. But this is only the beginning. We must continue understanding user needs, building powerful theories, and implementing practical tools to build a future where PBT is applied successfully across the software industry. In Chapter 6, I discuss that potential future in some detail; I outline plans for projects that target each step in the PBT process, streamlining them and making them more flexible for realistic use-cases.

I believe that by combining the theoretically grounded techniques with an empathetic understanding of real users, tools for maintaining software correctness can continue to keep pace with the ever accelerating world of software engineering.

CHAPTER 2

Property-Based Testing in Practice

In this chapter, I discuss an in-depth, 30-participant interview study that serves as motivation for the rest of the work in the dissertation. Through this study, we crystallized a picture of how PBT is used effectively in a software development context, and we established recommendations for future work in PBT. The content in this chapter is taken, mostly unchanged, from *Property-Based Testing in Practice* [59]; it was published as a Distinguished Paper at the International Conference on Software Engineering (ICSE) 2024. The paper was written in collaboration with Joseph W. Cutler, Benjamin C. Pierce, and Andrew Head at the University of Pennsylvania, along with Daniel Dickstein at Jane Street. I lead both the study itself and the writing effort, although all authors contributed; the study analysis was done primarily by myself and Prof. Head.

2.1 INTRODUCTION

Property-based testing (PBT) is a powerful tool for evaluating software correctness. The process of PBT starts with a developer deciding on a formal specification that they want their code to satisfy and encoding that specification as an executable *property*. An automated test harness checks the property against their code using hundreds or thousands of random inputs, produced by a *generator*. If this process discovers a counterexample to the property—an input value that causes it to fail—the developer is notified.

The research literature is full of accounts of PBT successes, e.g., in telecommunications software [4], replicated file [82] and key-value [11] stores, automotive software [5], and other complex systems [80]. PBT libraries are available in most major programming languages, and some now have significant user communities—e.g., Python's Hypothesis framework [113] had an estimated 500K users in 2021 according to a JetBrains survey [85]. Still, there is plenty of room for growth. Half a million Hypothesis users represent only 4% of the total Python user base, whereas the Hypothesis maintainers estimate [30] that the "addressable market" is at least 25%. (For comparison, the most popular testing framework, pytest, has 50% market share.)

To help move PBT toward wider adoption, the research community (ourselves included) needs to better understand the practical strengths and weaknesses of PBT and the places where further technical advances are required. Existing work in the software engineering literature has studied how other bug-finding tools are used in practice (see §2.6), but PBT offers a unique set of tools and warrants its own investigation. Accordingly, we interviewed PBT users at Jane Street, a financial technology firm that makes significant use of PBT, to learn how they use PBT, where they see its value, and in what ways they think it might be improved. Concretely, we aimed to answer two main questions:

RQ1: What are the characteristics of a successful and mature PBT culture at a software company?

RQ2: Are there opportunities for future work in the PBT space that are motivated by the needs of real developers?

The first question aims both to offer guidance for engineers and managers considering whether PBT might fit well in their organizations and to provide a basis for evaluating and comparing PBT technologies. The second question aims to help shape further research to maximize the impact of PBT.

Our findings contribute a wide range of observations about developers' experiences with PBT, adding nuance to the research community's understanding of PBT's real-world usage. Through our interviews, we gleaned several new insights about the situations in which property-based tests are deployed in practice. We found that developers use PBT mainly for testing components of complex systems, expecting the tests to provide greater confidence than conventional example-based unit tests yet still run quickly as part of their normal test suite. Interestingly, we also found that developers leverage PBT for the secondary benefit of communicating specifications: properties serve as a form of persistent documentation, demonstrating the semantics of the software to readers—a benefit less commonly discussed in the literature. Finally, we found that at Jane Street, PBT is primarily used in "high-leverage" scenarios, where properties are especially easy to identify and test. Beyond deepening our understanding of when and why developers reach for PBT, we also found that PBT technology can be improved in several ways to better support developers. In particular, study participants reported struggling to generate distributions of test examples that they were convinced effectively exercised the property, and sometimes viewed the process of designing random data generators as a distraction. They also lamented the lack of visible feedback on the effectiveness of their testing.

From these findings, we extract a list of opportunities for future research, including understanding the nuances of PBT performance requirements, exploring better support for differential testing, and expanding the high-leverage scenarios in which PBT is most effective. We also highlight opportunities around improving languages for random data generators, designing interfaces for test-case reduction ("shrinking"), evaluating testing success, and using developer feedback to improve the testing process.

We begin with background on PBT (§2.2), then present our study methodology and discuss of potential threats to validity (§2.3). We present the study's main results (§2.4) and detail lessons learned in the form of observations (§2.5.1) and research opportunities (§2.5.2) before discussing related work (§2.6) and concluding (§2.7).

2.2 BACKGROUND

Properties are executable specifications of programs. For example, suppose a developer is working on a new implementation of binary search trees (BSTs)—tree structures where each internal node is labeled with a data value that is greater than any labels in its left subtree and less than any in its right subtree. They know that all the operations on BSTs (insert, delete, etc.) must preserve this validity condition: given a valid BST, they should always produce a valid BST. To enforce this condition, they might write the following test for the `insert` function using OCaml's Core QuickCheck library [37]:

```
let test_insert_maintains_bst () =
  QuickCheck.test
    (both int_gen (filter valid_bst tree_gen))
    (fun (x, t) -> valid_bst (insert t x))
```

The second argument to `QuickCheck.test` (on the last line) is the property: It says, given an integer `x` and a BST `t`, that `insert t x` should return a BST. The first argument to `QuickCheck.test` is a random data generator, which here generates a pair of `both` an `int` and an arbitrary BST (obtained by generating an arbitrary binary tree using `tree_gen` and using `filter` to discard trees that are not valid BSTs). `QuickCheck.test` randomly generates hundreds or thousands of pairs `(x, t)` and invoke the property to check each pair. If this check ever fails, we have discovered a bug in `insert`.

Generators like `tree_gen` are usually written using an embedded domain specific language (eDSL) provided by the PBT framework, which includes base generators like `int_gen` and combinators like `both` that can be used to create generators for complex data types from generators for their parts. The generator language is embedded in OCaml (in this case), giving generator writers access to the full power of the host language. Generators can require some careful tuning to find bugs effectively. This is often because properties have *preconditions* that define what it means for an input to be valid. For example, "`filter valid_bst tree_gen`" is a correct generator of BSTs: it simply discards trees until `tree_gen` happens to generate a valid BST. However, this means that the *majority* of the generated trees will be discarded, wasting precious generation time. A better strategy is to hand-craft a generator that only produces valid BSTs, but such generators can be nontrivial to write.

If the property ever fails during testing, the failing value is presented to the user. This value might be overly complex, with parts that are irrelevant to the failure of the property, so most PBT frameworks provide tools for test-case reduction [112], usually called *shrinking* [79] in the PBT literature.

The BST example above uses OCaml's QuickCheck library, but the general approach—a module under test, a concise property, a data generator, and a shrinker—is shared by all PBT tools, from the original Haskell QuickCheck [20] to Python's Hypothesis library [113] and beyond. The power of PBT comes from the ability to test a huge number of system inputs with a single specification and generator, often uncovering edge and corner cases that the developer might not have considered. It thus hits a useful midpoint between example-based unit tests and heavier-weight formal methods, retaining the precision of traditional formal specifications and their ability to characterize a system's behavior on all possible inputs, but offering quick, best-effort validation instead of requiring developers to write formal proofs.

Of course, PBT is only one among many testing methodologies. Two primary alternatives are especially relevant to our study.

Example-based unit testing [26] (as supported by pytest, JUnit, and other frameworks) evaluates a program by testing it on individual example inputs. For each input, the developer writes down a short snippet of code that checks that the program's output is correct for this input. (We call this style "example-based unit testing" throughout the paper, rather than just "unit testing," because PBT is also typically used to test individual units within larger systems.)

Fuzz testing, invented by Miller [7; 121] and popularized by tools like AFL [180], also aims to find bugs by randomly generating program inputs. The technical foundations of fuzz testing and PBT overlap to a large (and increasing [159; 104]!) degree, but existing tools from the two communities tend to be tuned for different situations: fuzz testing is typically used in integration testing of complete systems, to detect catastrophic bugs like crash failures and memory unsafety, while PBT is used to flush out logical errors in smaller software modules.

2.3 METHODOLOGY

To address the research questions described in the introduction, we conducted an interview study of developers who use PBT in their work. This study was conducted in accordance with the ACM SIGSOFT Empirical Standards for qualitative surveys.

2.3.1 *Population*

As the setting for our study, we chose Jane Street, a financial technology firm that uses PBT extensively. We recruited 31 participants and carried out 30 interviews (one was a joint interview). Participants were recruited by a Jane Street developer who volunteered to coordinate the study: they found instances of PBT in the source tree and contacted the authors of those libraries; announced the study on an internal blog; and carried out snowball sampling by asking participants for others we should talk to. All participants had some experience with PBT, and most self-reported as having positive experiences. We also explicitly asked for participants who reported neutral or negative feelings about PBT, but they were difficult to find: most felt they did not have enough experience to speak to those feelings. Two self-described PBT-critical developers participated in the study. More details about the study participants can be found in Table 2.1.

Most of those we interviewed were *testers* (26/30), who use PBT tools in their day-to-day work, but we also spoke to several *maintainers* (4/30) who play a role in building and maintaining Jane Street's PBT infrastructure. These two groups were given different prompts (see below), reflecting our expectation that the maintainers would be able to offer a more "global" perspective, but their responses were coded uniformly since they address the same research questions.

Participants who answered an optional background questionnaire had between 1 and 26 years of professional Software Engineering experience (median 7) and between 1 and 20 years using PBT (median 6). This wide range of experience (for PBT, almost as wide as possible, since the first paper on PBT is only 23

Table 2.1: Participant backgrounds. Columns 3–5 reflect optional questions; participants that didn't respond to these are listed at the bottom. *Measured in years. **Stated comfort with PBT on a Likert scale: 7 most comfortable, 1 least. †Participant indicated skepticism of PBT. ‡Pair interview; coded as one participant. (At Jane Street's request, we do not associate individuals with their teams or company divisions.)

ID	Role	SE Exp.*	PBT Exp.*	Comfort**
1	Tester	12	3	(6)
3	Maint.	22	17	(7)
4	Maint.	15	16	(6)
5	Tester	5	7	(6)
6	Tester	16	16	(7)
9	Tester	4	4	(5)
11	Maint.	10	7	(6)
14	Tester	8	7	(5)
15	Tester	2	2	(6)
16	Tester	8	8	(3)
18	Tester	1	1	(5)
20†	Tester	5	2	(5)
23	Tester	6	6	(5)
25‡	Testers	26	20	(5)
26	Tester	2	2	(6)
28	Tester	7	1	(5)
29	Tester	4	5	(6)
Other Testers: 2, 7, 8, 10, 12, 13, 17, 19†, 21, 22, 24, 27, 30				

years old! [20]) also means we heard from developers at many points in their careers and with differing levels of software development experience. When asked to rate their comfort with PBT on a Likert scale, these participants were overall quite comfortable with PBT (median 6 out of 7): all but one were at least "somewhat comfortable" (5 out of 7). Working with developers who are already relatively fluent users of PBT allowed us to benefit from their well-informed ideas about how to make PBT better, as well as their frustrations and challenges.

Jane Street's overall financial operations are supported by a diverse set of software engineering efforts. We spoke to developers working on core data structures, distributed systems, compilers, graphical interfaces, statistical computation, and even custom hardware. Study participants spanned fifteen teams in four main areas: Trading (2/30, across 2 teams), working directly with traders to develop technology specific to individual trading desks; Trading Infrastructure (13/30, across 5 teams), building platforms to support Jane Street's trading; Quantitative Research (5/30, across 2 teams), writing software to enable research on trading algorithms; and Developer Infrastructure (11/30, across 6 teams), designing tools and languages

1. Tell us about a noteworthy time that you applied PBT.
   (a) What kinds of properties did you test?
   (b) How did you generate test inputs?
   (c) How did you evaluate the effectiveness of your testing?
   (d) What did you do to shrink your failing inputs?
2. Which parts of the PBT process are the most difficult?
3. What role does PBT play in your development workflow?
4. To whom would you recommend PBT?
5. In what contexts is PBT most useful?
6. Is there anything that would make PBT more useful to you?

(a) Prompts for testers.

1. Have you seen the type of adoption that you want from your PBT tools?
2. How can QuickCheck be improved?
3. What do you think it would take to get everyone at Jane Street using PBT? Would that be a good thing?
4. What do you hope we'll learn from this study?

(b) Prompts for maintainers.

Figure 2.1: Interview prompts.

upon which Jane Street's applications are built. Jane Street developers primarily work in OCaml, a programming language with strong typing, good mechanisms for modularity, and a focus on performance. OCaml is thought of as a functional language, but it has strong support for imperative programming as well.

2.3.2 *Protocol*

We conducted a semi-structured one-hour interview with each participant; Goldstein and Cutler led the interviews. The prompts for testers is shown in Figure 2.1a. The script was designed to attain depth by encouraging reflection on real, memorable experiences with PBT. It evolved somewhat over the course of the study, allowing us to validate interesting or unexpected observations from earlier interviews. A separate script was used with maintainers, shown in Figure 2.1b. Time permitting, we also asked maintainers about their use of PBT.

We did not explicitly interview until saturation; we simply tried to recruit a reasonably sized group that was representative of PBT users at the company. However, as we neared the end of the study we felt we were no longer learning about new aspects or perspectives on PBT, and we saw convergence on many of our findings (as evidenced by the numbers we report in §2.4). Thus, we do not expect there are significant holes in our results.

Once the interviews were complete, they were transcribed using an automated transcription service [133] and analyzed to extract important themes following a thematic analysis process [10]. Goldstein and Cutler carried out an open coding pass (reading through the transcripts and assigning thematic codes as they went). The codes we chose focused on participant goals, benefits of PBT, challenges encountered, and opportunities for improvement. Once the set of codes stabilized, they were validated by a third co-author, then clarified by the whole team to obtain a final set of clean codes. Finally, one co-author performed an axial coding pass (reading through the transcripts and codes again to make connections and ensure consistency) with the updated codebook. Our codebook is available online [52].

2.3.3 *Threats to Validity*

While our study covered a wide variety of software teams and tasks, it should be noted that participants mostly described using a single PBT toolset, in a single language and software development ecosystem. Our findings may under-represent the usage patterns and challenges experienced by those working with other PBT toolsets in other languages. To help us develop findings that generalize beyond the Jane Street toolset, we designed our interview protocol to focus considerable attention on the conceptual and methodological aspects of PBT, rather than on details of OCaml's QuickCheck implementation or specifics of how it is used at Jane Street. (In the cases where we make observations that are toolset-specific, e.g., where we know tools outside of OCaml address some of the problems we observed, we state this explicitly.) In addition, participants were mostly experienced developers who were comfortable with PBT. This means that our study under-reports the experiences of novice developers. Finally, as discussed above, we did not measure saturation as interviews progressed, so there is a chance that more interviews may still have uncovered new insights.

Another threat comes from the inherent limitations of interview studies. For one thing, participants may not always accurately recall the particulars of their experiences with tools. We did our best to mitigate this threat by asking developers to tell us about specific experiences—a standard technique for studies like ours. Additionally, our own biases as interviewers may naturally have skewed the results; Goldstein and Cutler, who carried out the interviews, and Pierce and Head, who also participated with additional questions, are all property-based testing researchers who have a vested interest in the outcome of the study. Outcomes that show PBT in a positive light may have unintentionally been highlighted, and criticisms that we have addressed in our prior work may have received extra attention.

2.4 RESULTS

We now present our findings. We start by describing the benefits that PBT delivered to developers and how it fit with the other testing approaches they used. Then we describe in detail their experiences writing specifications, designing generators, debugging, and evaluating testing success, focusing both on challenges they experienced and on opportunities for improving PBT tools and workflows.

2.4.1 *Benefits of PBT*

As might be expected, the primary reason participants used PBT was to assess whether their code was correct. Participants often used PBT to validate widely used or mission-critical code. For example, one participant described using PBT for code that was "used in 50,000 places across the code base" to establish reliable behavioral contracts for library consumers (P13). Another participant used PBT to check software that interacted with information they were "sending to the [stock] exchange" (P25), which, should it contain errors, could incur serious financial costs. An advantage of PBT in these cases was its ability to explore edge cases (mentioned by 10/30) since, in the words of one participant, "the bugs are always going to happen in cases you didn't think of" (P21).

For many participants, PBT served to increase confidence that their software was robust. They described code tested with PBT as "solid" (P6, P28), and some (9/30) explicitly mentioned that PBT increased their "confidence" that the code was correct. One described using PBT when they "wanted peace of mind" (P11). Confidence was accompanied by material evidence of success: a third of participants (10/30) said their property-based tests found bugs that they had not found via other methods.

A less obvious benefit of PBT was its ability to help participants better understand their work. One participant described properties as tools that "force [the developer] to think clearly" (P30) about their code, and another pointed out that sometimes properties fail because the specification (rather than the code) is wrong (P10). In such cases, failing tests can highlight gaps in the developer's understanding of the problem space that could lead to issues later.

Outside the testing loop, properties were useful for documentation and communication (12/30). One participant noted that properties "are part of the interface and part of the documentation" (P3) and emphasized that, unlike other documentation, properties never fall out of date: whereas a textual description in a comment might drift from the code's actual behavior, a property is repeatedly re-checked as the code changes. Many also talked about the value of PBT in the code review process (18/30) as a compact way to express what a particular program does. One remarked: "I feel like [PBT] is very nice for review…I really dread seeing 10,000 lines of tests where you just need to spend hours to read code and understand what's going on…And I think, if there is a QuickCheck test, and you look at the property that's been tested, which is usually much shorter…it gives you much more confidence that the code is correct." (P19)

The benefits of PBT were such that even those who were critical of it found it to be beneficial in some situations. During the recruiting process we tried to explicitly recruit participants who said they *did not* like PBT; we found two (P19 and P20), but the criticism in their interviews turned out to be constructive suggestions that were echoed by PBT's proponents as well, and both primarily spoke about situations where PBT was valuable.

Moreover, some participants who liked PBT *really* liked it: one said that they "use QuickCheck for everything" (P13) and another asserted that "everyone should be aware of it" (P27). Yet another participant argued that "[PBT] is so useful that it's worth trying to reorganize your code into the form that it is applicable. PBT is so helpful that if you can reinvent things that way, you should." (P10) Several participants (8/30) also talked about evangelizing PBT, encouraging others to use it and increasing its adoption.

We found that this sort of evangelism benefited all parties: PBT becomes more useful as more people on a team or in an organization use it. A culture of supporting PBT can save work: P24 and P27 both noted that PBT would be easier to use if more developers in the organization provided generators for the types exported by their modules, making it much easier to test code in other modules that uses those types. In addition, treating property-based tests as documentation, as many (12/30) did, requires that others in the organization can read and understand such documentation.

2.4.2 *Comparisons to Other Testing Approaches*

The most common approach to testing at Jane Street is not PBT, but rather an example-based unit testing framework called "expect tests" [123]. Expect tests are basically conventional example-based unit tests with editor integration that helps developers create unit tests from the outputs that the system produces. With expect tests, developers add code to the system under test that that logs interesting data to the standard output channel; the expect-testing framework then checks that the output matches the output string that was captured from the previous run of the system—it is up to the developer to decide which output is intended. Expect tests are integrated into Jane Street's editor workflow, and they are used pervasively in much the same way as frameworks like pytest and JUnit are used in other languages. Thus, we can use comparisons with expect tests as a proxy for comparisons with example-based tests in general where the specifics of expect tests are not relevant.

From a legibility perspective, PBT was sometimes seen as better, sometimes worse than expect tests. Expect tests were described as more transparent and "often easier to understand" (P5). P18 said "expect tests…explain what they're doing to a better degree. And it's easier for someone to come in and review my test," and P27 made a similar point. But expect tests can sometimes go past from "transparent" to just verbose. P17 complained that reading a whole output string was onerous, and another participant complained "[with expect tests] it's easy to get into a mode where you just like write a test, and you just print out a bunch of crap. And then it's really annoying to like code review that test because there's just too much stuff" (P7). Properties are more concise.

PBT and expect tests also present different strengths and weaknesses when it comes to test writing. One participant said that writing an individual expect test is "way easier than writing invariants," but they highlighted that at the scale of a whole test suite they "love not writing specific unit tests cases" (P13).

Ultimately there was no universal preference for properties or expect tests—both should be available in a developer's toolkit. We might infer from participant responses that when code is simple enough and examples communicate its behavior well enough, expect tests are an attractively lightweight option. In cases where the code is particularly difficult to get right, or where writing out enough examples becomes tedious, it seems PBT may be a better choice.

One area where properties seem to be at a clear advantage over expect tests is in the confidence they provide developers. P25 remarked of Jane Street developers that "[they] go to randomized testing when [they're] not so confident of what [they've] done." As discussed above, around a third of participants talked directly about PBT building confidence, and the same proportion reported PBT finding bugs that had not been found with other methods.

Expect tests and PBT were both used to test individual software units during development. PBT was almost always described as running in Jane Street's build system, and many developers actually test their properties "locally after each edit" (P2) Participants described strict time budgets for PBT—no more than "30 seconds" (P27) and as low as "50 milliseconds" (P11)—to ensure it would not slow down the build. These time budgets serve to keep PBT from triggering the 1-minute-per-library timeout that many Jane Street developers set for their unit test suite (P7).

Alongside these fast-turnaround unit testing tools, developers also used fuzzing tools like AFL [180] and AFL++ [39] for integration testing. A third of participants (10/30) mentioned fuzzing, and a few (3/30) described using AFL alongside PBT as part of their testing process. In contrast to the strict timeouts set for expect and property-based tests, one participant described a fuzzer that was forgotten and left running for a year and a half on a spare server (P9). Others did not run fuzzing this long, but still considered it to be running "out of band" (P28), outside the normal continuous integration and at time scales on the order of hours or days. The upshot is that fuzzing tools are given *much* longer to run than PBT tools.

2.4.3 *Writing Specifications*

The previous sections have dealt with PBT at a high level; we now begin a deeper dive into the specifics of the PBT process and how developers described actually using PBT tools. The first step in this process is defining one or more properties to test. While participants did describe struggling with this stage, many ultimately developed powerful strategies for finding properties.

Many participants (16/30) said the process of writing specifications slowed their progress. P4 summed it up like this: "I think the most common failure mode is actually not knowing what properties to test." Challenges ranged from systems that seemed not to have properties at all—P1 talked about "a server that serves queries…and has some…nuanced behavior" that is not compatible with "logical properties"—to systems whose properties were hard to articulate in the form of QuickCheck tests: “With [example-based unit tests], I kind of look at it, I can just make a snap judgment as to whether this is okay or not. Trying to formalize that judgment sometimes can be very difficult.” (P2)

Challenges in articulating properties can come from the way code is written. P7 explained that mutable state (e.g., a hidden variable that may change between calls to the same function) gets in the way of writing properties: “there’s…this hidden state component…that kind of makes it harder for me to think about what are the right laws.” P22 also pointed out that “integrating the outside world and…dealing with the interaction of very large and complicated systems” makes it less clear how to “meaningfully test with PBT.” (These findings are not surprising. It is well known that code that interacts with its environment is also a challenge for testing in general, not just PBT: *stateful code* uses mutable data structures such as hash tables, which might introduce differences between runs of the same test if the state is not reset properly; and *effectful code* might rely on external files, databases, or networks, which may not be safe to access over and over during testing.)

Despite these difficulties, the developers we spoke to were generally enthusiastic about PBT. One might therefore wonder: Did they simply forge ahead regardless, or did they have specific techniques for circumventing the difficulties of writing specifications?

What we found was that developers often succeeded in applying PBT by being opportunistic in their choice of *when* to apply it. Rather than try to apply PBT to every program at all times, participants looked for situations where it offered *high leverage*: more confidence for less work. Some participants described this in general terms as “go[ing] for the low-hanging fruit” (P2) or finding places where “the properties are sort of obvious” (P28), but many enumerated specific situations where they would reach for PBT.

Classical Properties (11/30). Certain forms of properties are familiar from the PBT literature and from library documentation for PBT frameworks. These include mathematical properties (e.g., the commutativity of addition), properties drawn from CS theory (e.g., repeated sorting has no effect), and properties that naturally fall out of a data structure’s invariants (e.g., the ordering condition of a BST). These properties may be more accessible because they are naturally top-of-mind.

Round-Trip Properties (11/30) are also common in the literature; we heard about them so much more often than the other classical cases that they seem worth calling out on their own. These properties check that a pair of functions are inverses of one another—for example, parsing and pretty-printing or encoding and decoding functions. This situation is easy to notice and easy to test since the properties are incredibly succinct (you call one function, then its inverse, and then check that the final result matches the original input), so round-trip properties are a popular choice.

Catastrophic Failure Properties (7/30). Rather than write logical specifications, some participants used PBT to try to provoke catastrophic failures such as assertion failures and uncaught exceptions. In general, these kinds of properties provide less confidence in code quality (there can still be logical errors, even if the code does not crash), but they help to rule out worst-case scenarios, and they are easy to write. (These kinds of specifications are identical to the ones often used for fuzzing; the line between the techniques is blurry, but since the participants were using PBT tools—including complex generators—and testing small units of software, we still consider this relevant for our purposes.)

Differential Properties (17/30). A "differential property" (in sense of differential testing [63]) compares the system under test to a reference implementation of the same functionality that serves as a specification; these are often also called model-based properties (c.f. model-based testing [165]), especially when the reference implementation is designed to be an abstract model of the original code. Differential properties were by far the most widely implemented kind of property. Differential properties were described as a "natural place to use property based testing" (P3)—indeed, one participant remarked that PBT was challenging in a particular situation in part because a good reference implementation was not available (P1).

The common theme of the high-leverage scenarios described above is the availability of a succinct abstraction that can be used in writing a property. These PBT scenarios were summed up by P9 with the following mantra: "[PBT is] most useful when. . . you have a really good abstraction with a complicated implementation." We discuss how these high-leverage scenarios should be taken into account to accelerate research in PBT in §2.5.

2.4.4 *Generating Test Data*

Participants had several goals for generating inputs. First and foremost, many participants (17/30) talked about needing to generate values that satisfy some precondition. This can be extremely important for effective testing: if too few of the generated values satisfy the property's precondition, then testing will fail to exercise the code in interesting ways; in the worst case, it may even report false positives. A few of the preconditions mentioned by participants are easy to satisfy randomly—for example non-empty strings or lists—but most are quite hard to satisfy: valid postal addresses, well-structured XML documents, red-black trees, and syntactically valid S-expressions are extremely unlikely to be generated by a naïve random generator. Participants also described test data requirements going beyond precondition validity. Some (5/30) said they wanted their test data to be "realistic"—i.e., similar to the distributions of data the application was likely to see in the real world. Others described heuristics for the distribution of the input data, for example generating lists or trees of a "reasonable size" (P9).

Participants described a few different ways that they generated inputs. Most talked about either handwritten random generators (19/30), which are the default approach in libraries like QuickCheck, or *derived* generators (19/30), which are inferred based on the types of the data to be generated (e.g., the type `int list` implies a generic generator for lists of integers).

The need for handwritten generators seemed to be a source of friction for many participants. P6 said that writing generators for data that satisfies property preconditions is "the biggest annoyance with trying to use QuickCheck" and many participants thought of writing generators as "tedious" (6/30) or "high-effort" (7/30). P2 described the task of writing generators as intruding into their development process and contributing to a general perception of PBT as "high cost and low value." Since PBT was usually described as an integral part of the development process, rather than as a separate quality-assurance task, it makes sense that requiring detours into a lengthy generator design process might make PBT less desirable.

Besides requiring significant effort to use, available tools for writing generators by hand do not easily enable developers to produce precondition-satisfying values that are also well distributed in the way they would like. Writing a precondition-satisfying generator on its own can be a large task—whole papers have been written about generators for a single complex data type [134]—and accounting for distributional considerations adds even more friction. P13 said, "there's a tension in…all these handwritten generators between 'I want kind of coverage of everything' and 'I want coverage that is realistic for most inputs.'" As a solution, P13 suggested that one might be able to carefully combine two different handwritten generators to achieve these competing goals, but thought "that way lies madness". Other participants had an idea of the kinds of values they wanted to generate more or less frequently, but they were not sure how to get there: "What should [the probabilities in my generator] look like?…I'm sure if I studied probability and statistics and fully understood how the QuickCheck generating system worked, I could give better guesses. But all my guesses…they're not educated guesses. They're just random…And that's a little bit of a mental strain." (P20)

In Jane Street's ecosystem, derived generators are enabled via the `ppx_quickcheck` library, which provides a preprocessor that runs before the compiler and synthesizes generator code from type information. Ideally, this approach is totally automatic, providing a generator without any additional user input, and it was described fondly by participants. Many included it in their workflows (19/30), and one called it "f***ing amazing" (P5). P13 went so far as to suggest that "you shouldn't really be handwriting generators, you should be changing the structure of your type [to improve generation]." (For example, if a function being tested takes an `int` flag, but the only valid values are `0`, `1`, and `2`, then replacing `int` with an appropriate enumerated type causes `ppx_quickcheck` to derive a better generator.) Others (7/30) leveraged a combination of derived and handwritten generators, using derived generators as a foundation on which to build more complex generator programs.

For both handwritten and derived generators, Jane Street developers recognized opportunities for tool improvements. P4 described the process of hand-writing a generator for a mutable data structure as "overwhelming," and P6 suggested that libraries could do a better job supporting that use case. As for improving PPX-derived generators, P26 wanted better support for generating values of *generalized algebraic data types* (GADTs), special types in OCaml and some other languages that can express fine-grained properties of data.

Stepping outside the world of random generation, participants also described several additional approaches to obtaining test inputs. P16 described an approach that involved mutating a few user-provided values, rather than generating new values (this idea is related to ideas in fuzz testing, but used outside of any fuzzing framework; we discuss fuzzing below). P19 and P22 used enumeration, testing the whole space of possible input values, rather than random generation. And P25, P27, and P30 all used some kind of real-world data to test their properties. These techniques are a powerful complement to random generation and may benefit from support from PBT frameworks.

2.4.5 *Understanding Failure*

Debugging is often tricky, but participants described ways that tracking down bugs detected by PBT can be especially so. One participant described their debugging process: after finding a large, unwieldy counter-example with PBT, "you have to pull that failure into a separate sort of regression test…which runs the same infrastructure but does it with much more detail…Having done that, it is still sometimes unclear—like what about this example actually causes us to fail?" (P1) This is an instance of a broader problem: random test generators often produce failing examples that are too large to make sense of during debugging. To help developers understand failing examples, PBT frameworks offer shrinkers that transform large inputs into smaller inputs that (presumably) introduce the same bug. One participant in our study deemed shrinking "necessary" (P15), and two who implemented their own ad-hoc PBT frameworks (P8 and P21) insisted that shrinking was one of the most important features they implemented.

That said, participants found it difficult to use the shrinking functionality in QuickCheck, which resembles shrinking in many PBT frameworks. In such frameworks, shrinking is achieved by having users manually write functions that incrementally reduce a large value (e.g., a string, list, or other more complex data structure) to a smaller one that triggers the same failure. Participants saw writing shrinkers as an undesirable activity: one plainly stated, "I hate writing shrinkers" (P4). One challenge of writing shrinkers was writing them in a way that preserved important non-trivial invariants: "It's easy to write a shrinker that accidentally does not preserve some invariant, and that makes your test fail." (P13) This led one participant to ask for a "generic solution [for shrinking], rather than having the user write shrinkers" (P10).

A few participants also described wanting more information from the shrinking process—for example, the progressively smaller intermediate values that were found during shrinking. P1 said "the shrinker gets it right…but [it's] still useful to see the evolution."

2.4.6 *Understanding Success*

Property-based testing not only requires a developer to understand test failures; it also requires developers to understand whether *passing* a test actually means the software is correct. If the inputs provided by the test harness fail to adequately exercise buggy code, a property-based test may pass misleadingly. One participant described this situation, recalling a bug in published code that their property-based tests failed to catch because the generator was "not generating the case that [they] had in mind" (P19). P14 worried they could not trust their generators because they had "no idea what it's generating." (P14)

But, while participants understood that, in principle, tests could pass erroneously, 11 of them—more than a third—said they did not think as hard as they should about testing effectiveness, or whether they had adequately tested their software with their generators. Some (3 of the 11) reported seeing their property catch a couple of bugs and deciding that they did not have to improve their properties any further; the rest trusted that the generators provided by OCaml's QuickCheck library or the ones they derived or wrote themselves would be good enough. While this group is a minority of our sample, even a signal of this size is striking: PBT tools such as derived generators should make it easy for developers to get started; they should not (but evidently sometimes do) discourage developers from being critical of their test suite.

On the other hand, several participants described techniques for validating their PBT tests:

Mutation Testing (7/30). Some participants intentionally added bugs to their code and checked that their tests successfully found those bugs. This technique is standard in the testing literature [135; 138] and used in testing benchmarks such as Magma [69] and Etna [153].

Example Inspection (8/30). An even simpler way to assess a generator's distribution is to look at a handful of examples that the generator produces; one participant (P26) even designed a small utility to graphically render one example at a time, to make them easier to understand at a glance.

Code Coverage (2/30). Participants evaluated the code coverage achieved by their tests and used code coverage measurements as an indication that their properties were thoroughly exploring the space of program behaviors.

Property Coverage (1/30). Another participant measured coverage not of the system under test, but of the property itself. This is a weaker measurement, since it does not say anything about the system under test, but it is much easier to make because it can be measured without complex tooling.

A couple of participants (2/30) compensated for gaps in their property-based tests by supplementing them with example-based unit tests. In these cases, example-based tests were written to test complementary functionality that could not be easily tested with properties. As one participant put it, "it's kind of not a good idea to use QuickCheck in isolation" (P6), as doing so limits the breadth of software behaviors that can be tested.

How might PBT frameworks provide better support these (and other) techniques that give insight into how thoroughly properties have been tested? P15 described their ideal interaction with PBT tools, where the tools give "insight into what [PBT] is doing…I think a combination of knowing what it's doing and having more control of the space that it's exploring would be interesting." Participants desired greater awareness of what their tools were doing: P16 called this "visibility", and P30 called it "inspectability." Many participants wanted greater visibility to better understand the breadth of inputs generated. P18, for instance, desired "visualization of the test [inputs] being generated". Others wished to see details such as "the answer [of the function under test] on a smaller set of examples" (P9), "statistics on…edge cases" (P18), statistics describing generated inputs such as "the average length of [a generated] list" (P25), and code coverage (P9).

2.4.7 *Tradeoffs*

As the earlier sections indicate, using PBT is not without costs. From coming up with properties to evaluating testing success, participants found friction in many steps of PBT. Participants described seeing themselves as employing PBT more often if its "overhead" could be lowered (P26), or if there was less "effort necessary to integrate [it] into [their] workflow" (P10).

Amidst these costs, PBT was often seen as worth the effort. For some participants, development of properties was unremarkable. For others with more involved implementation, it was still worth the costs: in the words of P6,"[It was] a huge slog, but I was like 'this needs to be right,' and…I'm glad I did it." P17 describes the tradeoffs of using PBT as follows: "doing PBT is both putting in more effort and saving oneself effort as well."

2.5 LESSONS LEARNED

In this section we answer our research questions, setting a course for upcoming PBT research and tool development that is grounded in findings from the study. Our recommendations cut across the PBT process, and establish new goals and priorities for different areas of PBT research. We first answer **RQ1** with observations of PBT's practice that offer a reality check to PBT researchers and tool-builders (§2.5.1), then we answer **RQ2** with a technical and empirical research agenda motivated by our study (§2.5.2).

Observations

OB1	Property-based testing is being used successfully to build confidence in complex systems.
OB2	Properties are used as devices for communicating specifications.
OB3	Property-based tests need to be *fast*; they are expected to perform as well as other unit tests.
OB4	Property-based testing is used opportunistically in high-leverage scenarios where properties are readily available; developers rarely go out of their way to write subtle specifications.
OB5	Developers see writing generators as a distraction, preferring to use derived generators.
OB6	Developers may not interrogate properties that do not find bugs, even in cases where they acknowledge they should.

Research Opportunities

RO1	Understand time constraints for property-based tests.
RO2	Improve support for differential and model-based testing.
RO3	Make more testing scenarios high leverage.
RO4	Streamline the process of writing well-distributed, precondition-satisfying generators.
RO5	Improve interfaces for shrinking.
RO6	Improve tools for evaluating testing effectiveness.
RO7	Connect evaluation of testing effectiveness and generator improvement.

Figure 2.2: Summary of results.

2.5.1 *The State of Practice*

Recall that **RQ1** asks, "What are the characteristics of a successful and mature PBT culture at a software company?" Based on the results of our study, we answer: A mature PBT culture considers PBT a tool to be opportunistically applied in situations of high leverage to gain confidence in software and document its behavior; developers try to keep PBT "out of their way," expecting it to work quickly and easily. In this section, we synthesize the results from the previous section into observations that expand on these ideas. Some observations point forward to §2.5.2, where we present ideas for future research based on our findings.

OB1: *Property-based testing is being used successfully to build confidence in complex systems.* The broader research community sometimes situates PBT as a lower-effort, lower-reward alternative to formal proofs of correctness, but that perspective underestimates PBT as a practical tool for improving software quality. In §2.4.1, we show that the developers at Jane Street found PBT to be a valuable part of their testing toolbox, using it to gain confidence when they were unsure of code's correctness, especially where correctness was of critical importance. PBT gave them confidence that their software was operating correctly, finding bugs in code that had not been found via other methods.

OB2: *Properties are used as devices for communicating specifications.* Tests are generally a valuable form of documentation, and properties are no exception. Jane Street developers use them as executable evidence that code satisfies a specification, which has significant advantages over traditional documentation that might go stale as the code changes. As shown in §2.4.1 and §2.4.2, properties were deemed especially useful for communication during code review: when properties were available, they were considered a strong signal that the code was correct, and when unavailable reviewers sometimes asked the submitter to write some. These auxiliary uses for properties are worth keeping in mind for PBT framework designers; for example, some property languages try to make properties easier to write by providing terse syntax and expressive defaults, but those features may cause problems if they hurt readability.

OB3: *Property-based tests need to be* fast*; they are expected to perform as well as other unit tests.* The PBT literature is not always clear about exactly when in the software engineering process they expect properties to be written, but in §2.4.2 we observe that PBT's niche is testing module-level code during development. This is in contrast with fuzz testing, which, when used by Jane Street developers, is treated as a separate step and run outside of the standard testing workflow. This discrepancy makes sense in view of the differing goals that we observed for PBT and fuzzing: PBT is used for module-level tests with often-complex logical specifications, while fuzzing is used for integration-style tests of whole applications, to check for relatively basic (e.g., assertion failure or uncaught exception) errors. (Look ahead to RO1.)

Properties live alongside other unit tests, so they are expected to run as quickly as other unit tests. Knowing this may change priorities for some PBT researchers. If users are only testing their properties for 50 milliseconds, research advances that increase input generation or property execution speed might make the difference between bugs being found or not. Conversely, approaches that make generators more thorough but significantly slower may not be used unless developers are given reason to accept longer execution times.

OB4: *Property-based testing is used opportunistically in high-leverage scenarios where properties are readily available; developers rarely go out of their way to write subtle specifications.* Conventional wisdom in the PBT community sometimes assumes that developers decide to use PBT and then try to think of a specification, but the study participants generally did the opposite: they saw an obvious testable property and then decided to use PBT. We call situations with these readily available properties "high-leverage" testing scenarios.

The high-leverage scenarios varied, and we are not confident that we have exhaustively documented them, but a few are described in §2.4.3. The most popular was differential or model-based testing, which compares the code under test to some other available implementation. (We are not surprised our participants found these techniques useful, after all both differential and model-based testing have rich literatures on their own, but we did not expect to see them so seamlessly incorporated into PBT workflows.) Other high-leverage properties include round-trip properties that check an inverse relationship between functions and catastrophic failure properties that make sure a program does not fail completely.

What does this opportunistic use of PBT mean for researchers and tool builders? Frameworks can optimize for and automate these scenarios—tools like Hypothesis Ghostwriter [67] have already begun incorporating common PBT scenarios into an automation framework—improving the common case and accelerating PBT use even further. High-leverage scenarios should also be incorporated into PBT benchmarks like Etna [153] to make sure that the performance of PBT algorithms is evaluated in a way that reflects their use in real-world scenarios. (Look ahead to RO2 and RO3.)

OB5: *Developers see writing generators as a distraction, preferring to use derived generators.* Since PBT is often done in the midst of development, developers are reluctant to slow down and write a generator; the task was seen as both difficult and time-consuming. Instead, as seen in §2.4.4, many participants opted to test with generators derived from the types of their programs.

This has two implications. First, it means that ongoing work towards improving generator automation is critical. After all, as easy as current derived generators are to use, they are not always sufficient. The well-liked `ppx_quickcheck` library, for example, cannot derive generators for properties with complex preconditions. Broadening applicability of derived generators lowers barriers to using PBT. Second, it means that developers of manually written generator languages must carefully consider any added friction. Languages that make generators more difficult to write, perhaps because doing so enables desirable new features, should be cautious—these features may be unlikely to see adoption.

Separately, languages and tools for writing generators should provide a wealth of sensible defaults, and they should optimize the user experience around common patterns. Participants indicated that there was room for improvement in all of these areas, at least in OCaml's QuickCheck. (Look ahead to RO4.)

OB6: *Developers may not interrogate properties that do not find bugs, even in cases where they acknowledge they should.* A passing property sometimes means that a program is free of bugs, but it may also mean that the input values are not the right ones to *trigger* a latent bug. Developers acknowledged this fact, and they had expectations around the kinds of input values that they wanted when testing their properties. (Besides satisfying property preconditions, they wanted them to be realistic, well-distributed in the space, interesting enough to cover corner and edge cases, and more.) But, as shown in §2.4.6, when it came time to decide if their generators met expectations, developers did not analyze them closely. Developers of PBT frameworks, and especially PBT automation, should keep this in mind: automated tools for generating test inputs risk giving developers a false sense of security. They must ensure that their tools test thoroughly, because testers may not. (Look ahead to RO6.)

2.5.2 *Research Opportunities*

Now we tackle **RQ2**: "What opportunities exist for future work in the PBT space, motivated by the needs of real developers?" Based on our results, we conclude: Exciting avenues for PBT research include further studies into performance and usability of PBT generators, broader and deeper support of high-leverage PBT scenarios, better tools for shrinking, and better visibility into the testing process. This section discusses research opportunities in software engineering, programming languages, and human-computer interaction research that will unlock the yet-unrealized potential of PBT. Some research opportunities point backward to §2.5.1, where we synthesize observations that inform these ideas.

RO1: *Understand time constraints for property-based tests.* Future studies should more thoroughly explore how long developers actually budget for running PBT. In this study, we heard about numbers between 50 milliseconds and 30 seconds; that difference is massive, and tools that support PBT cannot make optimal decisions around optimization without clearer data. Moreover, developers of tools that combine ideas from PBT and fuzzing need to carefully examine their performance and recognize that many PBT users set strict timeouts that may preclude the overhead of measuring code coverage while running tests. This is encouraging for tools like Crowbar [31] and HypoFuzz [66] that allow developers to transition between short-running PBT and longer-running fuzzing with the same properties. (See OB3.)

RO2: *Improve support for differential and model-based testing.* As the most popular high-leverage scenario for PBT, differential testing seems particularly interesting as a topic of further research. Differential testing has a rich literature, as discussed in §2.6, but in the context of PBT there are still advances within reach. For example, in languages like OCaml with rich module structures, researchers should aim to increase automation around differential testing and produce a test harness for comparing modules without requiring any manual setup; Hypothesis Ghostwriter has begun to incorporate similar ideas with Python's classes. (See OB4.)

RO3: *Make more testing scenarios high leverage.* Some testing situations are just barely outside the realm of "high-leverage" scenarios, and improved tooling could make the difference. For example, P5 described a technique that they used to test a poorly abstracted module with an overly complicated interface: instead of writing normal properties, they instrumented the module with log statements, wrote properties about what those logs should look like, and then tested the module via an external interface that hides many of its internal details. Generalizing and operationalizing this technique—e.g., perhaps with temporal logic in the style of Quickstrom [130]—would allow for better testing leverage in cases where poor abstraction prevents traditional PBT.

Other testing situations might be supported better as well. Potential opportunities include improving PBT support for code with mutable state (e.g., by saving memory snapshots for repeatable tests) and code that interacts with the environment (e.g., via robust integration with tools for *mocking* [114]). Increasing the scenarios in which PBT works out of the box will naturally make it higher-value for developers. (See OB4.)

RO4: *Streamline the process of writing well-distributed, precondition-satisfying generators.* This has been a research goal for the PBT community since the beginning. Our study clarifies some promising paths forward, including improving tools for automated tuning, generalizing unified languages for defining generators alongside properties, and supporting alternatives to randomness as first-class.

There are two main options for tuning generator distributions. Actively tuned generators modify their distribution live, in response to feedback, whereas pre-tuned generators compute distributions ahead of time. Actively tuned generators are the norm in the fuzzing literature [39] and have been ported to PBT in many forms [104; 144; 31; 55], but they spend precious testing time on tuning analysis, making them less useful in time-constrained PBT scenarios. Pre-tuned generators can be much faster, but they currently require too much programmer effort. For example, reflective generators [58] (which build on example-based tuning *à la* the *Inputs from Hell* approach [155]) automate the process of generating realistic test inputs, but this automation is only possible if a reflective generator is already available. Moving forward, the community should continue to look for ways to use current active-tuning strategies for pre-tuning (e.g., by saving inputs to be run later) and ways to make pre-tuned generators easier to use (e.g., with interfaces that help developers write complex generators).

When it comes to precondition-satisfying generators, many seek to unify languages for writing generators with ones for writing preconditions. One approach, used in the QuickChick PBT framework [136], uses inductive relations as a language for both properties and generators [137; 103]. This approach works well in the Coq proof assistant, but complex inductive relations are not expressible in mainstream languages or accessible to non-specialists. Researchers should consider ways to generalize these results. Alternatively, dedicated languages such as ISLa [159] use a common language that is closer to the logical connectives one might use in a standard programming language; more research should be done to evaluate if this approach can be applied in common PBT scenarios. Whatever dual-purpose language is chosen, this is a compelling path forward. Finally, obtaining good inputs sometimes means eschewing randomness in favor of enumerators or some other approach. The variety of ways that study participants obtained test inputs supports recent efforts replace "generators" with a more generic "producers" or "strategies" that support random generation, enumeration, and other approaches. Frameworks like Hypothesis already do this, and other frameworks would benefit from following suit. (See OB5.)

RO5: *Improve interfaces for shrinking.* It is clear from the study results that shrinkers would be far more useful if they were both more automated and more informative. As we mention above, we recommend that existing frameworks improve automation by incorporating internal test-case reduction [112; 158], which uses generators to aid in shrinking, where possible. Internal shrinking has the added benefit of always producing valid values, which is difficult to achieve otherwise.

Participants also asked to see more intermediate examples from the shrinking process. Indeed, there are cases where the smallest failing example is *not* the most helpful one: e.g., if the shrinker outputs the tuple `(0, 0)`, one might conclude that any tuple of integers triggers the bug, but it may be that the bug is only found if the first component is actually `0`. This example motivated Hypothesis to begin developing a tool that will give users a variety of tools for exploring failing tests. Debugging cases using shrinking might mean showing many shrunk examples to the user, following related work in the model-finding literature [27; 36], or even providing control over the shrinking process (e.g., with novel interactions allowing the user to click on components of a value to shrink only those components).

RO6: *Improve tools for evaluating testing effectiveness.* Participants reported ad-hoc, piecemeal approaches to understanding their testing effectiveness, and they asked for better ways to visualize testing feedback. As a simple first step, tools should always announce counts of discarded test cases (i.e., ones that failed the property's precondition) so the developer can catch problems early. Many PBT tools provide some way to aggregate statistics while a property is running (see Haskell QuickCheck's `label` and `collect` functions) but OCaml's QuickCheck hides output when tests succeed, which obscures that information, and more should be done around the usability and legibility of these kinds of aggregations. Going further, participants desired (1) better interfaces for scanning through examples of generated values, (2) better ways of visualizing generated distributions, and (3) better integration of code and branch coverage information. These could significantly improve developers' understanding of how thoroughly their code has been tested. (See OB6.)

RO7: *Connect evaluation of testing effectiveness and generator improvement.* How might a developer clearly express their test data goals *and* ensure that the generator achieves them? Future tools could tighten the feedback loop wherein a developer evaluates and then expresses how to improve their generator at the same time. For instance, a future user interface might display a generator's distribution in some graphical form like a bar chart and allow the developer to directly manipulate the distribution by dragging bars up and down. Alternatively, the developer might be able to click on those annotations to request that the distribution try harder to cover a particular line or balance a particular branch, in the style of AFLGo [17]. In an ideal world, an insufficient generator could be caught and fixed in a few clicks, without allowing bugs slip by.

2.6 RELATED WORK

We focus, in this section, on related work exploring the usability of testing and formal methods tools. Prior work on PBT, testing, and formal methods usability has made important observations about the challenges of specification and bug finding, but it has lacked the depth and domain focus to paint a clear picture of PBT, its usage, and opportunities for improvement.

Property-based testing. As a precursor to this study, our group did a smaller-scale pilot study with developers using Hypothesis [57]. The full-scale study is far more in-depth, and presents more detailed and nuanced findings, although talking to Python developers did raise a few concerns that were less prevalent at Jane Street, especially when it comes to difficulty coming up with specifications.

A study analyzing open-source libraries using Hypothesis [25] evaluated the kinds of properties that developers test in practice, and found significant overlap with our "high-leverage" testing scenarios. In particular, they found that both round-trip and differential or model-based testing are overrepresented in real-world tests.

An experience report from Amazon [11] described *differential testing* as a major use-case of PBT and reported that developers used shrinking tools for debugging and *mutation testing* to ensure testing effectiveness. Our study confirms these patterns and explores further PBT use-cases (§2.4.3), insights around shrinking (§2.4.5), and concerns about testing effectiveness (§2.4.6).

Other studies highlight a more narrow set of specific challenges faced by developers using PBT. One experience report describing PBT use at DropBox [82] cited usability problems when testing timing-dependent code. The report found that sequences of timed operations resist shrinking, often remaining unwieldy, and that timing dependence caused tests to be flaky. An education-focused study using PBT [176] observed that the PBT community lacks good motivating examples. While the former concern only appeared in passing in our study (when discussing PBT's handling of stateful software), the latter—a dearth of good motivating examples—might be ameliorated by our characterization of *high-leverage* testing scenarios in §2.5.

Testing more generally. Beyond PBT, there is considerable work studying usability software of testing in general. Our study sheds light into how these common testing issues manifest in PBT specifically. Two studies of developers who use IDEs [8; 9] conclude that testing, especially Test Driven Development, is not as prevalent as conventional wisdom would suggest. Based on these studies, Beller et al. coined the term "Test *Guided* Development" to describe the strategy that programmers actually use. Our study agrees with the idea that developers use testing to guide their thinking during development (§2.4.1). We also corroborate challenges that another study found around integration testing: Greiler et al. [62] found that while unit testing is common, integration testing is difficult and often left to the software's users; in our study, developers struggled to use PBT for integration testing because it is difficult to write specifications of entire systems' behaviors. (§2.4.3).

Our study also suggests that PBT addresses testing difficulties raised in the literature. Aniche et al. [3] observed that developers try to write relatively "random" test cases, with the hope of accidentally stumbling on bugs; this is related to developers' goals for PBT generators, discussed in §2.4.4, although PBT has the advantage of automating this process in many cases. Another study [63], focusing on differential testing, found PBT to be a valuable tool in a developer's toolbox, providing a coherence check for important code; however, they also found some problems with differential testing systems, such as naïve sampling algorithms that failed to trigger bugs and large counter-examples that were difficult to reason about. Our study suggests new tools may address these problems.

Formal methods. PBT is sometimes described as a "lightweight formal method." Indeed, a recent position paper [145] argued that testing techniques like PBT and fuzzing were important steps on the way towards more formal verification. Formal methods as a field have seen similar calls for usability improvements. A report out of the Naval Research Lab [71] says, "to be useful to software practitioners, most of whom lack advanced mathematical training and theorem proving skills, current formal methods need a number of additional attributes, including more user-friendly notations, completely automatic (i.e., push-button) analysis, and useful, easy to understand feedback." This report was published in 1998, but, as our study shows, some of their usability criteria are still not adequately met by modern PBT tools. A more contemporary account agrees that "the user experience of formal methods tools has largely been under-studied” [99] and calls for better education and tools for writing and understanding specifications.

2.7 CONCLUSION

Our study reveals that, even after two decades of active exploration—and, increasingly, exploitation—of PBT, there is still much to learn about how it is being used and what challenges it faces in practice. We contribute a wealth of observations about PBT’s use in an industrial setting, along with well-founded ideas for future research in PBT that we, with the help of the broader community, hope to pursue in the coming years.

CHAPTER 3

Parsing Randomness

This chapter tackles one of the research opportunities from Chapter 2, RO4, which highlights the need for better ways to express precondition-satisfying PBT generators. I present *free generators* a formalization of random data generators that clarifies folklore understanding of generator operation and enables new generation algorithms. While the work in this chapter was done before the *Property-Based Testing in Practice* study, the abstractions and algorithms I present were not yet available to the study participants. Since the study, I have begun conversations with PBT practitioners to see if ideas from this chapter can improve real-world users' experiences. The content in this chapter is taken from *Parsing Randomness* [55], published at the conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2022. The paper was written with help and guidance from Benjamin C. Pierce.

3.1 INTRODUCTION

"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 [20]—is a transformer from a series of random choices into a data structure, just as a parser is a transformer from a series of characters into a data structure.

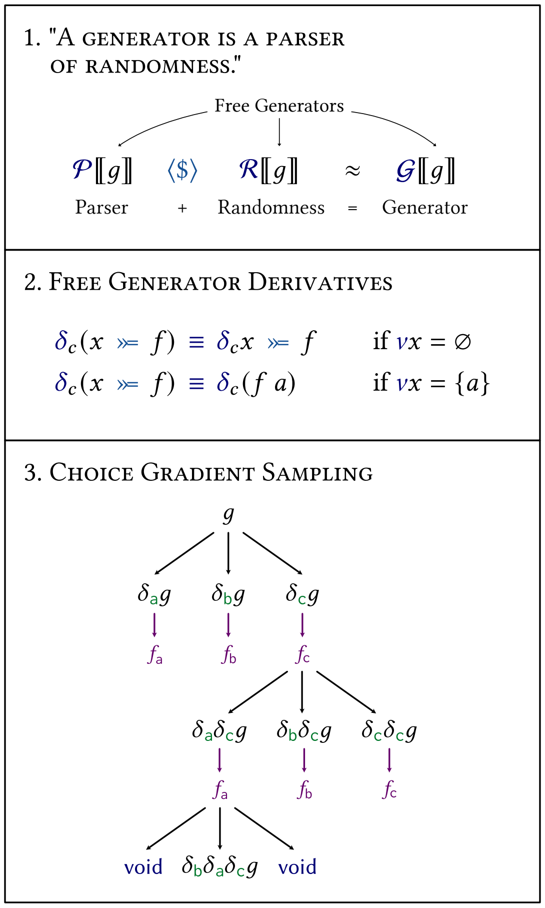
While this connection may be obvious once it is pointed out, few actually think of generators this way. Indeed, to our knowledge the framing of random generators as parsers has never been explored formally. The relationship between these fundamental concepts deserves a deeper look!

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 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 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: “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.

We use derivatives of free generators to tackle a well-known problem—we call it the *valid generation problem*. The challenge is to generate many random values that satisfy some validity condition. This problem comes up often in property-based testing, where the validity condition is the precondition of some functional specification. Since generator derivatives give a way of previewing the effects of a particular choice, we can use *gradients* (derivatives with respect to a vector of choices) to preview all possible choices and pick a promising one. This leads us to an elegant algorithm that takes a free generator and replaces its distribution with one that produces only valid values. Replacing the distribution in this way trades the benefits of the programmer’s tuning effort for a higher chance of finding valid inputs to test with.



1. "A generator is a parser of randomness."
Free Generators
$\mathcal{P}[\![g]\!]$ $\langle\$\rangle$ $\mathcal{R}[\![g]\!]$ $\approx$ $\mathcal{G}[\![g]\!]$
Parser + Randomness = Generator
2. Free Generator Derivatives
$\delta_c(x \gg\!= f) \equiv \delta_c x \gg\!= f$ if $\nu x = \varnothing$
$\delta_c(x \gg\!= f) \equiv \delta_c(f\ a)$ if $\nu x = \{a\}$
3. Choice Gradient Sampling
g
$\delta_a g$ $\delta_b g$ $\delta_c g$
f_a f_b f_c
$\delta_a\delta_c g$ $\delta_b\delta_c g$ $\delta_c\delta_c g$
f_a f_b f_c
void $\delta_b\delta_a\delta_c g$ void

Figure 3.1: Our contributions.

In §3.2 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 (§3.3). We use free generators to prove that any finitely supported *monadic generator* can be factored into a parser and a distribution over strings.

- We exploit free generators to transport an idea from formal languages—the *Brzozowski derivative*—to the context of generators (§3.4).

- To illustrate the potential applications of these formal results, we present an algorithm that uses derivatives to turn a naïve generator into one with a different distribution, assigning nonzero probability only to values satisfying a Boolean precondition (§3.5). Our algorithm performs well on a set of simple benchmarks, in most cases producing more than twice as many valid values as a naïve "rejection sampling" generator in the same amount of time (§3.6).

We conclude with related and future work (§3.8 and §3.9).

3.2 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
```

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.

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

`n5ll` into `Node Leaf 5 Leaf` and `n5ln8ll` 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 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`.

3.2.1 *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.

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:

```
fgenTree $h$ =
    if $h = 0$ then
        return Leaf
    else
        $c \leftarrow$ pick [(1, l, return False), (3, n, return True)]
        if $c$ == False then return Leaf
        if $c$ == True then
            $x \leftarrow$ fgenInt ()
            $l \leftarrow$ fgenTree $(h - 1)$
            $r \leftarrow$ 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 is simply an abstract syntax tree (AST).

The syntactic nature of free generators means that they can simultaneously represent generators, parsers, and more. In §3.3 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,

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

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 produces the distribution of sequences of choices that the random generator interpretation makes. Now, for any free generator g, we have

$$\mathcal{P}[\![g]\!] \mathbin{<\$>} \mathcal{R}[\![g]\!] \approx \mathcal{G}[\![g]\!]$$

where <$> is a "mapping" operation that applies a function to samples from a distribution (see Theorem 1 below). 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 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!

3.2.2 *Derivatives of Free Generators*

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 [14]. 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:

```
parseTree 10 =                      $\delta_{\mathtt{n}}^{\mathcal{L}}$(parseTree 10) ≈         $\delta_{\mathtt{5}}^{\mathcal{L}}\delta_{\mathtt{n}}^{\mathcal{L}}$(parseTree 10) ≈
    $c$ ← consume()
    if $c$ == l then
        return Leaf
    if $c$ == n then
        $x$ ← parseInt()                $x$ ← parseInt()
        $l$ ← parseTree 9               $l$ ← parseTree 9               $l$ ← parseTree 9
        $r$ ← parseTree 9               $r$ ← parseTree 9               $r$ ← parseTree 9
        return Node $l$ $x$ $r$         return Node $l$ $x$ $r$         return Node $l$ 5 $r$
    else fail
```

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:

```
fgenTree 10 =                  $\delta^{\mathcal{L}}_{\mathsf{n}}$(fgenTree 10) $\approx$          $\delta^{\mathcal{L}}_{5}\delta^{\mathcal{L}}_{\mathsf{n}}$(fgenTree 10) $\approx$
    $c \leftarrow$ pick [...]
    if $c$ == False then
        return Leaf
    if $c$ == True then
        $x \leftarrow$ fgenInt()           $x \leftarrow$ fgenInt()
        $l \leftarrow$ fgenTree 9          $l \leftarrow$ fgenTree 9              $l \leftarrow$ fgenTree 9
        $r \leftarrow$ fgenTree 9          $r \leftarrow$ fgenTree 9              $r \leftarrow$ fgenTree 9
        return Node $l$ $x$ $r$        return Node $l$ $x$ $r$            return Node $l$ 5 $r$
    else fail
```

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 derivative of a free generator via a simple and efficient syntactic transformation.

In §3.4 we define a procedure, $\delta^{\mathcal{L}}_{c}$, for computing the derivative of a free generator and prove it correct, in the sense that, for all free generators g,

$$\delta^{\mathcal{L}}_{c}\mathcal{L}[\![g]\!] = \mathcal{L}[\![\delta_c g]\!].$$

In other words, the derivative of the language of g is equal to the language of the derivative of g. (See Theorem 2.)

3.2.3 *Putting Free Generators to Work*

The derivative of a free generator is *the generator that remains after a particular choice.* This gives us a way of "previewing" the effect of making a choice by looking at the generator after fixing that choice.

In §3.5 and §3.6 we present and evaluate an algorithm called Choice Gradient Sampling that uses free generators to address the *valid generation problem.* Given a validity predicate on a data structure, the goal is to generate as many unique, valid structures as possible in a given amount of time. Starting from a simple free generator, our algorithm uses derivatives to evaluate choices and search for ones that produce valid values.

We evaluate the choice gradient sampling algorithm on four small benchmarks, all standard in the property-based testing literature. For each, we compare our algorithm to rejection sampling—sampling from a naïve generator and discarding invalid results—as a simple but useful baseline for understanding how well or algorithm performs. Our algorithm does remarkably well on three out of four benchmarks, generating more than double the valid values per minute of rejection sampling.

3.3 FREE GENERATORS

We now turn to developing the theory of free generators, beginning with some background on monadic abstractions for parsing and random generation.

3.3.1 *Background: Monadic Parsers and Generators*

In §3.2 we represented generators and parsers as pseudocode. Here we flesh out the details, presenting all definitions as Haskell programs, both for the sake of concreteness and also because Haskell's abstraction features (e.g., type-classes) allow us to focus on the key concepts. Haskell is a lazy functional language, but, as we focus our attention on finite programs, our results should apply directly to eager functional languages. It may also be possible, with appropriate domain knowledge, to translate these ideas to idiomatic constructs in popular imperative languages [139].

We represent both generators and parsers using *monads* [125]. 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 [20] and parseTree like we would if using libraries like Parsec [106]:

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

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

In the first program, genTree, 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

```
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." Formally, genTree denotes a distribution over binary trees (e.g., an arrow in an appropriate category [44]), and running the program samples from that distribution.

We can see these same combinators (used with a different monad) in parseTree. There, 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:

```
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.

Expressiveness Relative to Other Abstractions Monadic parsers and generators are maximally expressive in their respective domains. Monadic parsers can parse arbitrary computable languages, subsuming more restricted parser descriptions like context-free grammars and regular expressions. Likewise, monadic generators 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.

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)]
  case c of
    False -> return Leaf
    True -> do
      x <- genRange (lo, hi)
      l <- genBST (lo, x - 1)
      r <- genBST (x + 1, hi)
      return (Node l x r)
```

The generator maintains the BST invariant by keeping track of the minimum and maximum values available for a given subtree 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 stochastic CFG, since there is dependence between the choice of value `x` and the choices of subtrees. Our examples focus on simple (non-dependent) generators to streamline the exposition, but our theory applies to the full class of monadic generators with finitely supported distributions.

3.3.2 *Representing Free Generators*

With the monad interface in mind, we can now give the formal definition of a free generator.[1]

[1]For 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.

Type Definition

The actual type of free generators is based on a structure called a *freer monad* [94]:

```
data Freer f a where
  Return :: a -> Freer f a
  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 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 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:

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

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 §3.2. 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 if and only if 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 -> ... )
```

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`).

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, 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:

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

(>>=) :: FGen a -> (a -> FGen b) -> FGen b
Return a >>= f = f a
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 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:

```
(<$>) :: (a -> b) -> FGen a -> FGen b
f <$> 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 §3.4, 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:

```
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:

```
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`:

```
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 handwritten 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
    [] -> 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 that the user has not duplicated any of the choice labels; this would introduce a nondeterministic choice that would complicate the interpretation considerably (see §3.7). 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.

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
  c <- pick [(1, l, return False), (3, n, return True)]
  case c of
    False -> return Leaf
    True -> do
      x <- fgenInt
      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 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 <$> fgenInt),
         (1, p, do
           e1 <- fgenExpr (h - 1)
           2 <- fgenExpr (h - 1)
           return (Plus e1 e2)),
         (1, l, do
           t <- fgenType
           e <- fgenExpr (h - 1)
           return (Lam t e)),
         (1, a, do
           e1 <- fgenExpr (h - 1)
           e2 <- fgenExpr (h - 1)
           return (App e1 e2)),
         (1, v, Var <$> 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. We will return to this problem in §3.6.

3.3.3 *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 §3.2 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]

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

```
𝒢⟦·⟧ :: FGen a -> Gen⊥ a
𝒢⟦Void⟧              = ⊥
𝒢⟦Return v⟧          = return v
𝒢⟦Bind (Pick xs) f⟧ = do
  x <- frequency (map (\ (w, _, x) -> (w, return x)) xs)
  a <- 𝒢⟦x⟧
  𝒢⟦f a⟧
```

Note that the operations on the right-hand side of this definition do *not* build a free generator; they are `Gen⊥` 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.

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:

```
𝒫⟦·⟧ :: FGen a -> Parser a
𝒫⟦Void⟧              = \s -> Nothing
𝒫⟦Return a⟧          = return a
𝒫⟦Bind (Pick xs) f⟧ = do
  c <- consume
  x <- case find ((== c) . snd) xs of
    Just (_, _, x) -> return x
    Nothing -> fail
  a <- 𝒫⟦x⟧
  𝒫⟦f a⟧
```

This time the `do`-notation on the right-hand side is interpreted using the `Parser` monad (as before, defined as `String -> Maybe (a, String)`). In the case for `Bind`, the parser consumes a character and attempts

[2]Our Haskell development implements partial generators with `Gen (Maybe a)`.

to make the corresponding choice from the list provided by `Pick`. If it succeeds, it 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 parser interpretation discards. We define the randomness interpretation of a free generator to be:

```
𝓡⟦·⟧ :: FGen a -> Gen⊥ String
𝓡⟦Void⟧              = ⊥
𝓡⟦Return a⟧          = return ε
𝓡⟦Bind (Pick xs) f⟧ = do
  (c, x) <- frequency (map (\ (w, c, x) -> (w, return (c, x))) xs)
  s <- 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 $g_1 \equiv g_2$, if and only if the generators 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 choice* *sequences that are, together, equivalent to its interpretation as a generator. In other words, for all free generators g,*

$$\mathcal{P}[\![g]\!] \texttt{<\$>} \mathcal{R}[\![g]\!] \equiv (\lambda x \to (x,\ \varepsilon)) \texttt{<\$>} \mathcal{G}[\![g]\!].$$

Proof sketch. By induction on the structure of g; see the Appendix 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.*

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,

$$(\lambda x \to (x, \varepsilon)) \texttt{<\$>} \gamma = (\lambda x \to (x, \varepsilon)) \texttt{<\$>} \mathcal{G}[\![g]\!] \equiv \mathcal{P}[\![g]\!] \texttt{<\$>} \mathcal{R}[\![g]\!],$$

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:

```
ℒ⟦·⟧ :: FGen a -> Set String
ℒ⟦Void⟧              = ∅
ℒ⟦Return a⟧          = ε
ℒ⟦Bind (Pick xs) f⟧ = [ c : s | (w, c, x) <- xs, s <- ℒ⟦x >>= f⟧ ]
```

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 >>= 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`, 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 every sequence 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 interpretation can make and that the parser interpretation can parse.

3.4 DERIVATIVES OF FREE GENERATORS

Next, we review the notion of Brzozowski 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 justifies taking the correspondence between generators and parsers seriously.

3.4.1 *Background: Derivatives of Languages*

The *Brzozowski derivative* [14] 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,

$$\delta_{\mathtt{a}}^{\mathcal{L}} \{\mathtt{abc}, \mathtt{aaa}, \mathtt{bba}\} = \{\mathtt{bc}, \mathtt{aa}\}.$$

Many formalisms for defining languages support syntactic transformations that correspond to Brzozowski derivatives. For example, we can take the derivative of a regular expression like this:

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

$$\begin{aligned}
\delta^{\mathcal{L}}_c \varnothing &= \varnothing \\
\delta^{\mathcal{L}}_c \varepsilon &= \varnothing \\
\delta^{\mathcal{L}}_c \mathtt{c} &= \varepsilon \quad (c = \mathtt{c}) \\
\delta^{\mathcal{L}}_c \mathtt{d} &= \varnothing \quad (c \neq \mathtt{d}) \\
\delta^{\mathcal{L}}_c (r_1 + r_2) &= \delta^{\mathcal{L}}_c r_1 + \delta^{\mathcal{L}}_c r_2 \\
\delta^{\mathcal{L}}_c (r_1 \cdot r_2) &= \delta^{\mathcal{L}}_c r_1 \cdot r_2 + \nu^{\mathcal{L}} r_1 \cdot \delta^{\mathcal{L}}_c r_2 \\
\delta^{\mathcal{L}}_c (r^*) &= \delta^{\mathcal{L}}_c r \cdot r^*
\end{aligned}
\qquad
\begin{aligned}
\nu^{\mathcal{L}} \varnothing &= \varnothing \\
\nu^{\mathcal{L}} \varepsilon &= \varepsilon \\
\nu^{\mathcal{L}} \mathtt{c} &= \varnothing \\
\nu^{\mathcal{L}} (r_1 + r_2) &= \nu^{\mathcal{L}} r_1 + \nu^{\mathcal{L}} r_2 \\
\nu^{\mathcal{L}} (r_1 \cdot r_2) &= \nu^{\mathcal{L}} r_1 \cdot \nu^{\mathcal{L}} r_2 \\
\nu^{\mathcal{L}} (r^*) &= \varepsilon
\end{aligned}$$

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 $\nu^{\mathcal{L}} r = \varnothing$.

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

3.4.2 *The Free Generator Derivative*

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 expression r, the expression $\nu^{\mathcal{L}} r$ is either $\varnothing$ or ε. Here, the null check returns either $\varnothing$ 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.

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:

$$
\begin{aligned}
\delta_c\,\texttt{void} &\equiv \texttt{void} && && (3.1)\\
\delta_c(\texttt{return}\ v) &\equiv \texttt{void} && && (3.2)\\
\delta_c(\texttt{pick}\ xs) &\equiv x && \text{if } (c,\ x) \in xs && (3.3)\\
\delta_c(\texttt{pick}\ xs) &\equiv \texttt{void} && \text{if } (c,\ x) \notin xs &&\\
\delta_c(x \ \texttt{»=}\ f) &\equiv \delta_c(f\ a) && \text{if } \nu x = \{a\} && (3.4)\\
\delta_c(x \ \texttt{»=}\ f) &\equiv \delta_c x \ \texttt{»=}\ f && \text{if } \nu x = \varnothing &&
\end{aligned}
$$

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 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 equations for `(>>=)` 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
δc(Return v)         = void
δc(Bind (Pick xs) f) =
  case find ((== c) . snd) xs of
    Just (_, _, x) -> x >>= f
    Nothing -> void
```

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 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 (3.1), (3.2), (3.3), and (3.4). In other words, the free generator derivative behaves similarly to the regular expression derivative.*

Proof sketch. See the Appendix 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 derivative. That is, for all free generators g and choices c,*

$$\delta_c^{\mathcal{L}} \mathcal{L}[\![g]\!] = \mathcal{L}[\![\delta_c g]\!].$$

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

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 repeated derivatives. Each derivative fixes a particular choice, so a sequence of derivatives fixes a choice sequence.

3.5 GENERATING VALID RESULTS WITH GRADIENTS

We now put the theory of free generators and their derivatives into practice. We introduce Choice Gradient Sampling (CGS), a novel algorithm for generating data that satisfies some given validity condition, given a simple free generator for data of the appropriate type.

The Choice Gradient Sampling algorithm starts with a free generator for data of some type and uses derivatives to step the generator through choices one at a time. This process guides the generator towards values that are valid with respect to a given validity condition. At each step, the algorithm looks at all available choices and takes the free generator's derivative with respect to each one. Since this is, in a sense, a vector of all possible derivatives, we call this the *gradient* of the free generator, by analogy with calculus. We write

$$\nabla g = \langle \delta_{\texttt{a}} g,\ \delta_{\texttt{b}} g,\ \delta_{\texttt{c}} g \rangle$$

for the gradient of g with respect to the available choices $\{\texttt{a}, \texttt{b}, \texttt{c}\}$.

Since each derivative in the gradient is itself a free generator, the derivatives can be interpreted as value generators and sampled. If the derivative with respect to `c` produces lots of valid samples, then `c` is a good choice. If it produces mostly invalid samples, maybe other choices would be better. As we discuss below, this process is not faithful to the distribution of the original generator, but it provides a metric that guides the algorithm toward a series of "good" choices, leading to more valid inputs in many cases.

```
1:  $g \leftarrow G$
2:  $\mathcal{V} \leftarrow \varnothing$
3:  while true do
4:      if $vg \neq \varnothing$ then return $vg \cup \mathcal{V}$
5:      if isVoid g then $g \leftarrow G$
6:      $C \leftarrow$ choices $g$
7:      $\nabla g \leftarrow \langle \delta_c g \mid c \in C \rangle$                ▷ $\nabla g$ is the gradient of $g$
8:      for $\delta_c g \in \nabla g$ do
9:          if isVoid $\delta_c g$ then
10:             $v \leftarrow \varnothing$
11:         else
12:             $x_1, \dots, x_N \leftsquigarrow \mathcal{G}[\![\delta_c g]\!]$        ▷ Sample $\mathcal{G}[\![\delta_c g]\!]$
13:             $v \leftarrow \{x_j \mid \varphi(x_j)\}$
14:         $f_c \leftarrow |v|$                                ▷ $f_c$ is the *fitness* of c
15:         $\mathcal{V} \leftarrow \mathcal{V} \cup v$
16:     if $\max_{c \in C} f_c = 0$ then
17:         for $c \in C$ do $f_c \leftarrow$ weightOf $c$ $G$
18:     $g \leftsquigarrow$ frequency $[(f_c, \delta_c g) \mid c \in C]$
```

Figure 3.3: Choice Gradient Sampling: Given a free generator G, a sample rate constant N, and a validity predicate φ, this algorithm produces a set of outputs that all satisfy $\varphi(x)$.

We present the CGS algorithm in detail in Figure 3.3. Lines 7–14 are the core of the algorithm; their execution is shown pictorially in Figure 3.4. We take the gradient of g by taking the derivative with respect to each possible choice, in this case a, b, and c. Then we evaluate each of the derivatives by interpreting the free generator with $\mathcal{G}[\![\cdot]\!]$, sampling values from the resulting value generator, and counting how many of those results are valid with respect to φ. The precise number of samples is controlled by N, the sample rate constant; this is up to the user, but in general higher values for N will give better information about each derivative

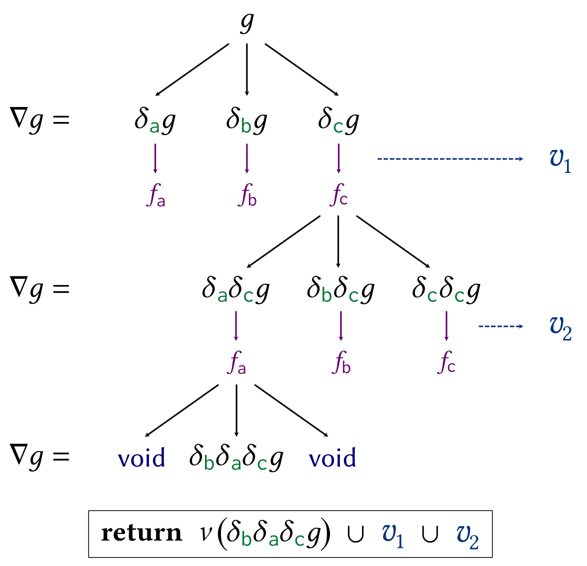


Figure 3.4: The main loop of Choice Gradient Sampling.

at the expense of time spent sampling. At the end of sampling, we have values f_{a}, f_{b}, and f_{c}, which we can think of as the "fitness" of each choice. We then pick a choice randomly, weighted based on fitness, and continue until our choices produce a valid output.

Critically, we avoid wasting effort by saving the samples ($\mathcal{V}$) that we use to evaluate the gradients. Many of those samples will be valid results that we can use, so there is no reason to throw them away. Still, note that the performance of this sampling does depend on $|C|$, the number of choices available at this point. If the generator has many valid choices at a given point, it will need to do a lot of sampling to decide which choice to make.

This sampling procedure would not be possible with a traditional monadic generator: free generators are key. Trying to take a derivative of a traditional monadic generator would be like taking the derivative of a black-box function—there would be no generic way to incrementalize evaluation. Free generators expose more structure, making derivatives (and thus CGS) possible.

3.5.1 *Impact on Distribution*

As noted above, this algorithm is not faithful to the original distribution of G. In particular, the observable behavior of the algorithm is *not* to sample from the original generator's distribution, conditioned on validity. While this property would arguably be ideal, it seems quite difficult to obtain. Moreover, its absence need not significantly detract from the value that CGS provides, for two reasons.

First, while the distribution produced by CGS is not faithful to the original distribution, it is certainly informed by it. At any given point in the algorithm, the weight given to a choice is based on how often making future choices, weighted by the original distribution, results in valid values. This means that valid values that are unlikely results from G will be unlikely results from CGS, and likely results from G will also be likely from CGS. Doing better than this would be quite difficult, since the preconditions we care about are black-box functions. This means that the only information they can provide is whether or not a particular value is valid, forcing us into rejection-based approaches. Standard rejection sampling does, in fact, sample from the ideal conditional distribution but it does so very slowly. Rather than sample from that distribution, CGS allows the predicate to guide its generation, reaching valid inputs more quickly.

Second, and more importantly, the primary use case for CGS is to improve the performance of free generators that are either automatically derived or else handwritten but not carefully tuned. That is, the algorithm is most effective as a low-effort way to get from a useless generator to a usable one. If a tester has strict requirements for the distribution they are after, CGS will likely not be sufficient; but as a quick way of getting up and running it can be quite helpful.

We have implemented our Choice Gradient Sampling algorithm in Haskell, along with the definitions presented throughout the paper [51].

3.6 EXPLORATORY EVALUATION

The Choice Gradient Sampling algorithm is not a tightly optimized production algorithm: it is a proof of concept. Primarily, CGS exists to illustrate the theory of free generators and their derivatives. Still, there is much to learn by exploring how well CGS is able to guide realistic generators to valid outputs.

We set out to answer two basic research questions:

RQ1 Does CGS produce more useful test inputs than standard sampling procedures, in the same period of time?

RQ2 Are the test inputs obtained from CGS well distributed in shape and size?

Our experimental results suggest that, with a few (interesting) caveats, these questions can both be answered in the affirmative. We find that CGS generally produces at least twice as many valid values as *rejection sampling* (explained in the next section) in the same period of time, and we also find that CGS's values are at least as diverse as the ones from rejection sampling. This indicates that guiding generation with derivatives is a promising approach to the valid generation problem.

3.6.1 *Experimental Setup*

Our experiments explore how well CGS improves on a canonical generation strategy. We compare our algorithm to the standard rejection sampling approach used by default in frameworks like QuickCheck, which takes a naïve generator, samples from it, and discards any results that are not valid. Rejection sampling is a useful point of comparison because, like our approach, it requires no extra effort from the user.

We use four simple free generators to test four different benchmarks: **BST**, **SORTED**, **AVL**, and **STLC**. Details about each of these benchmarks are given in Table 3.1.

Table 3.1: Overview of benchmarks.

	Free Generator	Validity Condition	N	Depth
BST	Binary trees with values 0–9	Is a valid BST	50	5
SORTED	Lists with values 0–9	Is sorted	50	20
AVL	AVL trees with values 0–9	Is a balanced AVL tree	500	5
STLC	Arbitrary ASTs for λ-terms	Is well-typed	400	5

Each of our benchmarks requires a simple free generator to act as a baseline and as a starting point for CGS. For consistency, and to avoid potential biases, our generators follow their respective inductive data types as closely as possible. For example, fgenTree, shown in §3.3 and used in the **BST** benchmark, follows the structure of the definition of the Tree type exactly. All generators use uniform choice weights, to avoid potential biases introduced by manual tuning.

The parameter N, used by CGS to decide how many samples to use for each iteration, was chosen via trial and error in order to balance fitness accuracy with sampling time. It is possible that some of our best-case results might improve with a more careful choice of N.

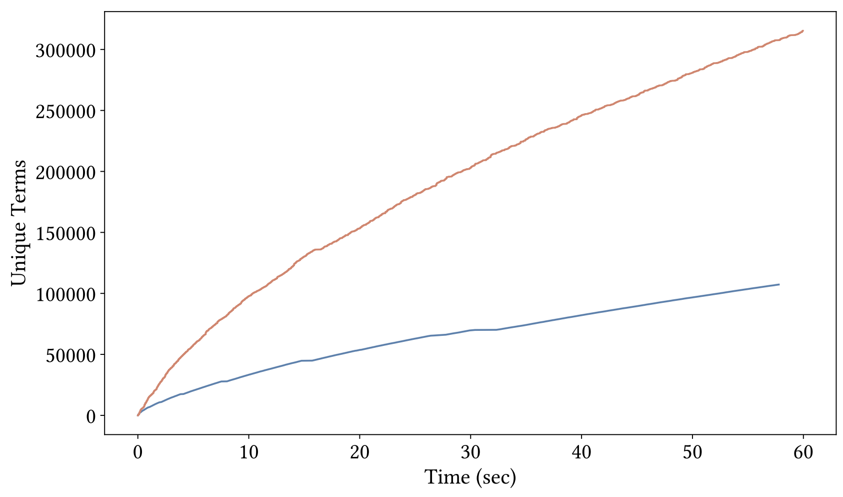
3.6.2 *Results*

We ran CGS and Rejection on each benchmark for one minute (on a MacBook Pro with an M1 processor and 16 GB RAM) and recorded the unique valid values produced. We counted unique values because duplicate tests are generally less useful than fresh ones (if the system under test is pure, duplicate tests add no value). The totals, averaged over 10 trials, are presented in Table 3.2.

Table 3.2: Unique valid values generated in 60 seconds (n = 10 trials). Standard deviation in parentheses.

	BST	**SORTED**	**AVL**	**STLC**
Rejection	7354 (109)	5768 (88)	129 (6)	70,127 (711)
CGS	22,107 (338)	59,677 (1634)	219 (2)	280,091 (7265)

These measurements show that CGS is always able to generate more unique values than Rejection in the same amount of time, often significantly more. The exception is the **AVL** benchmark; we discuss this below.



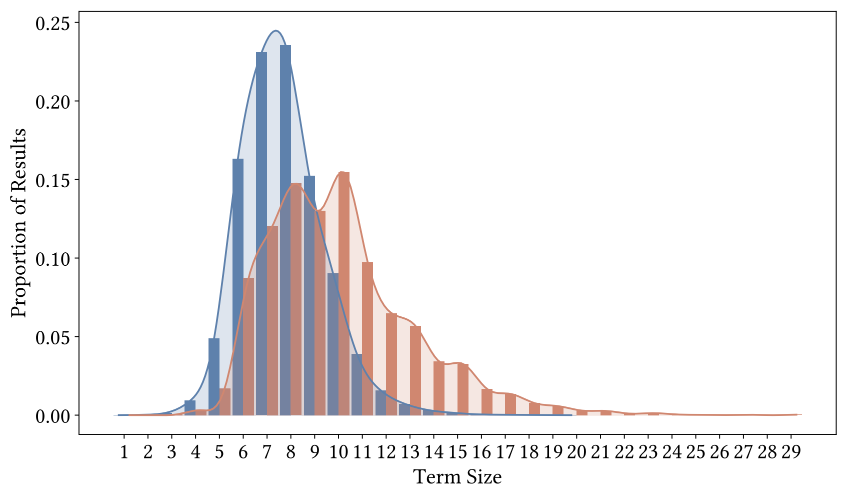


Figure 3.5: Unique values and term sizes for the **STLC** benchmark, averaged over values in a single trial.

Besides unique values, we measured some other metrics; the charts in Figure 3.5 show the results for the **STLC** benchmark. The first plot ("Unique Terms over Time") shows how CGS behaves over time. Not only does CGS find more unique terms than Rejection overall, but its lead continues to grow over time. Additionally, the "Normalized Size Distribution" chart shows the size distributions terms generated by both algorithms. The CGS distribution is skewed farther to the right, showing that it generates larger terms on average; this is good from the perspective of property-based testing, where test size is often positively correlated with bug-finding power, since larger test inputs tend to exercise more of the implementation code. Analogous charts for the remaining benchmarks can be found in the Appendix of the full paper.

3.6.3 *Measuring Diversity*

In the original paper, this section discussed the diversity of the inputs generated by CGS. We used Levenshtein distance [108] of choice sequences as a proxy for distance and argued that the average distances between CGS-generated values was larger. Upon reflection, I no longer like that evaluation; it made some weak arguments based on the shapes of confusing-looking graphs, rather than doing a thorough statistical analysis. Looking back at the data, I am comfortable arguing that CGS does not reduce diversity (the average distances do appear to be marginally higher for CGS) but the variance of the data is too high to say anything stronger.

3.6.4 *The Problem with **AVL**: Very Sparse Validity Conditions*

The **AVL** benchmark is an outlier in most of our measurements: CGS only manages to find a modest number of extra valid AVL trees. Understanding this phenomenon provides insight into a critical assumption underlying the CGS algorithm—namely that it is not too difficult to find valid values randomly.

It is clear that AVL trees *are* quite difficult to find randomly: balanced binary search trees are hard to generate on their own, and AVL trees are even more difficult because the generator must guess the correct height to cache at each node. This is why rejection sampling only finds 156 AVL trees in the time it takes to find 9762 binary search trees.

In domains like this, CGS is unlikely to find *any* valid trees while sampling. In particular, the check in line 15 of Figure 3.4 will often be true, meaning that choices will be made at random rather than guided by the fitness of the appropriate derivatives. We could reduce this effect by significantly increasing the sample rate constant N, but then sampling time would likely dominate generation time, resulting in worse performance overall.

The lesson here seems to be that the CGS algorithm does not work well with especially hard-to-satisfy validity conditions. In §3.9, we present an idea that would do some of the hard work ahead of time and help with this issue.

3.7 LIMITATIONS

Our free generator abstraction is extremely general and demonstrably useful, but a few technical weaknesses are worth discussing.

The biggest limitation has to do with the kinds of distributions our free generators can represent. Our exposition uses weighted choices (`frequency`) as the randomness primitive, but QuickCheck is technically built using a primitive like:

```
choose :: Random r ⇒ (r, r) → Gen r
```

Intuitively, `choose (x, y)` uniformly picks a value in the *range* from `x` to `y`, and this range can technically be infinite (e.g., if `r = Rational`). This cannot be replicated with `frequency` or `pick`. Thus, our results only apply to generators whose distributions are finitely supported.

Another small issue is that we have intentionally neglected one common element of monadic generators in the style of QuickCheck: size. Generators in standard QuickCheck track size bounds dynamically, allowing the testing framework to externally control the size distribution of the inputs that it generates. This does not impact our theoretical results (sizes can always be passed around manually, as we do in the examples in this paper), and sizes would be relatively easy to add to the free generator language in practice.

Finally, a note on the class of languages that free generators can parse (when interpreted with $\mathcal{P}[\![\cdot]\!]$). Free generators are limited in their nondeterminism (by the definition of `pick`, and by assumptions made in the definition of $\mathcal{P}[\![\cdot]\!]$); choices in a free generator are always unambiguous. This means that the parser interpretation of a free generator cannot parse arbitrary languages of choices, even though monadic parsers in general can parse arbitrary languages. Ultimately this is not a practical concern, as free generators parse sequences of choices, not realistic languages, but it is aesthetically disappointing. We believe it would be straightforward to add an operator for explicit nondeterminism and extend the interpretations accordingly.

3.8 RELATED WORK

We discuss a variety of publications that relate to the present work via connections either to free generators or to our Choice Gradient Sampling algorithm.

3.8.1 *Parsing and Generation*

The connection between parsers and generators has been employed implicitly in some generator implementations. Two popular property-based testing libraries, Hypothesis [113] and Crowbar [31], implement generators by parsing a stream of random choices. In fact, Hypothesis even takes advantage of parsing concepts when *shrinking* test inputs to make failing test-cases more readable for uses. However, neither of these frameworks has formalized the relationship between parsing and generation.

3.8.2 *Free Generators*

Garnock-Jones et al. present a formalism based on parsing expression grammars (PEGs) with some of the same goals as ours. They give a derivative-based algorithm that somewhat resembles CGS, which constructs sentences that match a particular PEG. Their work does not attempt to solve the valid generation problem for complex validity conditions like the ones we tackle, but it does provide further evidence that connecting parsing and generation is advantageous.

Claessen et al. [22] present a generator representation that is related to our free generator structure, but used in a very different way. They primarily use the syntactic structure of their generators (they call them "spaces") to control the size distribution of generated outputs; in particular, spaces do not make choice information explicit in the way free generators do. Claessen et al.'s generation approach uses Haskell's laziness, rather than derivatives and sampling, to prune unhelpful paths in the generation process. This pruning procedure performs well when validity conditions take advantage of laziness, but it is highly dependent on evaluation order and limited in its analysis of what makes a choice invalid. In contrast, CGS does not require that predicates be written in a specific way and has a much more nuanced notion of "unhelpful" choices.

3.8.3 *The Valid Generation Problem*

The valid generation problem is well studied. The most obvious solution existing solution is to write a bespoke generator. For example, the CSmith project famously developed a generator for valid C programs that was very successful at finding bugs in C compilers [179]. More generally, the domain-specific language for generators provided by the QuickCheck library [79] provides a whole framework for writing manual generators that produce valid inputs by construction. The primary issue with these manual approaches is effort: writing a bespoke generator is labor-intensive and difficult. The CGS algorithm aims to avoid manual techniques like this in the hopes of making property-based testing more accessible to programmers that do not have the time or expertise to write their own custom generators.

The constraint logic programming (CLP) generators proposed by Dewey [29] represent a different approach to valid generation, more automated than QuickCheck. Users of CLP generators have a constraint solver to help them, making it easier to express certain kinds of validity conditions in the generator. Even so, the CLP approach is not truly automatic: testers still need to express validity condition as annotated logic programs. Depending on the testers' background, this may be ideal, or it may be a deal-breaker. In contrast, CGS only requires that the validity condition be encoded as a Boolean predicate in the host programming language, which the tester may very well already have written for other reasons.

The Luck language [102] provides a similar semi-automatic solution; users are still required to put in some effort, but they are able to define generators and validity predicates at the same time. Again, this solution might be satisfying if users are starting from scratch and willing to learn a domain-specific language, but if validity predicates have already been written or users do not want to learn a new language, a more automated solution may be preferable.

When validity predicates are expressed as inductive relations, approaches like the one in *Generating Good Generators for Inductive Relations* [103] are extremely powerful. In the QuickChick framework, users can extract generators from the inductive relations that they likely already have for their proofs. This is incredibly convenient for testing lemmas that will eventually be proved, to establish confidence before attempting the proof. Unfortunately, the kinds of inductive relations that QuickChick depends on generally require dependent types to express, so this approach does not work in most mainstream programming languages.

Target [111] uses search strategies like hill climbing and simulated annealing to supplement random generation and significantly streamline property-based testing. Löscher and Sagonas's approach works well when inputs have a sensible notion of "utility," but in the case of valid generation the utility is often degenerate—0 if the input is invalid, and 1 if it is valid—with no good way to say if an input is "better" or "worse." In these cases, derivative-based searches may make more sense.

Some approaches use machine learning to automatically generate valid inputs. Learn&Fuzz [49] 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 the same vein, RLCheck [144] uses reinforcement learning to guide a generator to valid inputs. This approach served as early inspiration for our work, and we think that the theoretical advance of generator derivatives may lead improved learning algorithms in the future (see §3.9).

3.9 CONCLUSION

Free generators and their derivatives are powerful structures that give a flexible perspective on random generation. This formalism yields a useful algorithm for addressing the valid generation problem, and it clarifies the folklore that a generator is a parser of randomness. Moving forward, there are a number of paths to explore, some continuing our theoretical exploration and others looking towards algorithmic improvements.

3.9.1 *Bidirectional Free Generators*

We have only scratched the surface of what seems possible with free generators. One concrete next step is to merge the theory of free generators with the emerging theory of *ungenerators* [50]. This work describes generators that can be run both forward (to generate values as usual) and *backward*. In the backward direction, the program takes a value that the generator might have generated and "un-generates" it to give a sequence of choices that the generator might have made when generating that value.

Free generators are quite compatible with these ideas, and turning a free generator into a bidirectional generator that can both generate and ungenerate should be fairly straightforward. From there, we can build on the ideas in the ungenerators work and use the backward direction of the generator to learn a distribution of choices that approximates some user-provided samples of "desirable" values.

3.9.2 *Algorithmic Optimizations*

In §3.6.4, we saw some problems with the Choice Gradient Sampling algorithm: because CGS evaluates derivatives via sampling, it does poorly when validity conditions are very difficult to satisfy. This begs the question: might it be possible to evaluate the fitness of a derivative without naïvely sampling?

One potential approach involves staging the sampling process. Given a free generator with a depth parameter, we can first evaluate choices on generators for size 1, then evaluate choices for size 2, etc. These intermediate stages would make gradient sampling more successful at larger sizes, and might significantly improve the results on benchmarks like **AVL**. Unfortunately, this approach might perform poorly on benchmarks like **STLC** where the validity condition is not uniform: size-1 generators would avoid generating variables, leading larger generators to avoid variables as well. Nevertheless, this design space seems well worth exploring.

3.9.3 *Making Choices with Neural Networks*

Another algorithmic optimization is a bit farther afield: using recurrent neural networks (RNNs) to improve our generation procedure.

As Choice Gradient Sampling makes choices, it generates useful data about the frequencies with which choices should be made. Specifically, every iteration of the algorithm produces a pair of a history and a distribution over next choices that looks something like this:

$$\texttt{abcca} \mapsto \{\texttt{a} : 0.3, \texttt{b} : 0.7, \texttt{c} : 0.0\}$$

In the course of CGS, this information is used once (to make the next choice) and then forgotten—but what if there was a way to learn from it? Pairs like this could be used to train an RNN to make choices that are similar to the ones made by CGS.

There are details to work out, including network architecture, hyperparameters, etc., but in theory we could run CGS for a while, train an RNN, and after that point only use the RNN to generate valid data. Setting things up this way would recover some of the time that is currently spent sampling of derivative generators.

One could imagine a user writing a definition of a type and a predicate for that type, and then setting the model to train while they work on their algorithm. By the time the algorithm is finished and ready to test, the RNN model would be trained and ready to produce valid test inputs. A workflow like this might help increase adoption of property-based testing in industry.

CHAPTER 4

Reflecting on Randomness

This chapter addresses two research opportunities from Chapter 2, RO5, which calls for better technologies for test-case shrinkers, and RO4, which once again asks for better generation technologies. I describe an extension to free generators, *reflective generators* which can be run "backward" to extract information from generated values. Reflective generators give new algorithms for both shrinking and generation that are more generally applicable than previous results. The work in this chapter was done concurrently with the *Property-Based Testing in Practice* study, so, again, these technologies were not available to the developers we spoke with. I plan to explore the usability of reflective generators in real-world scenarios in the future. The content in this chapter is taken from *Reflecting on Random Generation* [58], which was published as a Distinguished Paper at the International Conference on Functional Programming (ICFP) 2023. An original version of the paper was written with equal contribution between myself and Samantha Frohlich at the University of Bristol, with revised versions rewritten primarily by me; Meng Wang (also at Bristol) and Benjamin C. Pierce advised the project and helped edit the papers.

4.1 INTRODUCTION

Property-based testing, popularized by Haskell's QuickCheck library [20], draws much of its bug-finding power from *generators* for random data. These programs are carefully crafted and encode important information about the system under test. In particular, QuickCheck generators like the one in Figure 4.1a capture what it means for a test input to be *valid*—here, ensuring that a tree satisfies the binary search tree (BST) invariant by keeping track of the minimum and maximum allowable values in each subtree. This generator is not just a program for generating BSTs, it *defines* BSTs in the sense that its range is precisely the set of valid BSTs.

Developers of tools like Hypothesis [113]—arguably the most popular PBT framework, with 6,500 stars on GitHub and an estimated 500,000 users [30]—capitalize on this observation and repurpose generators for other tasks, including test-case *shrinking* and *mutation.* These algorithms do not operate directly on

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

(a) QuickCheck generator.

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

(b) Reflective generator.

Figure 4.1: Generators for binary search trees.

data values; rather, shrinking or mutating a value is accomplished by shrinking or mutating the *random choices* that produced that value, and then re-running the generator on the modified choices [112]. This amounts to treating generators as *parsers*, taking unstructured randomness and parsing it into structured values—a perspective formalized in terms of *free generators* [55]. Viewing generators as parsers has two advantages: (1) shrinking and mutation can be implemented generically, rather than once per generator, and (2) the modified data values will automatically satisfy any preconditions that the generator was designed to enforce (e.g., the BST invariant), since they are ultimately produced by pushing modified choices through the generator.

Ideally, Hypothesis's type-agnostic, validity-preserving algorithms should completely subsume more manual ones. Unfortunately, the current Hypothesis approach assumes that the shrinker, for example, is given access to the original random choices that the generator made when producing the value it is shrinking; this doesn't work when shrinking is separated (in time or space) from generation. In particular, Hypothesis can't shrink values that it did not generate in the first place—e.g., because they were provided as pathological examples or from crash reports. More subtly, Hypothesis's shrinking also breaks if the value was modified between generation and shrinking, or saved without also saving a record of the choices.

To use Hypothesis-style shrinking on an arbitrary value, the shrinker needs some way of reconstituting a set of random choices that produce that value. Luckily, inspiration for how to do this can be drawn from the grammar-based testing literature, specifically *Inputs from Hell* [155], which describes a way to produce test inputs that are similar to an existing one. Starting with a grammar-based generator [48], they first use the grammar to parse a value, determining which *productions* must be expanded to produce that value. Then, they bias the generator to expand those productions more often, thus resulting in more values that are similar to the example. In essence, this approach determines which generator choices lead to a desired value by *going backward*, parsing the value with the same grammar that (could have) generated it.

The *Inputs from Hell* approach works well for grammar-based generators, but does not apply to generators that enforce validity. For this, we need a more general solution—one that works with the kinds of *monadic* generators used in QuickCheck, which can enforce arbitrary validity conditions. Such a solution can be found in the bidirectional programming literature. Xia et al. [177] describe *partial monadic profunctors*, which enrich standard monads with extra operations for describing bidirectional computations. This infrastructure, along with the parsing-as-generation perspective of free generators, enables exactly the kind of bidirectional generation needed to extract random choices from a monadically produced value.

Our main contribution, *reflective generators*, is a language for writing bidirectional generators that can "reflect" on a value to analyze which choices produce that value (see Figure 4.1b). They subsume the grammar-based generators of *Inputs from Hell*, and, critically, they enable Hypothesis-style shrinking and mutation for arbitrary values in the range of a monadic generator. Furthermore, since reflective generators are built on *freer monads*, they can be interpreted numerous ways besides generation and reflection. We discuss three more use cases that demonstrate the versatility of reflective generators as testing utilities.

Following a brief tour through some background (§4.2), we offer the following contributions:

- We present *reflective generators*, a framework that fuses *free generators* and *partial monadic profunctors* into a flexible domain-specific language for PBT generators that can reflect on a value to obtain choices that produce it. (§4.3)
- We develop the theory of reflective generators, defining what it means for reflective generators to be correct along a number of axes and comparing their expressive power to other generation abstractions. (§4.4)
- We demonstrate the core behavior of reflective generators by generalizing prior work on example-based generation. Our implementation subsumes the *Inputs from Hell* algorithm and extends it to work with monadic generators. (§4.5)

- We apply reflective generators to manipulate user-provided values. Reflective generators enable generator-based shrinking and mutation algorithms to work even when the randomness used to generate test inputs is not available. We show that our shrinkers are at least as effective as other automated shrinking techniques. (§4.6)
- We leverage the reflective generator abstraction to implement other testing tools: checkers for test input validity, "completers" that can randomly complete a partially defined value, and test input producers like enumerators and fuzzers. (§4.7)

We conclude by discussing related work (§4.8) and future directions (§4.9).

4.2 BACKGROUND

The abstractions we present in this paper rely on a significant amount of prior work. In this section, we review the structures that make reflective generators possible: monadic random generators (§4.2.1), free generators (§4.2.2), and partial monadic profunctors (§4.2.3).

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

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

type Gen a = Int -> a
```

(a) Definitions for monadic generators.

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

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

(b) Definitions for free generators.

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

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

class (Monad (p b), PartialP p)
  => PMP p
```

(c) Definitions for partial monadic profunctors.

Figure 4.2: Background definitions.

4.2.1 *Monadic Random Generators*

The idea of testing executable properties using monadic generators was popularized by QuickCheck and has endured for more than two decades. The core structures used by QuickCheck are shown in Figure 4.2a, where `Gen` represents the type of random generators. It treats the input `Int` as a random seed and uses it to produce a value of the appropriate type. Generators like the one in Figure 4.1a are easy to write because they are *monads* [125]; this structure provides a neat interface for chaining effectful computations.

4.2.2 *Free Generators*

Interfaces like the `Monad` type class can be reified into "free" structures that represent each operation as a data constructor, allowing it to be interpreted in multiple ways. There are several such free structures for the monad interface [95; 164, etc.]; we focus on the *freer monad* [94] structure shown in Figure 4.2b, which reifies the `return` operation as the constructor `Return` and `(>>=)` as `Bind`. The extra type constructor `f` ranges over the operations that are specific to each given monad, for example, the `Get :: State s s` and `Put :: s -> State s ()` operations for `State`.

Our prior work, *free generators* [55], are an instance of this scheme, instantiating `f` as the type constructor `Pick` (Figure 4.2b), which represents a choice between sub-generators. `Pick` is used to implement familiar QuickCheck-style combinators like `choose`, which generates an integer in a given range, and `oneof`, which randomly selects between a list of generators. We use free generators to draw a formal connection between random generation and parsing, interpreting the same free generator as both a generator and a parser.

4.2.3 *Partial Monadic Profunctors*

Profunctors are a standard construction from category theory, generalizing ordinary functors—structure-preserving maps—to allow for both covariant and contravariant mapping operations. They are realized as the `Profunctor` class (Figure 4.2c), popularized in Haskell by Pickering et al. [140], where the mapping operations are called `rmap` and `lmap` respectively. The quintessential example of a profunctor is the function type constructor `(->)`. This makes sense, since `b -> a` is contravariant in `b` and covariant in `a`.

In that case, `rmap` implements post-composition (of some function `a -> a'`) and `lmap` implements pre-composition (of a function `b' -> b`). Indeed, it is often useful to think of profunctors as function-like: a value of type `p b a` "examines a value of type `b` to produce a value of type `a`" (potentially doing some other effects).

A *monadic profunctor* is a profunctor that is also a monad (i.e., a profunctor `p` such that for any `b`, `p b` is a monad). Xia et al. [177] use this extra structure to implement composable bidirectional computations. For example, consider the classic bidirectional programming example of a parser and a pretty printer, which invert one-another. These dual functions can be implemented using the same monadic profunctor:

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

This `Biparser` is a single program that gets "interpreted" in two different ways. As a parser, it ignores the `b` parameter entirely, and simply acts as a parser in the style of Parsec [106] to produce a value of type `a`. As a pretty printer, it still needs to produce a value of type `a`—after all, the two interpretations share the same code—but now there is no input `String` to parse. Instead, the pretty printer interpretation has a value of type `b` it uses as instructions to produce both the `a` and a `String`. This scheme makes much more sense when the profunctor is *aligned*, that is of type `Biparser a a`; in that case, a properly written pretty printer acts as an identity function, taking a value and reproducing it while also recording its `String` representation.

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

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

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

The problem is the first argument: we have a value of type `p a a` but we need one of type `p b a` since `p b` is the type of the monad. Luckily, the `lmap` operation (§4.2c) makes it possible to take an aligned profunctor, `p a a` and turn it into a `p b a` by providing an *annotation* of type `b -> a` that says how to focus on some part of a value of type `b` that has type `a`. We can thus build up a `Biparser` by writing a program that looks essentially like a standard monadic parser, but with monadic binds annotated with calls to `lmap` that fix the alignment.

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

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

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

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

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

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

A concrete example of a partial profunctor is the partial arrow constructor:

```
newtype PartialArr a b = PartialArr {unPA :: a -> Maybe b}
```

This follows on naturally from the intuition that `(->)` is a profunctor. In fact, these are also PMPs, you can find the relevant instances in the full paper's Appendix.

PMPs are complex, and they can be used in a wide variety of ways. We recommend *Composing Bidirectional Programs Monadically* [177] for a more thorough explanation.

4.3 THE REFLECTIVE GENERATOR LANGUAGE

Reflective generators combine free generators with PMPs, enabling a host of enhanced testing algorithms. In this section, we explain the intuition behind reflective generators (§4.3.1), describe their implementation (§4.3.2), and discuss their various interpretations (§4.3.3).

The basic structure of a reflective generator comes from adding the partial monadic profunctor operations, `lmap` and `prune`, to the `Pick` datatype. We call this extended type `R` (for Reflective) and implement it like this:

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

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

The `Pick` constructor here has two small changes from the free generator presentation: we add an extra contravariant type variable `b`, and we modify the `choice` type to optionally elide choice labels.[4] Then `Lmap` captures contravariant annotations (there is no need to explicitly represent `rmap`, as we will be able to encode it using the monad structure), and `Prune` represents its PMP counterpart with an analogous type. A reflective generator is then a freer monad over `R b`:

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

4.3.1 *Intuition*

The type `Reflective b a` of reflective generators should be understood to mean a program that can "reflect on a value `b`, recording choices, while generating an `a`."

[4]As in *Parsing Randomness*, we represent weights in `Pick` as integers for simplicity. Formally they are required to be strictly positive; this will be necessary to prove Theorem 3.

Like PMPs, reflective generators use annotations to fix the types around monadic binds. More intuition of how these should be defined follows from the intuitive interpretation of the types: the goal is to take a generator that reflects on choices in an `a` and turn it into one that reflects on choices in a `b`. Here's the key: it suffices to show how to *focus on a part of the* `b` *that contains an* `a`, because that focusing turns `a` choices into `b` choices. Put another way, the annotation should embed a mapping of type `b -> a` or `b -> Maybe a` that focuses on the `a` part of the `b`. To see this in action, consider the example in Figure 4.1b, paying attention to the first bind in the `Node` branch,

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

where `...` continues on to produce the rest of the tree. The call to `choose` results in a `Reflective Int Int`, but the type of the enclosing monad is `Reflective Tree`; as discussed in §4.2.3 we need to add an annotation on the bind that focuses on an `Int` in a `Tree` to get a `Reflective Tree Int`. In the example, we annotate with `focus (_Node._2)` (this syntax is introduced in the next section) but the following is equivalent:

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

As with PMPs like `Biparser`, the process of reflecting on choices is all about the interaction between binds and `comap`s. A value flows through the program, and, at each bind, the `comap` focuses on the part of the value that the left side of the bind should reflect on. If the focusing fails, that branch gets pruned—there is no way to produce the desired value—but if it succeeds, then the left side can reflect on that part of the value, extracting some choices, and reflection can continue on the right side.

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

4.3.2 *Implementation*

We next describe how reflective generators are implemented and what we've done to make them feel familiar and easy to work with. (In §4.9, we discuss plans to validate this belief with a proper user study.)

The Full Story The actual type `R` defined in the Haskell artifact [53] is a bit more complicated than the one explained above. The actual implementation looks like:

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

First, we add a constructor `ChooseInteger` for picking integers from a range. Technically, this is implementable via `Pick` by simply enumerating the integers in the desired range, but doing so is inefficient if the range is large. Adding a separate function for choosing within a range of integers allows us to bootstrap other generators over large ranges, and it makes it easy to implement much more efficient interpretation functions later on.

Second, we add two constructors, `GetSize` and `Resize`, that are analogous to similar operations implemented by QuickCheck. Maintaining size control is critical for ensuring generator termination, and, although it is possible to implement sized generators by passing size parameters around manually, internalizing size control cleans up the API of the combinator library and makes generators more readable.

In future sections, we often elide parts of definitions pertaining to these operations, to streamline the presentation, but they do present a couple of theoretical complications that we note in §4.4.

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

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

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

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

A brief aside on choice labels: whether or not a user decides to label the choices in a generator depends on two factors. First, it depends on how the generators will be used. In §4.5.1, we discuss a use case that relies heavily on choice labels, and in §4.6 we discuss one that ignores them. Second, whether or not a particular sub-generator is labeled can impact the behavior of use cases that pay attention to labels; in §4.5.1 we discuss intentionally eliding labels as a way of marking parts of the generator whose distributions should not be tuned. As a general rule, we recommend labeling choices in generators, and all generators provided by the reflective generators library are labeled by default, but it is convenient to be able to elide labels when upgrading from a QuickCheck generator.

Choice operators alone are not enough to build a reflective generator, we need infrastructure to glue them together. The bulk of these glue operations follow from the fact that reflective generators are, as expected, PMPs:[5]

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

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

Both `lmap` and `prune` commute over `Bind` and do nothing to a `Return` (see the laws in §4.4), so these implementations are straightforward. Behind the scenes, the `Functor`, `Applicative`, and `Monad` operations are implemented for free from the freer monad.

Using `lmap` and `prune` on their own is a bit tedious, so we give two combinators that make common use cases much simpler. The `focus` combinator makes it possible to replace pattern matches in `lmap` annotations with *lenses* [40].

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

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

The curious reader can dig into the gory details of the types involved [96], but it suffices to understand `focus` as a notational convenience that gives a terse syntax for pattern matches:

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

This call to `focus` identifies the part of the value that `g` produces: the second argument to the `Node` constructor. The neat thing about this setup is that `_Node`, a lens that "pattern matches" on a node, can be automatically generated from the `Tree` datatype, and `_2`, which extracts the second element of a k-tuple, is included in the lens library. Together they can match on and extract the part of the value that the reflective generator needs to focus on.

Another convenient helper built from `lmap` and `prune` is `exact`, which operates like `return` but ensures that the returned value is exactly the expected one:

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

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

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

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

The `Void` type in Haskell is uninhabited, and thus a reflective generator of type `Reflective Void a` can only be used in limited cases, but `voidAnn` makes it possible to perform the upgrade from Figure 4.1 in stages.

The `Void` version looks like this:

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

First the user goes from Figure 4.1a to this partially converted generator, and then they can convert to Figure 4.1b by replacing `Void` with the correct output type, replacing `voidAnn` annotations with ones that do appropriate focusing, and replacing `return` with `exact` where appropriate. All three generators are shown together in the original paper's Appendix. Experienced reflective generator writers do the upgrade in a single step, but when starting out it may be easier to go through a simpler intermediate generator.

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

4.3.3 *Interpretation*

Like free generators, reflective generators do not do anything interesting until they are *interpreted*. An interpretation describes how the inert syntax of the generator program should be executed. As with PMPs, most reflective generator interpretations can be thought of as working either "forward," simply producing an output of their covariant type, or "backward," reflecting on a value (and reproduce it *en passant*) while tracking choices. Unlike PMPs, reflective generators do not explicitly pair a forward and a backward interpretation together—in fact, the interpretation in §4.7.2 actually uses both directions at once. Still, directionality of interpretations is often a useful intuition.

The simplest "forward" interpretation turns a reflective generator into a standard QuickCheck generator, shown in Figure 4.3 The free monad part of the syntax is implemented as expected, with `Return` implemented as `return` in the `Gen` monad, and `Bind` as the monad's bind. The rest of the syntax is similarly straightforward, with `Lmap` and `Prune` doing nothing and `Pick` interpreted as a weighted random choice.

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

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

Figure 4.3: The “generate” interpretation.

Of course, the value of reflective generators lies in their ability to run “backward,” focusing on subparts of a value and reflecting on how they are constructed. This process can be seen using the `reflect` function in Figure 4.4, which interprets a generator to determine which choices could lead to a given value. For example, it would behave as follows when run on `bst` (adapted to record choices):

```
ghci> reflect bst Leaf
⟦"leaf"⟧
ghci> reflect bst (Node Leaf 4 Leaf)
⟦"node","4","leaf","leaf"⟧
```

Here, the constructor choice is indicated with `"leaf"` or `"node"`, and the number choice by printing the number. A list of lists is produced so that all possible choice traces are covered. (In these examples, there just happens to be only one trace of choices that could have led to the provided values.)

When interpreting `Pick` the computation splits, each branch representing making one particular choice. In each branch, `Lmap` nodes focus on parts of the value being reflected on; if the focusing fails, a following `Prune` node will filter that branch out of the computation. The monad structure, `Return` and `Bind`, threads the list of recorded choices through the computation, so the final result is a list of the different branches of the computation that were not pruned, along with the choices made in each of those branches. For termination, the `reflect` interpretation requires that any recursion is guarded by `focus` annotations that focus on smaller parts of the value being reflected on; see §4.4.2 for more details.

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

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

Figure 4.4: The “reflect” interpretation.

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

4.4 THEORY OF REFLECTIVE GENERATORS

In this section, we describe more of the theory underlying reflective generators. We discuss various formulations of correctness, including defining what it means to correctly interpret a reflective generator and what it means to correctly write an individual reflective generator (§4.4.1). Next, we explore an interesting property of reflective generators—*overlap*—which has implications for generator efficiency (§4.4.2). Finally, we discuss the expressive power of reflective generators, comparing it to grammar-based generators and to standard monadic ones (§4.4.3).

4.4.1 *Correctness*

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

Correctness of Interpretations Reflective generators should obey the laws of monads [125], profunctors, and PMPs:

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

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

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

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

Lemma 2. *Reflective generators always obey* (*M*1)*,* (*M*3)*,* (*PMP*3)*, and* (*PMP*4)*.*

Proof. By induction on the structure of the generator, using the definitions of `return` and `(>>=)` from Kiselyov and Ishii [94] and the definitions of `lmap` and `prune` from §4.3.2. See the Appendix. □

The other equations do not hold in general: they must be established for each interpretation.

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

Definition 1. An interpretation

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

is *lawful* iff p obeys the laws of monads, profunctors, and partial monadic profunctors and there exists an *R-interpretation*

```
⟦·⟧R :: R b a -> p b a
```

such that the following equations hold:

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

An alternative approach would be to simply define an interpretation of a reflective generator as a PMP homomorphism along with an interpretation for `Pick`, rather than giving the programmer the freedom to implement lawless interpretations. From a programming perspective, this would behave like a tagless-final embedding [93]. We found this tagless-final approach more tedious to program with, but it is available to users if desired (see the Appendix).

The `generate` instance is indeed lawful, modulo one technical caveat. The classic `Gen` "monad" itself is not actually a lawful monad, but it *is* lawful up to distributional equivalence [20]—i.e., generators that produce equivalent probability distributions of values are equivalent, even if they are not equal as Haskell terms. The same caveat applies to the other laws.

Theorem 3. *The* `generate` *interpretation is lawful up to distributional equivalence.*

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

The `reflect` interpretation is also lawful, ignoring the `map snd` projection that discards the accumulator values.

Theorem 4. *The* `reflect` *interpretation is lawful.*

Proof. We choose

```
p b a = b -> [(a, [String])] = ReaderT b (WriterT [String] []) a
```

which is simply a stack of three monads (reader, writer, and list). Lawfulness follows straightforwardly, by aligning the definition of `reflect` with the lawful implementations of the monad, profunctor, and partial profunctor operations for this combined monad. □

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

Our correctness criteria are based on similar notions to those of Xia et al. [177]. We formulate correctness using two interpretations. The `generate` interpretation is the canonical "forward" interpretation, characterizing the set of values that can be produced by a reflective generator when it ignores its contravariant parameter. The canonical "backward" interpretation should characterize the generator's operation as a generalized identity function, taking a value and reproducing it—`reflect` is almost the right interpretation, but it does extra work to keep track of choices. Thus, we define:

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

The `reflect'` interpretation has the same behavior as `reflect`, but it skips the code that tracks choices. It also has a more general, unaligned type. (Alignment is an artificial restriction anyway; `reflect` could also be given an unaligned type, but the alignment better-communicates the intended use.) The full code for this and all other interpretation functions can be found in our artifact.

We define soundness and completeness of a reflective generator as follows:

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

```
a ~ generate g ==> (not . null) (reflect' g a).
```
[6]

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

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

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

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

for all `n` in a reasonable range.

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

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

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

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

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

As there is no way to check `a ~ generate g` directly, completeness is impossible to test. Luckily, Xia et al. give an alternative. First they define *weak completeness*:

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

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

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

Lemma 3. *All reflective generators are weak complete.*

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

Xia et al. also gives a so-called *pure projection property*, which is testable (albeit sometimes intractably[7]):

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

```
a' ∈ reflect' g a ==> a = a'.
```

To complete the picture, we prove the following:

[7]The precondition of this property is often difficult to satisfy, leading to discards, but since needs only be tested once for given generator, the user can likely afford to spend time trying to falsify it.

Theorem 5. *Any weak-complete reflective generator satisfying pure projection is complete.*

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

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

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

We define the following properties:

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

`x ∈ gen g ==> p x`.

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

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

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

4.4.2 *Overlap*

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

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

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

```
g1 = labelled [ ("Z", exact Z), ("S", S <$> (focus _S g1)) ]

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

Figure 4.5: Reflective generators with unit, exponential, and infinite overlap.

But overlap can cause problems for backward interpretations that care about examining all ways of producing a particular value (e.g., `probabilityOf` which we will define in §4.7). In these cases, overlap may lead to exponential blowup or even non-termination. For example, consider the three generators in Figure 4.5. The first, `g1`, generates natural numbers, each in exactly one way:

```
ghci> reflect g1 (S (S (S (S (S Z))))) -- 5
⟦"S","S","S","S","S","Z"⟧
```

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

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

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

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

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

4.4.3 *Expressiveness*

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

Grammar-Based and Monadic Generators Grammar-based generators [48] use a context-free grammar describing the program's input format as the basis for generating test inputs. For example, the following grammar fragment defines a generator of expression parse trees:

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

Read as a generator, this says "to generate a `term`, choose either a `factor`, a `"*"` node, or a `"/"` node." Grammar-based generators are useful for generating inputs to a program with a context-free input structure, like expression evaluators, JSON minifiers, or even some compilers. They are often used in fuzzing, which we will discuss more in §4.6.3, for this reason. But grammar-based generators cannot ensure that the values they generate satisfy context-sensitive constraints. One might, for example, want to ensure that the left-hand side of a division does not evaluate to zero:

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

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

Enter monadic generators. As described in §4.2.1, monadic generators were introduced with QuickCheck and are a domain-specific language for writing generators that produce values satisfying arbitrary computable constraints. Monadic generators can generate binary search trees [81], well-typed terms in a simply-typed lambda calculus, and more. Monadic generators subsume context-free generators, for example, the following generator subsumes the term generator above:

```
term = oneof [Factor <$> factor, liftM2 Mul term factor, liftM2 Div term factor]
```

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

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

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

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

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

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

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

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

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

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

In fact, reflective generators can implement all of the examples we previously listed as the motivation for monadic generators (see the binary search tree generator in Figure 4.1b and the STLC generator fragment below). There are also a variety of examples in §4.5.1 and §4.6 that are expressible by monadic generators and not context-free ones.

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

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

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

What Reflective Generators Can't Do Given that reflective generators seem to be able to express so much, it may be easier to characterize what they *can't* express. The biggest limitation of reflective generators is that they cannot represent any approach to generation that fundamentally loses information about previously generated data. For example, in the generator

```
lmap ??? g >>= \ _ -> g'
```

which is missing an annotation, there is no valid annotation to write: the value generated by `g` cannot be "focused" on as part of the final structure. Why might this come up? One case is when the generator generates a value and then computes some non-invertible function on it; there would be no way to recover the original value to analyze the choices made when producing that value.

We have run across very few generators that fundamentally require a non-invertible function, but one interesting examples is some formulations of System F [147; 43]. It is possible (though challenging) to write a monadic generator for System F [134], but impossible to do so for reflective generators. The problem can be seen when referring back to the reflective generator for STLC terms, which uses type inference to recover a type from an expression. If we tried to translate this generator to one for System F, we would have a problem: type inference for System F is undecidable! Thus, the generator may fail to run backward, even if it works correctly when run forward. Of course, in practice one could write a reflective generator for System F terms which would work modulo some time-outs, but this is a neat example of the dividing line between monadic and reflective generators.

System F is a rare example of a fundamental limitation of reflective generators, but there are a few common generation idioms that reflective generators need to work around. First, reflective generators cannot use the QuickCheck combinator "`suchThat`", which samples a value and then, if it does not satisfy a some predicate, throws it away, *increases the size parameter*, and samples another. The size manipulation is the problem here: to correctly reflect on a value, the backward direction would need to keep trying generators, recording and throwing away choices, until one succeeds; as far as we know this is not possible with the current structure. The solution is simply to avoid `suchThat` in favor of generators that satisfy predicates constructively—this would be our recommendation anyway, since `suchThat` can be extremely slow in complex cases. Second, reflective generators do not support a relatively common idiom where generators pick an integer and then use that integer to bias the generation distribution in some way. This is problematic because there is no way to recover that integer in the backward direction. One could theoretically encode this pattern via `pick`: instead of `do { i <- choose (0, n); k i }`, write a `pick` with `n` equally-weighted branches that each call `k` on a different value of `i`. But this is inefficient, so, in practice, a larger rewrite might be required to get the desired distribution of values.

Bottom line: reflective generators have some ergonomic limitations, but they are almost as powerful as monadic generators in practice.

4.5 EXAMPLE-BASED GENERATION

In this section, we demonstrate the power of reflective generators in the context of a clever generation technique that was an early inspiration for their design: example-based generation (§4.5.1). We replicate and generalize the prior work using reflective generators (§4.5.2).

4.5.1 Inputs from Hell

Inputs from Hell [155] (IFH) describes an approach to random testing that starts with a set of user-provided example test inputs and randomly produces values that are either quite similar to or quite different from those examples—the idea being that similar values represent "common" inputs and that different ones represent interestingly "uncommon" inputs. By drawing test cases from both of these classes, IFH is able to find bugs in realistic programs.

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

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

and the example `12`, the IFH technique might derive the following pCFGs:

```
-- common
num -> 33% "" | 66% digit num
digit -> 50% "1" | 50% "2" | 0% "3"

-- uncommon
num -> 66% "" | 33% digit num
digit -> 0%  "1" |  0% "2" | 100% "3"
```

Each production is given a weight based on the number of times it appears in the parse tree for the provided example (more or less weight, depending on whether the goal is to generate common or uncommon inputs). The first grammar puts more weight on the `1` and `2`, since it is trying to generate more inputs like the initial example, whereas the second puts more weight on `3` because it is trying to generate inputs *unlike* `12`.

4.5.2 *Reflecting on Examples*

This process—parse the input, analyze the parse tree, and re-weight the grammar—is straightforward to implement in the setting of grammar-based generation, but the IFH work does not extend to monadic generators. As we discuss in §4.4.3, this is a significant limitation. Reflective generators can bridge this gap by recapitulating the ideas in IFH but using a reflective generator for IFH's parsing and generation steps.

Implementation We define three functions, corresponding to the parsing, analysis, and re-weighting operations for grammars in the IFH paper:

```
reflect         :: Reflective a a -> a -> ⟦String⟧
analyzeWeights :: ⟦String⟧ -> Weights
genWithWeights :: Reflective b a -> Weights -> Gen a
```

We have already seen `reflect`: it reflects on a generated value and produces lists of choices that were made to produce that value. With the choice sequences in hand, `analyzeWeights` aggregates the choices together to produce a set of weights that say how often to expand one rule versus another. This allows a new interpretation, `genWithWeights`, to generate new values by making choices with the weights calculated from the user-provided examples.

When a reflective generator looks like a grammar (as with `term` in §4.4.3), this process replicates the IFH algorithm exactly. But we have seen that reflective generators are far more powerful than grammar-based generators, so the new algorithm both replicates *and generalizes* IFH, enabling example-based generation for a much larger class of generators.

These interpretations rely heavily on choice labels, which we discuss briefly in §4.3.2. These labels are used by `reflect` and `genWithWeights` to track the choices that should be weighted higher or lower based on the examples. This means that, rather than building reflective generators with `oneof` or `frequency`, the programmer should use `labeled` or `pick`. Recall that the reflective generators library provides base generators that are labeled as well. However, there is some flexibility here: if the programmer would prefer some choices *not* be re-weighted based on examples, they can simply elide the labels. This is another way that the reflective generators approach generalizes IFH.

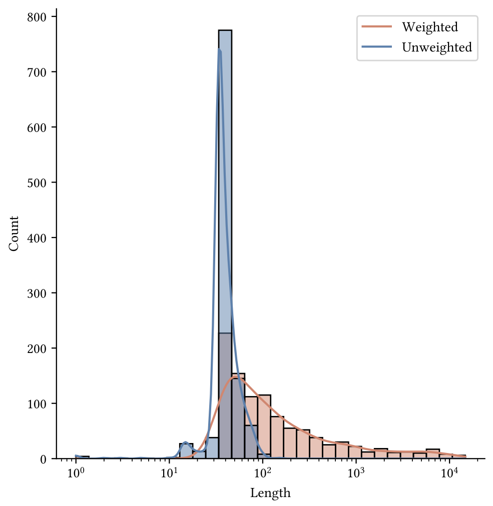
Example-Based Generation in Action Soremekun et al. [155] use the IFH tuning algorithm as part of a comprehensive testing regime; by contrast, we have found them to be most useful as a quick way to tune a generator to yield a reasonable distribution of sizes and shapes.

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

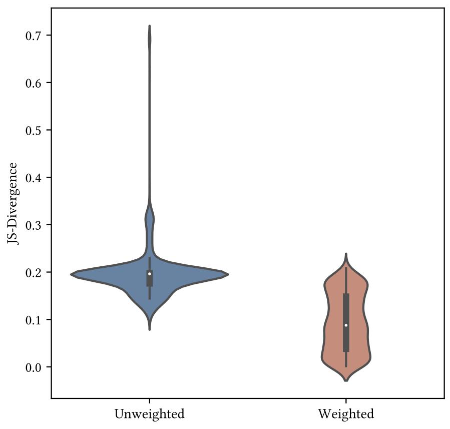
```
withHashCode :: Reflective String String
```

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

To demonstrate how these also achieve the goal of IFH-style generators (that the weighted generator is preferable to its unweighted counterpart), we sampled 10 JSON documents that were used in the IFH experiments, ranging from ~200-1,200 bytes long, and used them to weight `withHashCode` in the style of IFH. We generated 1,000 documents from that weighted generator, as well as from the unweighted generator, and compare the results in Figure 4.6.



(a) Length distributions of unweighted and weighted.



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

Figure 4.6: Analysis of `withChecksum` tuned by example in the style of *Inputs from Hell*.

Figure 4.6a demonstrates that the weighted generator is far preferable to the unweighted version in terms of its length distribution. The unweighted distribution, shown in orange, is skewed to the left (smaller values) and has a huge spike. Inspecting the data revealed that the payloads of these values are all either `{}` or `[]`, both relatively uninteresting and certainly not worth generating hundreds of times! In contrast, the weighted generator has a varied length distribution. It generates very few trivial values, instead producing a wide distribution that covers more of the input space.

Figure 4.6b focuses on the samples' *character distributions*. We counted the occurrences of each character across all 10 of the example documents, resulting in a probability distribution over characters. Then, for each sample, we computed the Jensen-Shannon divergence[8] [110], between the example distribution and the character distribution of the sample and plotted those divergences in a violin plot. JS divergence measures the difference between two probability distributions, so it is a simple way of getting a sense of how similar or different the characters in the samples are from the ones in the examples. The plots show that the unweighted samples are farther from the example distribution than the weighted samples.

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

4.6 VALIDITY-PRESERVING SHRINKING AND MUTATION

The example-based generation in the previous section illustrates some benefits of reflecting on choices. In this section, we explore those benefits further, using them to implement test input manipulation algorithms like shrinking and mutation. In this section, we discuss the "internal test-case reduction" algorithms implemented in the Hypothesis framework for PBT (§4.6.1), show that reflective generators make these algorithms much more flexible (§4.6.2), and finally sketch the ways that these ideas also apply to test-case mutation (§4.6.3).

4.6.1 *Test-Case Reduction in Hypothesis*

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

[8]Jensen-Shannon divergence is closely related to the more common Kullback-Leiber (KL) divergence [101], but it works better for distributions with differing support because its value is never infinite.

In QuickCheck, users can either use GHC's Generics [115] to derive a shrinker automatically for a given type, or they can write a shrinker by hand. The former is effective in simple cases, as we will see below, but it is not very general—these automatic shrinkers only know about the type structure, so they cannot ensure that the shrunken values satisfy important preconditions nor adequately shrink less structured data like strings. The latter is totally general, but many users find writing shrinkers by hand confusing and error-prone.

This unsatisfying situation led MacIver and Donaldson [112] to design Hypothesis's "internal test-case reduction," which gives the best of both worlds. It solves the generality issue without requiring user effort or understanding. The key insight is that the generator itself already has the information needed to produce precondition-satisfying inputs, so the generator should be used as part of the shrinking process. The accompanying clever trick is to *shrink the random choices* used to generate a value, rather than shrinking the value itself.

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

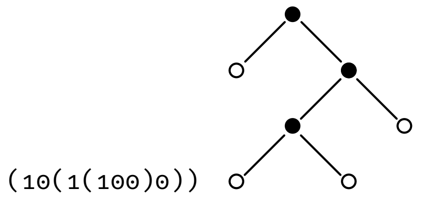


Figure 4.7: A Hypothesis-inspired reflective generator and a tree that the generator might produce.

Concretely, Hypothesis represents its input randomness as a bracketed string of bits. For example `(10(1(100)0))` produces the tree in Figure 4.7. The first bit says to expand the top-level node, the second says that the left-hand subtree is a leaf, and so on. Each level of bracketing delineates some choices that are logically nested together (in this case, on a particular level of the tree). Hypothesis aims to shrink these bitstrings by finding the *shortlex minimum* string of choices that results in a valid counterexample; shortlex order considers shorter strings to be less than longer strings, and follows lexicographic ordering otherwise (brackets are ignored for the purpose of ordering). In practice, shortlex order turns out to be an effective proxy for complexity of generated test cases: smaller bitstrings tend to produce smaller test cases.

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

4.6.2 *Reflective Shrinking*

The downside of the Hypothesis approach is that this style of shrinking only works if the random bitstring that produced the target value is still available—without it, there is nothing to shrink. But there are many reasons one might want to shrink a value for which one does *not* have a corresponding bitstring. In particular, shrinking can be useful for understanding externally provided values that were not produced by the generator at all; for example, if a user submits a bug report containing a printout of some large input that caused a crash, it might be much easier to debug the problem with the help of a shrinker. Similarly, internal shrinking does not work if the value has been modified at all between generation and shrinking, as might be desirable when doing fuzzing-style (see §4.6.3) testing where test inputs are mutated to explore values in a particular region. Luckily, reflective generators can help.

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

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

The `Choices` type describes rose trees with two types of nodes: `Choice`, which represents a single-bit choice, and `Draw` which represents some nested sequence of choices;[9] this type is isomorphic to the bracketed choice sequences that Hypothesis uses. The `choices` function takes a reflective generator and a value and produces the choice sequences that result in generating that value.

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

[9] In the Haskell artifact, we use a slightly more complicated type, caching size information to make shortlex comparisons faster.

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

Shrinking Strategies With an appropriate bracketed choice sequence in hand, shrinking can begin. We implemented a representative subset of the shrinking passes described in the Hypothesis paper. The `subTrees` pass tries shrinking to every available child sequence of the original:

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

The `zeroDraws` pass replaces `Draw` nodes with zeroes:

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

Finally, `swapBits` swaps ones and zeroes to produce lexically smaller choices strings:

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

Replicating Hypothesis Evaluation To check that we replicated Hypothesis shrinking correctly, we replicated one of the experiments from the Hypothesis paper. MacIver and Donaldson borrowed five examples from the SmartCheck repository [141] that represent a varied range of shrinking scenarios. Each example comes with a property, a buggy implementation, and a QuickCheck generator; the goal was to run the property to find a counterexample and shrink that counterexample to the smallest possible value.

We upgraded the existing QuickCheck generators to reflective ones, making minor modifications where necessary: we replaced uses of `suchThat` with generators that satisfied invariants constructively, modified some approaches to distribution management, and added appropriate reflective annotations. These modifications are based on the observations from §4.4.3. Then, we ran each experiment 1,000 times and reported the average size of the resulting counterexamples in Table 4.1. Note that the QuickCheck and baseline numbers come from the `generate` interpretation of the upgraded reflective generator, rather than the original generator.

Table 4.1: Average size of shrunk outputs after reflective shrinking, compared with Hypothesis shrinking, QuickCheck's `genericShrink`, and un-shrunk inputs. (Mean and two standard-deviation range.) *Hypothesis experiments not re-run, data taken from [112].

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

We find that reflective shrinkers perform just as well as QuickCheck's `genericShrink` in all cases, and significantly better in `bound5`. With a few caveats, reflective shrinkers also match Hypothesis. They exhibit a higher variance in the size of counterexamples that they produce, likely because they only implement a subset of Hypothesis's shrinking strategies, but nevertheless their counterexamples are on average within 1 unit of Hypothesis (and usually much closer). The worst experiment relative to Hypothesis is `bound5`; in that example, we suspect the difference is due to differing strategies for generating integers, rather than anything to do with shrinking directly.

A Realistic Example As a final demonstration that reflective shrinkers are useful, we return to a modified version of the JSON example used in §4.5. We define a generator for "`package.json`" files, which are used as a configuration format in Node.js. Programs that process these files may be used by millions of users, so a user may indeed find a bug in the program that a PBT regime did not.

Imagine a scenario where a user finds a bug where a program behaves incorrectly only when the file specifies a specific version of a specific package. In this case, shrinking would be extremely helpful: it would help the developers of the program isolate the exact lines in the JSON file that cause a problem. But shrinking a JSON file like this is impossible for both `genericShrink` and Hypothesis. The former does not work because the format is too unstructured: the generator produces JSON strings, rather than a Haskell datatype, so the best the shrinker could do is shorten the string (which would result in invalid JSON). The latter does not even start to shrink, since the JSON file came from a user, and therefore there is no random bitstring to shrink.

A reflective shrinker, however, works perfectly. The full version of the paper shows two JSON documents, the first a full "buggy" version and the second a shrunk version. The shrunk JSON document could point a developer to the precise issue with their program.

4.6.3 *Reflective Mutation*

HypoFuzz, a tool for coverage-guided fuzzing [39] of PBT properties, is a newer and lesser-known part of the Hypothesis ecosystem. Like Crowbar [31], HypoFuzz uses a PBT generator to aid the fuzzer. Fuzzers try to maximize code coverage by keeping track of a set of interesting inputs and *mutating* them, attempting to explore similar values and hopefully continue to cover new branches of the program. “Mutating well” can be challenging, since naïve mutations will rarely produce valid values; HypoFuzz gets around this concern with the same trick Hypothesis uses for fuzzing: mutate the randomness, not the value.

Internal mutation has the same benefits and drawbacks as internal shrinking. On the positive side, it is type agnostic, easy to use, and guarantees validity of the mutated values. On the other side, it assumes that the randomness used to produce a given value is available. It may seem like this drawback is less of an issue for mutation than it is for shrinking, since the fuzzer can just keep track of the random choices associated with each value it wants to mutate, but this is not true of the initial set of *seed inputs*. For optimal fuzzing, the seeds are provided by the user and represent some set of initially interesting values that the fuzzer can play with. But this does not work with Hypothesis-style mutation: the seeds needed for this style are not user-comprehensible values but choice sequences! Once again, reflective generators provide a compelling solution. We can simply extract a choice sequence from each seed using the `choices` interpretation.

4.7 IMPROVING THE TESTING WORKFLOW

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

4.7.1 *Reflective Checkers*

The "bigenerators" in the original work on PMPs [177] can be viewed as a special case of reflective generators. Rather than reflect on a value and produce choices, a bigenerator simply checks whether a value is in the range of the generator, effectively checking if the value satisfies the invariant that the generator enforces. Reflective generators can do this too, by reflecting on the generator's choices and asking whether there exists a set of choices that results in the desired value.

Going further, a reflective generator can calculate the *probability* of generating a particular value with the `generate` interpretation. We implement this in our artifact as an interpretation, `probabilityOf`, which tracks the different ways of generating a particular value and the likelihood of choosing those different ways. Obviously this works best when the generator's overlap is low (see §4.4.2)—in cases where overlap is exponential or infinite this may be slow or fail to terminate.

4.7.2 *Reflective Completers*

A rather different use case for reflective generators is generation based on a *partial value*. For example, imagine a binary search tree with holes:

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

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

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

This technique lets the user pick out a subspace of a generator, defined by some prefix of a value, and explore that subspace while maintaining any preconditions that generator enforces. We accomplish this with some carefully targeted hacks, representing a partial value as a value containing `undefined`: `Node (Node undefined 1 undefined) 5 undefined`.

Suppose, now, that this value were passed into a backward interpretation of `bst` from the introduction—where would things fail? The key insight is that the *only* place a reflective generator manipulates its focused value is when re-focusing. In other words, the only place a backward interpretation can crash on a partial value is while interpreting Lmap. Capitalizing on this insight, we wrap the standard Lmap interpretation in a call to catch, Haskell's exception handling mechanism:

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

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

This trick works best for "structural" generators that only do shallow pattern matching in Lmaps, things fall apart if the backward direction needs to evaluate the whole term. The clearest example of this is comap typeOf type_ >>= expr (recall, it generates a type and then a program of that type); in the backward direction, this generator immediately evaluates the whole term to compute its type. For this generator, complete would just generate a totally fresh program.

Users may be able to work around this by making their predicates lazier. For example, one could imagine writing an optimistic type checking algorithm optimisticTypeOf that maximizes laziness by blindly trusting user-provided type annotations. The user could then use the reflective generator

```
comap optimisticTypeOf type_ >>= expr
```

to complete an incomplete term like

```
App (Ann (Int :-> Int) undefined) (Ann Int undefined).
```

The completer would successfully determine that the type of the whole expression is Int, and then it would have enough information to complete the undefineds with well-typed expressions.

4.7.3 *Reflective Producers*

Weighted random generation in the style of QuickCheck is not the only way to get test inputs: both enumeration [151; 12; 35] and guided generation [39; 180] have been explored as alternatives. Indeed, much of the PBT literature has moved from talking about *generators* to talking about *producers* of test data, where the specific strategy does not matter [155; 137]. We say "generators" here because it is familiar and concrete, but reflective generators might better be considered as *reflective producers* because they can be used in these styles.

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

```
enumerate :: Reflective b a -> ⟦a⟧
```

for "roughly size-based" enumeration, leaning heavily on the combinators and techniques found in LeanCheck [12]. We say "roughly" because, whereas LeanCheck enumerators allow the user to define their own notion of size for each enumerator, reflective generators are limited to a single notion of size based on the number and order of choices needed to produce a given value. A thorough evaluation of this discrepancy would require its own study, but early experiments are promising. For example, `enumerate (bst (1, 10))` reaches size-4 BSTs before its 10th enumerated value and matches the sizes for an idiomatic LeanCheck enumerator (see the Appendix).

While fuzzing is sometimes treated as a separate topic from PBT—focused on finding crash failures by generating inputs external to a system rather than finding more subtle errors in individual functions—a number of recent projects have attempted to bridge the gap, and reflective generators may offer a useful unifying framework for such efforts. We already saw that reflective mutators are helpful in the context of HypoFuzz-style mutation; reflective generators can also be used to interface with an external fuzzer in the style of Crowbar [31], which is designed to get its inputs from popular fuzzers like AFL or AFL++ [180; 39]. Since Crowbar already uses a free-monad-like structure to represent its generators, we can imagine writing an equivalent reflective generator interpretation that works the same way. More generally, reflective generators can be used in any producer algorithm that uses a generator as a parser.

4.8 RELATED WORK

This work builds on the ideas of free generators [55] and partial monadic profunctors [177]. Free generators are, in turn, built on top of freer monads [94], which were initially invented as a better way to represent effectful code in pure languages. While our implementation remains faithful to the basic conception of freer monads, there are many insights from Kiselyov and Ishii that we have not yet explored. Likewise, PMPs are part of the long tradition of bidirectional programming [40], and it remains to be seen if there are stronger ways to tie reflective generators to work on other bidirectional abstractions.

The concrete realization of reflective generators is also related to Crowbar [31]. Both libraries are implemented with a syntactic, uninterpreted representation for generators, although the Crowbar version does not incorporate any ideas from monadic profunctors and uses a different type for bind that does not normalize as aggressively.

The idea of reflective generators was originally sparked by the tools developed in *Inputs from Hell* [155], and these tools in turn tie into the broader world of grammar-based generation [49; 76; 166; 170; 157; 6; 48]. Grammar-based approaches are less expressive than monadic ones, since they can only generate strings from a context-free grammar, and therefore cannot generate complex data structures with internal dependencies.

Replicating example distributions is a classic problem in *probabilistic programming* [60]. While the goals of probabilistic programs are usually quite different from those of PBT generators, there is some overlap in the formalisms used to express these ideas. In particular, one representation of probabilistic programs in the functional programming literature [183] uses a free monad that is similar to free and reflective generators.

Reflective shrinking and mutation build heavily on ideas in the Hypothesis framework [112; 113], but there are other approaches to automated shrinking. As mentioned in §4.6, QuickCheck provides `genericShrink`, which provides a competent shrinker for any Haskell type that implements `Generic`. While `genericShrink` is a decent starting point, it fails to shrink unstructured data (like strings) and values with complex preconditions. Another alternative is provided by Hedgehog [158], another Haskell PBT library. Hedgehog shrinking is similar to Hypothesis shrinking, both using the structure of the generator to automatically shrink values. It has the same limitation: externally provided values cannot be shrunk.

4.9 CONCLUSION AND FUTURE DIRECTIONS

Reflective generators are a powerful abstraction for producing and manipulating test inputs. We have developed their theory and demonstrated their utility in a variety of testing scenarios, including example-based generation, shrinking, mutation, precondition checking, value completion, enumeration, fuzzing, and more. We plan to build on reflective generators, automating their creation and improving their usability.

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

We plan to work with Hoogle+ [84], using its type-based synthesis algorithm to obtain candidate programs for the annotations with no user intervention. This is an especially compelling opportunity because it is easy to tell whether annotations are correct: they must be sound and complete, as described in §4.4. When synthesizing multiple annotations at the same time, the system can even use the number of examples that pass or fail the soundness and completeness properties as a way to infer which annotations are correct and which need to be re-synthesized—if changing an annotation increases the number of passing tests, it is more likely to be correct; if the change causes more tests to fail, it is likely wrong. If this idea works, it could make transitioning from QuickCheck generators to reflective generators almost entirely automatic.

Usability We have taken care to design an API for reflective generators that aligns with existing QuickCheck functions and minimizes programmer effort. Our own experience writing reflective generators studies has been positive, and, except for the aforementioned limitations (§4.4.3), we ran into no issues upgrading existing generators. The automation techniques hypothesized above could make reflective generators even more usable. Still, we certainly do not constitute a representative sample of PBT users: the usability of reflective generators should be studied empirically.

There is a growing push in the PL community to incorporate ideas and techniques from human-computer interaction (HCI) [19], and this is a perfect opportunity to join that movement. We will collaborate with HCI researchers on a usability analysis of reflective generators. Inspired by prior work [24], this analysis will be useful for refining our design.

CHAPTER 5

Understanding Randomness

The inspiration from this chapter came from OB6 in Chapter 2, and explores RO6: both concern the techniques developers use (or do not use) to evaluate testing effectiveness. I discuss a new interface called Tyche, which gives developers better and more interactive tools for understanding what happened when their property-based test executed. Tyche is currently available as an open-source project, and it has been adopted as an officially supported tool in the Hypothesis framework's ecosystem. The content in this chapter is taken from a paper that has been conditionally accepted at the Symposium on User Interface Software and Technology (UIST) 2024. The project is a collaboration with Jeffrey Tao, Benjamin C. Pierce, and Andrew Head at the University of Pennsylvania and Zac Hatfield-Dodds who has affiliations at the Australian National University and Anthropic.

5.1 INTRODUCTION

Software developers work hard to build systems that behave as intended. But software is rarely 100% correct when first implemented, so developers also write tests to validate their work, detect bugs, and check that bugs stay fixed. Traditionally, these tests take the form of manually written "example-based" tests, where developers write out specific sample inputs together with expected outputs; but this process is labor-intensive and can miss bugs in cases the programmer did not think to check. Instead, some programmers have adopted automated techniques to supplement or replace example-based tests. One such technique is *property-based testing* (PBT), which automatically samples many program inputs from a random distribution and checks, for each one, that the system's behavior satisfies a set of developer-provided properties. Used well, this leads to testing that is more thorough and less laborious; indeed, PBT has proven effective in identifying subtle bugs in a wide variety of real-world settings, including telecommunications software [4], replicated file and key-value stores [82; 11], automotive software [5], and other complex systems [80].

Of course, automation comes with tradeoffs, and PBT is no exception. In PBT, there are often too many automatically generated test inputs for a developer to understand at once, creating a gulf of evaluation for test suite quality. Indeed, in a recent study of the human factors of PBT [59], developers reported having difficulty understanding what was really being tested.

For example, suppose a developer is testing some mathematical function using randomly generated floating-point numbers. The developer might have a variety of questions about their test suite quality. They might ask if the distribution is broad enough (e.g., is it stuck between `0` and `1`), or too broad (e.g., does it span all possible floats, even if the function can only take positive ones). Or they may wonder if the distribution misses corner-cases like `0` or `-1`. Perhaps most importantly, they may want to know if the data generator produces too many malformed or invalid test inputs (e.g., `NaN`) that cannot be used for testing at all. State-of-the-art PBT tooling does not give adequate tools for answering these kinds of questions: any of these erroneous situations could go unnoticed because necessary information is not apparent to the user. As a result, developers may not realize that their tests are not thoroughly exercising some important system behaviors.

This gulf of evaluation presents an opportunity to rethink user interfaces for testing. HCI has made strides in helping developers make sense of large amounts of structured program data, whether by revealing patterns that manifest in many programs [47; 46; 178; 70] or comparing the behavior of program variants [168; 162; 182]. As developers adopt PBT, it is critical to tackle the related problem of helping programmers understand a summary of hundreds or thousands of executions of a single test.

To address this problem, we propose TYCHE[10], an interface that supports sensemaking and exploration for distributions of test inputs. TYCHE's design was inspired by a review of PBT usability and refined through iterative design with the help of expert PBT users; this refinement identified design principles that clarify the information needs of PBT users. TYCHE provides users with an ensemble of visualizations, and, while each individual visualization is well-understood by UI researchers, the specific combination of visualizations is novel and fine-tuned to the PBT setting. TYCHE's visualizations provide high-level insight about the distribution of test inputs as well as various aspects of test efficiency (see Figure 1). TYCHE also supports visualization and rapid drill-down into user-defined attributes of input data, taking advantage of existing hooks in PBT libraries.

To understand whether TYCHE actually changes how developers understand their tests, we conducted

[10]Named after the Greek goddess of randomness.

a 40-participant, self-guided, online study. In this study, participants were asked to view test distributions and rank them according to their power to identify program defects. Compared to using a standard PBT tool, using Tyche led to better judgments about the bug-finding power of test distributions.

To encourage broad adoption of Tyche, we further define OpenPBTStats, a standard format for reporting results of PBT. When a PBT framework generates data in this format, its results can be viewed in Tyche and perhaps other interfaces supporting the same standard in the future. We integrated OpenPBTStats into the main branch of Hypothesis [113], the most widely-used PBT framework, showing the way forward for other frameworks.

After discussing background (§5.2) and related work (§5.3), we offer the following contributions:

- We articulate design considerations for Tyche, motivated by a formative study with experienced PBT users. (§5.4)
- We detail the design of Tyche, an interface that helps developers evaluate the quality of their testing with an ensemble of visualizations fine-tuned to PBT with lightweight affordances to support exploration. (§5.5)
- We define the OpenPBTStats format for collecting and reporting PBT data to help standardize testing evaluation across different PBT frameworks. (§5.6)
- We evaluate Tyche in an online study, demonstrating that Tyche guides developers to significantly better assessments of test suite effectiveness. (§5.7)

We conclude with directions for future work (§5.8), including other automated testing disciplines that can benefit from Tyche and related interfaces.

5.2 BACKGROUND

We begin by describing property-based testing and reviewing what is known about its usability and contexts of use.

5.2.1 *Property-Based Testing*

In traditional unit testing, developers think up examples that demonstrate the ways a function is supposed to behave, writing one test for each envisioned behavior. For example, this unit test checks that inserting a key `"k"` and value `0` into a red–black tree [173] and then looking up `"k"` results in the value `0`:

```
def test_insert_example():
    t = Empty()
    assert lookup(insert("k", 0, t), "k") == 0
```

If one wanted to test more thoroughly, they could painstakingly write dozens of tests like this for many example trees, keys, and values. Property-based testing offers an alternative, succinct way to express many tests at once:

```
@given(trees(), integers(), integers())
def test_insert_lookup(t, k, v):
    assume(is_red_black_tree(t))
    assert lookup(insert(k, v, t), k) == v
```

This test is written in Hypothesis [113], a popular PBT library in Python. It randomly generates triples of trees, keys, and values, and for each triple, checks a parameterized property that resembles a unit-test assertion. This single test specification represents a massive collection of concrete individual tests, and using it can lead to more thorough testing (compared to a unit test suite), since the random generator may produce examples the user might not have thought of.

5.2.2 *PBT Process and Pitfalls*

At its core, the practice of PBT involves three distinct steps: defining executable properties, constructing random input generators,[11] and reviewing the result of checking these properties against a large number of sampled inputs; challenges can arise at any of these stages. There is significant work on each of the first two stages [99; 103; 102; 58, etc.]. We are focused here on the third stage: helping developers review the results of testing, in part to support the (often iterative) process of refining and improving the generators

[11]Some approaches to PBT use exhaustive enumeration [151] or guided search [104] instead of hand-written input generators; these change the story slightly, but ultimately results should always be reviewed to ensure that testing was successful.

constructed in the second step. For instance, in the example above, a developer might accidentally write a `trees()` generator that only produces the `Empty()` tree, in which case their property will be checked only against a single test input (over and over). Or, if the generator's strategy is not quite so broken but still too naïve, it might fail to produce very many trees that actually pass the `assume(is_red_black_tree(t))` guard.

In cases like these, developers need to remember that, although all their tests are succeeding, this does not necessarily mean their code is correct [105]: they may need to improve their generators to start seeing failing tests. Unfortunately, with conventional PBT tools, developers may feel they don't have easy access to this knowledge [59]. While the programming languages community is continually developing better techniques for generating well-distributed inputs [55; 159; 103; 124, etc.], developers still need to be able to check that the generators they are using are actually fit for the job.

5.3 RELATED WORK

In this section we situate our work on Tyche within the larger area of programming tools research.

5.3.1 *Current Affordances*

What support do PBT frameworks provide today for developers to inspect test input distributions? We surveyed the state of practice in the most popular PBT frameworks (by GitHub stars) across six different languages: Python [113], TypeScript / JavaScript [34], Rust [41], Scala [129], Java [77], and Haskell [20]. These frameworks provide users with the following kinds of information. (A detailed comparison of framework features can be found in the Appendix of the full paper.)

Raw Examples All of these frameworks could print generated inputs to the terminal. Some (3/6) provided a flag or option to do so; the others did not provide this feature natively, although users might simply print examples to the terminal themselves.

Number of Tests Run vs. Discarded Many frameworks (4/6) report how many examples were run vs. discarded (because they did not pass a quality filter), sometimes (2/6) hidden behind a command line flag.

Event / Label Aggregation Many frameworks (4/6) could report aggregates of user-defined features of the data distribution—e.g., lengths of generated lists. Information about such features typically appeared in a simple textual list or table, as in this example from QuickCheck [160]:

```
7% length of input is 7
6% length of input is 3
5% length of input is 4
...
```

I.e., among the generated lists for some test run, 7% were 7 elements long, 6% were 3 elements long, etc.

Time One framework reported how long the test run took.

Warnings One framework provided warnings about test distributions, in particular warning users when their generators produced a very high proportion of discarded examples.

The affordances for evaluation available in existing frameworks are situationally useful, but inconsistently implemented and incomplete. In §5.5 and §5.6 we discuss how TYCHE improves on the state of the art.

5.3.2 *Interactive Tools for Testing*

Some of the earliest research on improved interfaces for testing focused on spreadsheets. Rothermel et al. [148] proposed a model of testing called "what you see is what you test" (WYSIWYT), wherein users "test" their spreadsheet by checking values that they see and marking them as correct. This approach has appeared in many domains of programming, including visual dataflow [91] and screen transition languages [13]. Complementary to WYSIWYT are features that encourage programmers' curiosity [174], for instance by detecting and calling attention to likely anomalies [174; 122].

Many of the testing tools developed by the HCI community have sought to accelerate manual testing with rich, explorable traces of program behavior [15; 131; 32; 117; 118]. These tools instrument a program, record its behavior during execution, and then provide visualizations and augmentations to source code to help programmers pinpoint what is going wrong in their code. Tools can also help programmers create automated tests from user demonstrations. For instance, Sikuli Test [18] lets application developers create automated tests of interfaces by demonstrating a usage flow with the interface and then entering assertions of what interface elements should or should not be on the screen at the end of the flow.

Recent research has explored new ways to bring users into the loop of randomized testing. One research system, NaNoFuzz [28], shows programmers examples of program outputs and helps them to notice problematic results like `NaN` or crash failures. NaNoFuzz is superficially the closest comparison available for Tyche, but the two serve different, complementary purposes. NanoFuzz's strengths reside in calling attention to failures; Tyche's strengths reside in exposing patterns in input distributions. One could imagine a user leveraging both in concert during the testing process.

5.3.3 *Making Sense of Program Executions*

In a broad sense, Tyche's aim is to help developers reason about the behavior of a program across many executions. This problem has been explored by the HCI community. Tools have been developed to reveal the behavior of a program over many synthesized examples [181], and of an expression over many loops [90; 107; 154; 73]. The problem of understanding input distributions has been of interest in the area of AI interpretability, where tools have been built to support inspection of input distributions and corresponding outputs (e.g., [74; 75]). Tyche's aim is to tailor data views and exploration mechanisms to tightly fit the concerns and context of randomized testing with professional-grade software and potentially-complex inputs (e.g., logs, trees).

Prior work has sought to help programmers make sense of similarities and differences across sets of programs. Some of these tools cluster programs on the basis of aspects, semantics, or structure [46; 70; 178; 47]. Others highlight differences in the source and/or behavior of program variants [150; 168; 162; 182]. Tyche itself does some lightweight clustering of test cases (in this case, input examples), and affordances for program differencing could be brought to Tyche to help programmers pinpoint where some instantiations of a property fail and others succeed.

5.3.4 *Formal Methods in the Editor*

Property-based testing can be seen as a kind of *lightweight formal method* [175], in that it allows programmers to specify precisely the behavior of their program and then verify that the specification is satisfied. Tyche joins a family of research projects that bring formal methods into the interactive editing experience, whether to support repetitive edits [116; 128], code search [126], program synthesis [181; 143; 169; 33], or bidirectional editing of programs and outputs [72].

5.4 FORMATIVE RESEARCH

Our design for Tyche is informed by formative research into the user experience of PBT. Below, we describe our methods for formative research (§5.4.1), followed by a crystallization of user needs (§5.4.2) and a set of design considerations for Tyche (§5.4.3).

5.4.1 *Methods*

To better understand what developers need in understanding their PBT distributions, we collected two kinds of data.

Review of related work

We reviewed user needs relating to evaluating testing effectiveness as discussed at length in Goldstein et al. [59]'s recent paper on the human factors of PBT.

Iterative design feedback

As we developed Tyche, we continually sought and integrated feedback on its design from experienced users of PBT. We recruited 5 such users through X (formerly Twitter) and our personal networks. We refer to them as P1–P5.

For each of these users, we conducted a 1-hour observation and interview session. Each session was split into two parts. In the first part, participants showed us PBT tests they had written, described those tests, and answered questions about how they evaluate (or could evaluate) whether those tests are effective. In the second part, participants installed our then-current prototype and used it to explore the effectiveness of their own tests.[12] Study sessions were staggered throughout the design process. We altered the design to incorporate feedback after each session.

Initial prototype All Tyche prototypes were developed as VSCode [120] extensions. All prototypes focused on providing visual summaries of PBT data in a web view pane in the editor. The initial Tyche prototype was informed by observations from Goldstein et al. [59]'s study and the authors' experiences us-

[12] P5 showed us older code that they no longer had the test runner for, so they only saw Tyche running on our examples.

ing and building PBT tools. The initial prototype was published at UIST 2023 as a demo [54]; the prototype summarized the following aspects of test data:

- "Number of Unique Inputs": New PBT users are sometimes surprised that their test harness produces duplicate data. Knowing how many unique inputs were tested is therefore one important signal of the test harness' efficiency.
- "Proportion of Valid Inputs": As discussed in §5.2.2, PBT test harnesses sometimes discard data that does not satisfy necessary preconditions. Developers need to know how much of the data is discarded and how much is kept.
- "Size Distribution": Testers need to keep track of the size of their inputs. It is commonly believed in the PBT community that software can be tested well by exhaustive sets of small inputs (i.e., the *small scope hypothesis* [2]), and alternatively, that large tests have a *combinatorial advantage* [79] in finding more bugs.[13] Whichever viewpoint a tester subscribes to, it is important to know the sizes of inputs.

Analysis Interviews were automatically transcribed by video conferencing software[14], and analyzed via thematic analysis [10].

5.4.2 *Testing Goals and Strategies*

The first result of our formative research was a clarification of PBT users' goals and strategies when they were attempting to determine the effectiveness of their tests. One might imagine that testing effectiveness could be measured by the proportion of bugs found, but this is a fantastical measure: if we had it, we would know what all the bugs are and wouldn't need to do any testing! As we found in our study sessions, developers pay attention to proxy metrics to gain confidence in their test suite. Ideally, PBT tools will surface these metrics. Here, we discuss the various metrics that developers paid attention to and how they measured them.

[13]The truth seems to be that each of these viewpoints is correct in some situations; a recent study [153] has a nice discussion.
[14]P3's interview audio was lost due to technical difficulties, so we instead analyzed the notes we took during their interview.

Test Input Distribution

Participants reported checking that their distributions covered potential edge cases like $x = 0$, $x = -1$, or $x =$ `Integer.MAX_VALUE` (P2), which are widely understood as bug-triggering values. They also checked that their distributions covered regions in the input space like $x = 0$, $x < 0$, and $x > 0$ (P2, P4, P5); this kind of coverage is similar to notions of "combinatorial coverage" discussed in the literature [100; 56].

Multiple participants (P1, P3, P4) wanted to know that their test data was realistic. Their justification was that the most important bugs are the ones that users were likely to hit. Another participant (P5) wanted their test data to be uniformly distributed across a space of values. They thought that this would make it easier for them to estimate the probability that there was still a bug in the program. Whether to test with realistic or uniform distributions is a topic of debate in the literature, with some tools favoring uniformity [22; 124] and others realism [155]. In either case, developers should be able to see the shape of the distribution.

Participants used a combination of strategies to review these proxy metrics of test quality. Some (P1, P3, P5) read through the list of examples. Others (P2, P5) described using evaluation tools already present in their PBT framework of choice; one participant used `events` in Hypothesis and another used `labels` in QuickCheck, both to understand coverage of attributes of interest (e.g., how often does a particular variant of an enumerated type appear). As we show in §5.2, while some PBT frameworks provide views of distributions of user-defined input features, they are difficult to interpret at a glance and can easily get drowned out among other terminal messages.

Coverage of the System Under Test

Three participants (P2, P4, P5) mentioned coverage of the system under test (e.g., line coverage, branch coverage, etc.) was an important proxy metric. Two participants (P2, P4) reported actually measuring code coverage via code instrumentation, although P2 did point out the potential failings of code coverage (calling it "necessary but not sufficient"). This view is supported by the literature, which suggests that coverage alone does not guarantee testing success [97].

Test Performance

Finally, two participants (P1, P2) discussed timing performance as an important proxy for testing effectiveness. They argued that they have a limited time in which to run tests (e.g., because the tests run every time a new commit is pushed or even every time a file is saved), so faster tests (more examples per second) will exercise the system better. They measured performance with the help of tools built into the PBT framework.

Besides these metrics, participants also expressed being more confident in their tests when they understood them (P3), when the test had failed previously (P3), and when a sufficiently large (for some definition of large) number of examples had been executed (P1, P4).

5.4.3 *Design Considerations*

Our formative research further clarified what is required of usable tools for understanding PBT effectiveness. We call these requirements our five design considerations for the Tyche interface. They are:

Visual Feedback Our goal to provide better visual feedback from testing arose from Goldstein et al. [59]'s study and was validated by participants. Most participants (P1, P2, P3, P5) appreciated the interface's visual charts, stating that the visual charts are "a lot easier to digest than" the built-in statistics printed by Hypothesis (P2). The previous section (§5.4.2) clarifies the specific proxy metrics that developers were interested in visualizing.

Workflow Integration Our initial prototype was built to have tight editor integration. It could be installed into VSCode in one step, and updated live as code changed. But, while some participants validated this choice (P1 and P2), another said they were "not always a big fan of extensions" because they use a non-VSCode IDE at work (P4). For that participant, an editor extension actually *dis*courages use. We therefore refocused on workflow integration instead of editor integration, and re-architected our design so that it could plug into other editing workflows.

Customizability Participants found that the default set of visualizations was a good start (P1, P2, P3, P5), but they also suggested a slew of other visualizations that they thought might improve their testing evaluation. Many of these visualizations (e.g., code coverage (P1, P3, P5) and timing (P1)—see §5.5.2) were integrated into Tyche. What we could not do was add views that summarized the interesting attributes of each person's data: every testing domain was different. Thus, tools should be customizable so developers can acquire visual feedback for the information that is important in their testing domain.

Details on Demand Almost all participants (P1, P2, P3, P4) expressed a desire to dig deeper into the visualizations they were presented. This means that Tyche should provide ways for developers to look deeper into the details of the data that is being displayed by the visual interfaces.

Standardization Participants used PBT in multiple programming languages, including Python (P1, P2, P3, P4), Java (P4), and Haskell (P5). We posit that to improve the testing experience for all of these languages and their PBT frameworks without significant duplicated effort, Tyche needs to standardize the way it communicates with PBT frameworks. Since PBT frameworks largely implement the same test loop, despite superficial implementation differences, this standardization seems technically feasible.

5.5 SYSTEM

In this section, we describe the design of Tyche, bringing together the principles we identified in §5.4.3. We describe the interaction model that we imagine for Tyche (§5.5.1), Tyche's visual displays that answer PBT users' questions (§5.5.2), and integrations with PBT frameworks that support easy use and configuration of displays (§5.5.3).

5.5.1 *Interaction Model*

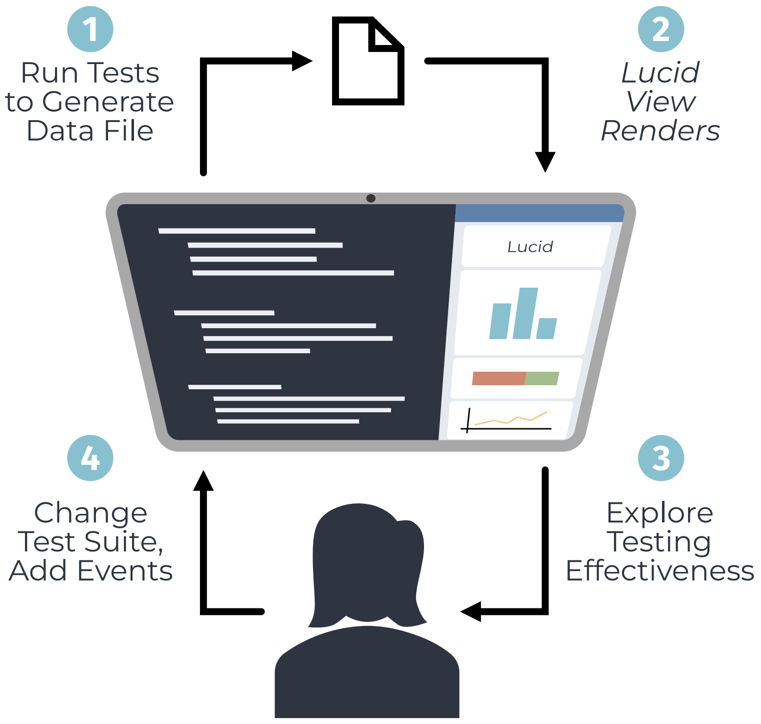

1
Run Tests to Generate Data File
2
Lucid View Renders
Lucid
3
Explore Testing Effectiveness
4
Change Test Suite, Add Events

Figure 5.1: The Tyche interaction loop.

We envision user interactions with Tyche to follow roughly the steps outlined in Figure 5.1.

(1) At the start of the loop, the developer runs their tests, and the test framework (e.g., Hypothesis) collects relevant data into a OpenPBTStats log (we discuss the details of this format in §5.6).

(2) Once the data has been logged, the user sees Tyche render an interface with variety of visualizations (see §5.5.2).

(3) The user interacts with the interface. This may be as simple as seeing a visualization and immediately noticing that something is wrong, but they may also explore the views to seek details about surprising results or generate hypotheses about what might need to change in their test suite. If the user is happy with the quality of the test suite at this point, they may finish their testing session.

(4) Finally, the user can customize their Tyche visualizations or make changes to their test (e.g., random

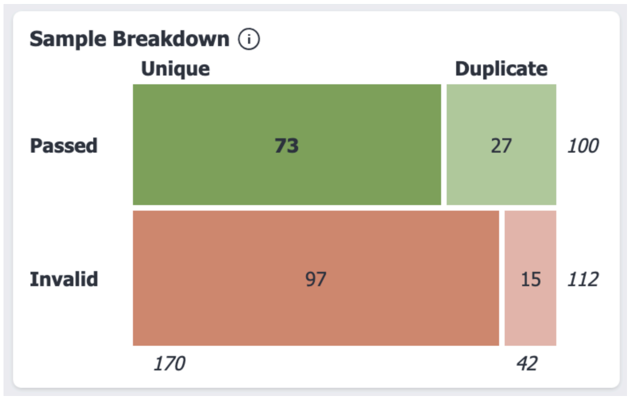
generation strategies or Hypothesis parameters) before going back around the loop.

5.5.2 *Visual Feedback*

The Tyche interface presents the user with a novel ensemble of visualizations that are fine-tuned to the PBT setting and enriched with lightweight affordances to support exploration. We describe these visualizations in the context of the kinds of questions they answer for developers.

How many meaningful tests were run?

Perhaps the most important thing for a developer to know about a test run is how many meaningful examples were tested. Tyche communicates this information through the "Sample Breakdown" chart:



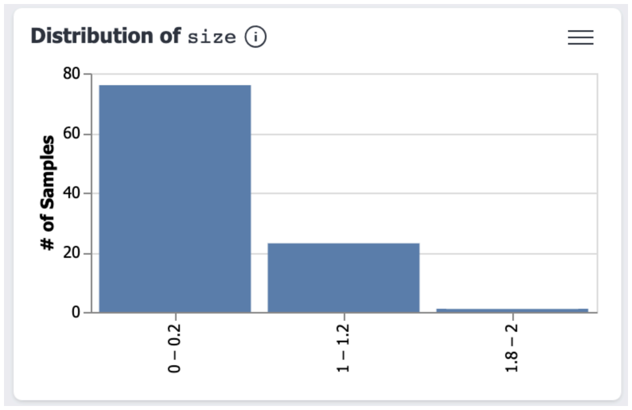
The chart communicates a high-level understanding of how many test inputs were sampled versus how many were run (because they were found to be "valid"). Ideally, the entire chart would be taken up by the dark green "Unique / Passed" bar. If the "Invalid" bars are a large portion of the chart's height or the "Duplicate" bars are a large portion of the width, the developer can see that it might be worth investing time in a generation strategy that is better calibrated for the property at hand. (If any tests had failed, there would be two more horizontal bars with the label "Failed.")

The use of a mosaic chart [64] here allows Tyche to communicate information about validity and uniqueness in a single chart. We chose this chart after feedback from study participants suggested that seeing validity and uniqueness metrics separately made it hard to tell when and how they overlapped.

How are test inputs distributed?

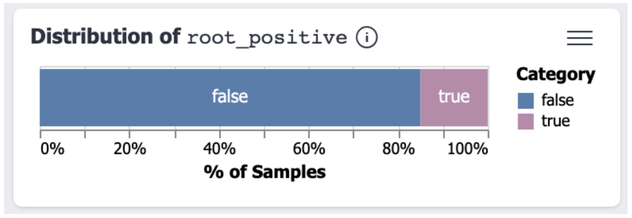
After checking the high-level breakdown of test inputs, the next questions in the user's mind will likely be about the *distribution* of inputs used to test their property. Since test inputs are structured objects (e.g., trees, event logs, etc.), it is difficult to observe their distribution directly: what would it even mean to plot a distribution of trees? Instead, the developer can visualize *features* of the distribution by plotting numerical or categorical data extracted from their test inputs.

For example, the following chart shows a distribution of `sizes` projected from a distribution of red–black trees:



Charts like these give developers windows into their distributions that are much easier to interpret than either the raw examples or the statistics reported by frameworks like Hypothesis: the chart above, for example, shows that the distribution skews quite small (actually, most trees are size 0!), which is a significant problem for test success (see in §5.4).

Distributions for categorical features (e.g., whether the value at the root of a red-black tree is positive or not) are displayed in a different format:

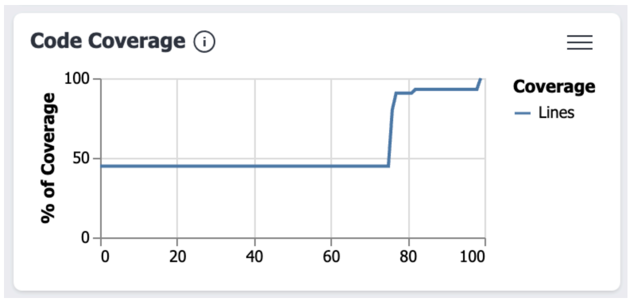


Categorical feature charts can be especially useful for helping developers understand whether there are portions of the broader input space that their tests are currently missing. In this case, the developer may want to check on why so few roots are positive—in fact, it is because an empty tree does not have a positive root, and the distribution is full of empty trees!

Our formative research suggested that just these two kinds of charts covered the kinds of projections that developers cared about. In fact, participants seemed concerned that adding more kinds of feature charts could be distracting; they felt they may waste time trying to find data to plot for the sake of using the charts, rather than plotting the few signals that would actually help with their understanding. In §5.6.3 we describe how developers can design their own visualizations outside of Tyche if needed.

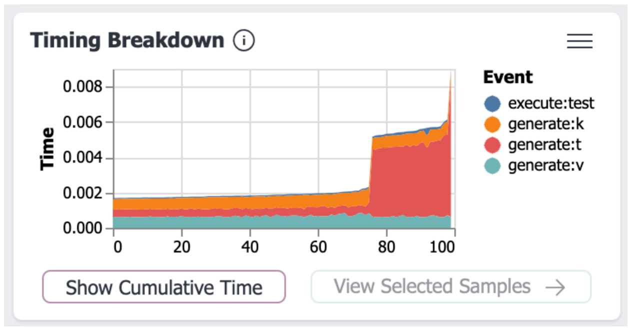
How did the tests execute overall?

The previous visualizations show information about test inputs, but developers may also have higher-level questions about what happened during testing. For example, early users, including formative research participants, asked for ways to visualize code coverage for their properties. Tyche provides the following coverage chart:



This Tyche chart shows the total code coverage achieved over the course of the test run. Note that this example is from a very small codebase, so there were really only a few disjoint paths to cover. Big jumps (around the 1st and 75th inputs) indicate inputs that meaningfully improved code coverage, whereas plateaus indicate periods where no new code was reached. As discussed in §5.4, code coverage is an incomplete way to measure testing success, but knowing that the first 70+ test inputs all covered the same lines suggests that the generation strategy may spend too long exploring a particular type of input.

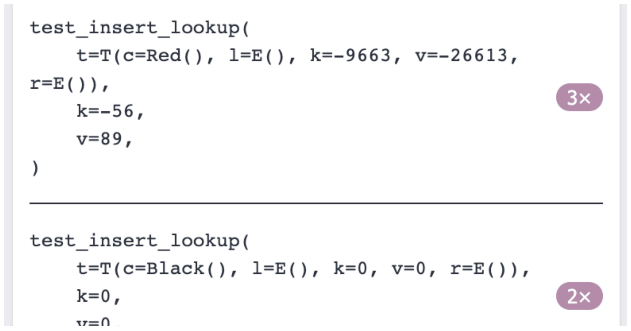
Tyche also provides charts with timing feedback, again answering a high-level question about execution that was requested by formative research participants:



The chart above shows that a majority of inputs execute quite quickly (less than 0.002 seconds) but that some twice or three times that. For the most expensive tests, the red area, signifying the time it takes to generate trees, is the largest. While users did request this chart, we are not clear how useful it is in practice. In our evaluation study (§5.7.1), participants did not generally find it helpful. However, the timing data can be used to corroborate and expand on information from other charts. For example, notice how the timing breakdown above actually mirrors the `size` chart from the previous section. The combination of these charts suggests that larger trees take much longer to generate, which suggests as trade-off that a developer should be aware of.

What test inputs were actually generated?

Although much of the point of Tyche is to avoid programmers needing to sift through individual test input examples, Tyche does make those examples available, in line with the design consideration of "details on demand:"



```
test_insert_lookup(
    t=T(c=Red(), l=E(), k=-9663, v=-26613,
r=E()),
    k=-56,
    v=89,
)
```

```
test_insert_lookup(
    t=T(c=Black(), l=E(), k=0, v=0, r=E()),
    k=0,
    v=0
```

Each example in the view shows a textual representation of the generated sample that can be expanded to see metadata like execution duration and code coverage for the individual example. Examples are grouped, so that identical examples are only shown once; this manifests in the "3x" and "2x" annotations shown in the above screenshot. This grouping aligns better with the design principle of "visual feedback," and it cuts down on clutter.

The main way a user reaches the example view is by clicking on one of the selectable bars of the sample breakdown or feature distribution charts. The user can dig into the data to answer questions about why a chart looks a certain way (e.g., if they want to explore why so few of the red–black tree's root nodes are positive). Secondarily, the example view can be used to search for particular examples to make sure they appear in the sample (e.g., important corner cases that indicate thorough testing).

5.5.3 *Reactivity and Customizability*

The visualizations provided by Tyche are reactive and customizable, allowing them to integrate neatly into the developer's workflow as dictated by our design considerations.

Reactivity

Reactivity has been incorporated into an astonishing variety of programming tools (see the review by Rein et al. [146]). It is a common feature of many modern developer tools—two modern examples are Create React App [38], which reloads a web app on each source change and pytest-watch [87], one of many testing harnesses that live-reruns tests upon code changes. When run as a VSCode extension, Tyche automatically refreshes the view when the user's tests re-run. When used in conjunction with a test suite watcher (e.g., pytest-watch, which reruns Hypothesis tests when the test file is saved) this yields an end-to-end experience with "level 3 liveness" on Tanimoto's hierarchy of levels of liveness [163].

Customizability

Specifically, in step (4) of the Tyche loop, the user can tweak their testing code in ways that change the visualizations that are shown the next time around the loop.[15]

Assumptions As discussed in §5.2 with the red–black tree example, developers make assumptions about on what inputs are valid for their property. Concretely, this happens via the Hypothesis `assume` function;

[15]While Tyche works with many PBT frameworks, we describe these customizations in detail for Python's Hypothesis specifically. Other frameworks may choose to implement user customization in other ways that are more idiomatic for their users.

for example:

```
def test_insert_lookup(t, k, v):
    assume(is_red_black_tree(t))
    assert lookup(insert(k, v, t), k) == v
```

The `assume` function filters out any tree that does not satisfy the provided Boolean check—in this case, that the generated tree is a valid red–black tree. In the sample breakdown, inputs that break assumptions are shown as "Invalid."

Events Hypothesis also has a feature called "events" to label property executions interesting. For example, the programmer might write:

```
if some_condition:
    event("hit_condition")
```

and then Hypothesis would output "`hit_condition: 42%`." To support richer visual displays of features, we extended the Hypothesis API (with the support of the Hypothesis developers) to allow events to include "payloads" that correspond to the numerical and categorical features in the feature charts above. Adding an event to the above property gives:

```
def test_insert_lookup(t, k, v):
    event("size", payload=size(t))
    assume(is_red_black_tree(t))
    assert lookup(insert(k, v, t), k) == v
```

These user events correspond to feature charts: the one shown here generates the `size` chart shown in the previous section.

By reusing Hypothesis's existing idioms for assumptions and events, Tyche hooks into existing developer workflows and makes them more powerful. The same pattern also applies to other PBT frameworks.

5.6 IMPLEMENTATION

In this section, we outline the implementation of the Tyche interface. We begin with the mechanics of the system itself (§5.6.1), but the most interesting part is the standardized OpenPBTStats format that PBT frameworks use to send data to Tyche (§5.6.2). In §5.6.3 we explain how the Tyche architecture makes it easy to extend the ecosystem of related tools.

5.6.1 *UI Implementation*

At the implementation level, Tyche is a web-based interface that is easy to integrate into existing PBT frameworks.

React Application

Tyche is a React [167] web application that consumes raw data about the results of one or more PBT runs and produces interactive visualizations to help users make sense of the underlying data. The primary way to use Tyche is in the context of an extension for VSCode that shows the interface alongside the tests that it pertains to, but it is also available as a standalone webpage to support workflow integration for non-VSCode users. (When running as an extension, Tyche is still fundamentally a web application: VSCode can render web applications in an editor pane.)

The mosaic chart described in §5.5.2 is implemented with custom HTML and CSS, but all other charts and visualizations are generated with Vega-Lite [152]. Vega-Lite has good default layout algorithms for the types of data we care about.

Framework Integration

As discussed in §5.5.1, we worked with the Hypothesis developers to make a few small changes to enable Tyche; other PBT tools require similar changes. The Hypothesis developers added callback to capture data on each test run, and it was easy to use this callback to produce data for Tyche. This data is dumped in the OpenPBTStats format, which we discuss in §5.6.2.

In Hypothesis specifically, we also adapted the `event` function to have a richer API, described in §5.5.3.

5.6.2 *OpenPBTStats Data Format*

Tyche uses an open standard that PBT frameworks can use to integrate with it and related tools.

```
{
  line_type: "example",
  run_start: number,
  property: string,
  status: "passed" | "failed" | "discarded",
  representation: string,
  features: {[key: string]: number | string}
  coverage: ...,
  timing: ...,
  ...
}
```

Figure 5.2: The OpenPBTStats line format.

OpenPBTStats is based on JSON Lines [171]: each line in the file is a JSON object that corresponds to one *example*. An example is the smallest unit of data that a test might emit; each represents a single test case. The JSON schema in Figure 5.2 defines the format of a single example line. Each example has a `run_start` timestamp, used to group examples from the same run of a property and disambiguate between multiple runs of data that are stored in the same file. The `property` field names the property being tested (extracted from language internals) and the `status` field says whether this example `"passed"`, meaning the property passed, or `"failed"` indicating that the example is a counterexample to the property, or `"discarded"` meaning that the value did not pass assumptions. The `representation` is a human-readable string describing the example (e.g., as produced by a class's `__repr__` in Python). Finally, the `features` reflect the data collected for user-defined events.

The full format includes a few extra fields, including some human-readable details (e.g., to explain why a particular value was discarded), optional fields naming the particular generator that was used to produce a value, and a free-form metadata field for any additional information that might be useful in the example view. A guide to using the format can be found online.[16]

[16]See [68]. Some field names been changed to clarify the explanations in the paper.

5.6.3 *Expanding the Ecosystem*

The clean divide between Tyche and OpenPBTStats means that PBT frameworks require only the modest work of implementing OpenPBTStats to get access to the visualizations implemented by Tyche, and conversely that front ends other than Tyche will work with any PBT tool that implements OpenPBTStats.

Supporting New Frameworks

Supporting a new PBT framework is as simple as extending it with some lightweight logging infrastructure. Framework developers can start small: supporting just five fields—`type`, `run_start`, `property`, `status`, and `representation`—is enough to enable a substantial portion of Tyche's features. After that, adding `features` will enable user control of visualizations; `coverage` and `timing` may be harder to implement in some programming languages, but worthwhile to support the full breadth of Tyche charts.

So far, support for OpenPBTStats exists in Hypothesis, Haskell QuickCheck, and OCaml's base-quickcheck. Our minimal Haskell QuickCheck implementation is an external library comprising about 100 lines of code and took an afternoon to write.

Adding New Analyses

Basing OpenPBTStats on JSON Lines and making each line a mostly-flat record means that processing the data is very simple. This simplifies the Tyche codebase, but it also makes it easy to process the data with other tools. For example, getting started visualizing OpenPBTStats data in a Jupyter notebook requires two lines of code

```
import pandas as pd
pd.read_json(<file>, lines=True)
```

This means that if a developer starts out using Tyche but finds that they need a visualization that cannot be generated by adding an assumption or event, they can simply load the data into a notebook and start building their own analyses.

In the open-source community, we also expect that developers may find entirely new use-cases for OpenPBTStats data that are not tied to Tyche. For example, OpenPBTStats data could be used for reporting testing performance to other developers or managers (this use-case that was mentioned by participants during our formative exploration).

5.7 EVALUATION

In this section, we evaluate Tyche. §5.7.1 presents an online self-guided study to assess Tyche's impact on users' judgments about the quality of test suites. §5.7.2 describes the concrete impact that Tyche has already had through identifying bugs in the Hypothesis testing framework itself.

5.7.1 *Online Study*

We chose this study to validate what we saw as the most critical question about the design: whether the kinds of visual feedback offered by Tyche led to improved understanding of test suites. We regarded this question as most critical because we had relatively less confidence in the effectiveness of visual feedback for helping find bugs than in other aspects of the Tyche design—indeed, it is a tall order for *any* kind of feedback to provide an effective proxy for the bug-finding power of tests. (By contrast, we felt our choices around customizability, workflow integration, details on demand, and standardization were already on solid ground—these choices were more conservative, and had previously received positive feedback from developers and PBT tool builders.)

Accordingly, we designed a study to address the following research questions:

RQ1 Does Tyche help developers to predict the bug-finding power of a given test suite?

RQ2 Which aspects of Tyche best support sensemaking about test results?

To go beyond qualitative feedback alone, we designed the study to support statistical inference about whether we had improved judgments about test distributions. This led us a self-guided, online usability study that centered on focused usage of Tyche's visual displays. The study allowed us to collect sufficiently many responses from diverse and sufficiently-qualified programmers to support the analysis we wanted.

Study Population

We recruited study participants both from social media users on X (formerly Twitter) and Mastodon and from graduate and undergraduate students in the University of Pennsylvania's Computer and Information Science department, aiming to recruit a diverse set of programmers ranging from relative beginners with no PBT experience to experts who may have some exposure.

In all, we recruited 44 participants; 4 responses were discarded because they did not correctly answer our screening questions, leaving 40 valid responses. All but one of these reported that they were at least proficient with Python, with 12 self-reporting as advanced and 9 as expert. Half reported being beginners at PBT, 13 proficient, 6 advanced, and 0 experts. Almost all participants reported being inexperienced with the Python Hypothesis framework; only 7 reporting being proficient. To summarize, the average participant had experience with Python but not PBT, and if they did know about PBT it was often not via Hypothesis.

When reporting education level, 4 participants had a high school diploma, 15 an undergraduate degree, and 20 a graduate degree. The majority of participants (24) described themselves as students; 7 were engineers; 3 were professors; 6 had other occupations; 28 participants self-identified as male, 5 as female, 2 as another gender, and 5 did not specify.

We discuss the limitations of this sample in §5.7.1.

Study Procedure

We hypothesized that Tyche would improve a developer's ability to determine how well a property-based test exercises the code under test—and therefore, how likely it is to find bugs. At its core, our study consisted of four "tasks," each presenting the participant with a PBT property plus three sets of sampled inputs for testing that property, drawn from three different distributions. The goal of each task was to rank the distributions, in order of their bug-finding power, with the help of either Tyche or a control interface that mimicked the existing user experience of Hypothesis. Concretely, the control interface consisted of Hypothesis's "statistics" output and a list of pretty-printed test input examples; the statistics output included Hypothesis's warnings (e.g., when $< 10\%$ of the sample inputs were valid). Both interfaces were styled the same way and embedded in HTML iframes, so participants could interact with them as they would if the display were visible in their editor; Tyche was re-labeled "Charts" and the control was labeled "Examples," to reduce demand characteristics.

The distributions that participants had to rank were chosen carefully: one distribution was the best we could come up with; one was a realistic generator that a developer might write, but with some flaw or inefficiency; and one was a low-quality starter generator that a developer might obtain from an automated tool. To establish a ground truth, we benchmarked each trio of input distributions using a state-of-the-art tool called Etna [153]. Etna greatly simplifies the process of *mutation testing* as a technique for determining the bug-finding power of a particular generation strategy: the programmer specifies a collection of synthetic bugs to be injected into a particular bug-free program, and Etna does the work of measuring how quickly (on average) a generator is able to trigger a particular bug with a particular property. Prior work has shown that test quality as measured by mutation testing is well correlated with the power of tests to expose real faults [89]. These ground truth measurements agreed with the original intent of the generators, with the best ones finding the most bugs, followed by the flawed ones, followed by the intentionally bad ones.

The study as experienced by the user is summarized in Figure 5.3. We started by providing participants some general instruction on PBT, since we did not require that participants had worked with it before. After some screening questions to ensure that participants had understood the instruction, we presented the main study tasks. In each, the participants ranked three test distributions based on how likely they thought they were to find bugs. Each of the four tasks was focused on a distinct property and data structure:

- *Red–Black Tree* The property described in §5.2.1 about the `insert` function for a red–black tree implementation.
- *Topological Sort* A property checking that a topological sorting function works properly on directed acyclic graphs.
- *Python Interpreter* A property checking that a simple Python interpreter behaves the same as the real Python interpreter on straight-line programs.
- *Name Server* A property checking that a realistic name server [172] behaves the same as a simpler

model implementation.

These tasks were designed to be representative common PBT scenarios: red–black trees are a standard case study in the literature [151; 153], topological sort has been called an ideal pedagogical example for PBT [127], programming language implementations are a common PBT application domain [134], and name servers capture some challenges with using PBT on systems with significant internal state [80].

To counterbalance potential biases due to the order that different tasks or conditions were encountered, we randomized the participants' experience in three ways: (1) two tasks were randomly assigned Tyche, while the other two received the control interface, (2) tasks were shown to users in a random order, and (3) the three distributions for each task were shown in a random order.

Four participants took over an hour to complete the study; we suspect this is because they started, got up for a while, and then returned to the study. Of the rest, participants took 32 minutes on average ($\sigma = 12$) to complete the study; only one took less than 15 minutes. Participants took about 3 minutes on average ($\sigma = 2.5$) to complete each task.

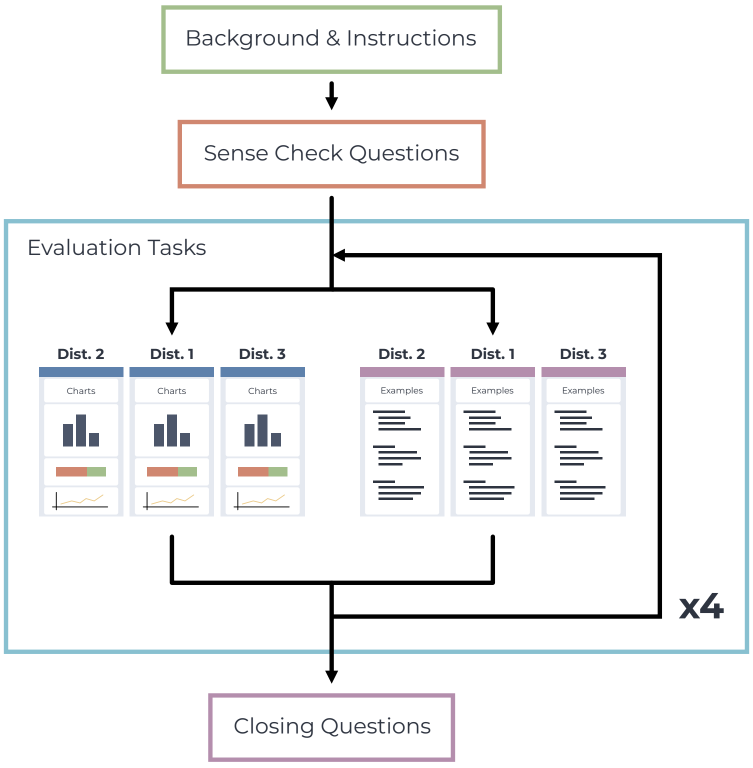


Figure 5.3: The procedure of the self-guided, online usability study.

Results

To answer **RQ1**, whether Tyche helps developers to predict test suite bug-finding power, we analyzed how well participants' rankings of the three distributions for each task agreed with the true rankings as determined by mutation testing. Given a participant's ranking, for example $D_2 > D_1 > D_3$, we compared it to the true ranking (say, $D_1 > D_2 > D_3$) by counting the number of correct pairwise comparisons—here, for example, the participant correctly deduced that $D_1 > D_3$ and $D_2 > D_3$, but they incorrectly concluded that $D_2 > D_1$, so this counted as one *incorrect comparison.*[17]

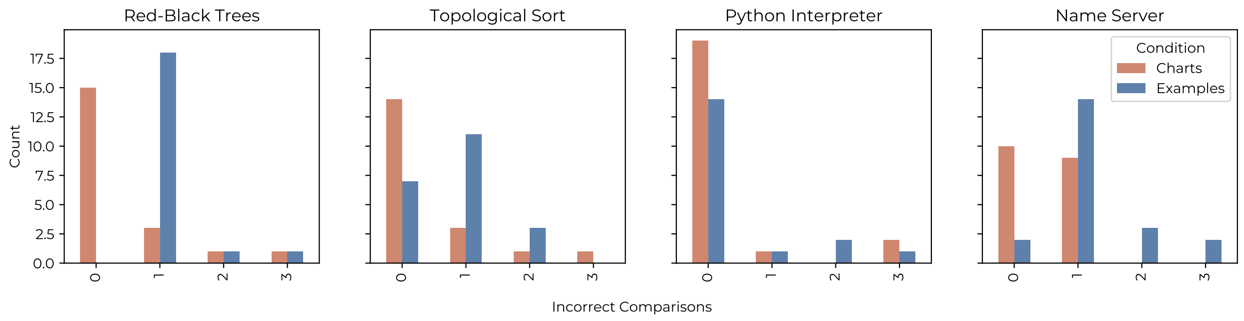


Figure 5.4: Distribution of errors made for each task, where each incorrect relative ranking between two test suites counts as one error. When using Tyche, significantly fewer errors were made for Red Black Trees, Topological Sort, and Name Server. 🟥 = Tyche, 🟦 = Control.

Figure 5.4 shows the breakdown of incorrect comparisons made with and without Tyche, separated out by task. To assess whether Tyche impacts correctness, we performed a one-tailed Mann-Whitney U test [119] for each task, with the null hypothesis that Tyche does not lead to fewer incorrect comparisons. The results appear in Table 5.1.

Table 5.1: Values for Mann-Whitney U test measuring Tyche's impact on incorrect comparisons. All sample sizes were between 18 and 22, totaling 40, depending on the random variation in the way conditions were assigned; r is common language effect size, m is the median number of incorrect comparisons.

	U	p	r	m_{Tyche}	m_{Examples}
Red–Black Trees	65	< 0.01	0.84	0	1
Topological Sort	127	0.01	0.68	0	1
Python Interp.	182	0.26	-	0	0
Name Server	91	< 0.01	0.77	0	1

[17]This metric is isomorphic to Spearman's ρ [156] in this simple case. Making 0 incorrect comparisons equates to $\rho = 1$, making 1 is $\rho = 0.5$, 2 is $\rho = -0.5$, and 3 is $\rho = -1$. We found counting incorrect comparisons to be the most intuitive way of conceptualizing the data.

For three of the four tasks (all but *Python Interpreter*), participants made significantly fewer incorrect comparisons when using Tyche, with strong common-language effect sizes, meaning that participants were better at assessing testing effectiveness with Tyche than without. Furthermore, a majority of participants got a completely correct ranking for all 4 of the tasks with Tyche, while this was only the case for 1 of the tasks without it. (For *Python Interpreter*, participants overwhelmingly found the correct answer with both conditions—in other words, the task was simply too easy—but precisely why it was too easy is interesting; see §5.7.1.) Despite this difference in accuracy, participants took around the same time with both treatments; the mean time to complete a task with Tyche was 183 seconds ($\sigma = 125$), verses 203 seconds ($\sigma = 165$) for the control. These results support answering **RQ1** with "yes."

To answer **RQ2** we used a post-study survey, asking participants for feedback on which of Tyche's visualizations they found useful. The vast majority of participants (37/40) stated that Tyche's "bar charts" were helpful. (Unfortunately, we phrased this question poorly: we intended for it to refer only to feature charts, but participants may have interpreted it to include the mosaic chart as well.) Additionally, 20/40 participants found the code coverage visualization useful, 17/40 found the warnings useful, and 14/40 found the listed examples useful. Only 4/40 found the timing breakdown useful; we may need to rethink that chart's design, although it may also simply be that the tasks chosen for the study did not require timing data to complete. These results suggest that the customizable parts of the interface—the feature and/or mosaic charts—were the most useful, followed by some of the more general affordances.

To get a sense of participants' overall impression of Tyche, we also asked "Which view [Tyche or the control] made the difference between test suites clearer?" with five options on a Likert scale. All but one participant said Tyche made the differences clearer, with 35/40 saying Tyche was "much clearer."

Discussion

Overall, the online study was an encouraging evaluation of Tyche's impact on developer understanding. In addition to the core observations above, we also made a couple of other smaller observations.

[17]This corresponds to the probability that randomly sampled Tyche participant will make fewer errors than a control participant, computed as $r = \frac{U_1}{n_1 * n_2}$.

Confidence Alongside each ranking, we asked developers how confident they were in it, on a scale from 1 to 5 ("Not at all" = 1, "A little confident" = 2, "Moderately confident" = 3, "Very confident" = 4, "Certain" = 5). We found that reported confidence was significantly higher with Tyche than without on two tasks (Red–Black Tree and Topological Sort), as computed via a similar one-sided Mann-Whitney U test to the one before ($p < 0.01$ and $p = 0.03$ respectively), with no significant difference for the other tasks. However, these results have no clear interpretation. When we computed Spearman's ρ [156] between the confidence scores and incorrect comparison counts, we found no significant relationship; in other words, participants' confidence was not, broadly, a good predictor of their success.

Non-significant Result for "Python Interpreter" Task As mentioned above, the Python Interpreter task seems to have been too easy; participants made very few mistakes across the board. We propose that this is, at least in part, because the existing statistics output available in Hypothesis was already good enough. For the worst of the three distributions, Hypothesis clearly displayed a warning that "< 10% of examples satisfied assumptions," an obvious sign of something wrong. Conversely, for the best distribution of the three, Hypothesis showed a wide variety of values for the `variable_uses` event, which was only ever 0 for the other two distributions. Critically, the list displayed was *visually longer*, so it was easy to see the variety at a glance. (We show an example of what the user saw in the Appendix of the full paper.) This result shows that Hypothesis's existing tools can be quite helpful in some cases: in particular, they seem to be useful when the distributions have big discrepancies that make a visual difference (e.g., adding significant volume) in the statistics output.

Limitations

We are aware of two significant limitations of the online study: sampling bias and ecological validity.

The sample we obtained was appropriate from the perspective of prior experience and general level of education, but it under-represents important groups regarding both gender and occupation. For gender, prior work has shown that user interfaces often demonstrate a bias for cognitive strategies that correlate with gender [16; 161], so a more gender-diverse sample would have been more informative for the study. For occupation, we reached a significant portion of students and proportionally fewer working developers. Many of those students are in computer science programs and therefore will likely be developers someday, but software developers are ultimately the population we would like to impact, so we would like to have more direct confirmation that Tyche works for them.

The other significant limitation is ecological validity. Because this study was not run *in situ*, aspects of the experimental design may have impacted the results. For example, study participants did not write the events and assumptions for the property themselves; if they had been unable to do so in practice, the outcomes may have been very different. Additionally, participants saw snippets of code, but they were not intimately familiar with, nor could they inspect, the code under test. In a real testing scenario, a developer's understanding of their testing success would depend in part on their understanding of the code under test itself. We did control for other ecological issues: for example, we used live instances of Tyche in an iframe to maintain the interactivity of the visual displays, and we developed tasks that span a range of testing scenarios. We discuss plans to evaluate Tyche *in situ* in §5.8.1

5.7.2 *Impact on the Testing Ecosystem*

Since Tyche is an open-source project that is beginning to engage with the PBT community, we can also evaluate its design by looking at its impact on practice. The biggest sign of this so far is that Tyche has led to 5 concrete bug-fixes and enhancements in the Hypothesis codebase itself. As of this writing, Hypothesis developers have found and fixed three bugs—one causing test input sizes to be artificially limited, another that badly skewed test input distributions, and a third that impacted performance of stateful generation strategies—and two long-standing issues pertaining to user experience: a nine-year-old issue about surfacing important feedback about the `assume` function and a seven-year-old issue asking to clarify terminal error messages. All five of these issues are threats to developers' evaluation of their tests; the problems were found and fixed after study participants and other Tyche users noticed deficiencies in their test suites that turned out to be library issues.

The ongoing development of Tyche has the support of the Hypothesis developers, and it has also begun to take root in other parts of the open-source testing ecosystem. One of the authors was contacted by the developers of PyCharm, an IDE focused on Python specifically, to ask about the OpenPBTStats format. They realized that the coverage information therein would provide them a shortcut for code coverage highlighting features that integrate cleanly with Hypothesis and other testing frameworks.

5.8 CONCLUSIONS AND FUTURE WORK

Tyche rethinks the PBT process as more interactive and empowering to developers. Rather than hide the results of running properties, which may lead to confusion and false confidence, the OpenPBTStats protocol and interfaces like Tyche give developers rich insight into their testing process. Tyche provides visual feedback, integrates with developer workflows, provides hooks for customization, shows details on demand, and works with other tools in the ecosystem to provide a standardized way to evaluate testing success. Our evaluation shows that Tyche helps developers to tell the difference between good and bad test suites; its demonstrated real-world impact on the Hypothesis framework backs up those conclusions.

Moving forward, we see a number of directions where further research would be valuable.

5.8.1 *Evaluation in Long-Term Deployments*

Our formative research and online evaluation study have provided evidence that Tyche is usable, but there is more to explore. For one thing, we would like to get *in-situ* empirical validation for the second half of the loop in Figure 5.1. As Tyche is deployed over longer periods of time in real-world software development settings, we are excited to assess its usability and continued impact.

5.8.2 *Improving Data Presentation for Tyche*

As the Tyche project evolves, we plan to add new visualizations and workflows to support developer exploration and understanding.

Code Coverage Visualization The visualization we provide for displaying code coverage over time was not considered particularly important by study participants: it may be useful to explore alternative designs.

One path forward is in-situ line-coverage highlighting, like that provided by Tarantula [88]. Indeed, it would be easy to implement Tarantula's algorithm, which highlights lines based on the proportion of passed versus failed tests that hit that line, in Tyche (supported by OpenPBTStats). In cases where no failing examples are found, each line could simply be highlighted with a brightness proportional to the number of times it was covered.[18]

Line highlighting is useful for asking questions about particular parts of the codebase, but developers may also have questions about how code is exercised for different parts of the input space. To address these questions, we plan to experiment with visualizations that cluster test inputs based on the coverage that they have in common. This would let developers answer questions like "which inputs could be considered redundant in terms of coverage?" and "which inputs cover parts of the space that are rarely reached?"

Mutation Testing In cases where developers implement mutation testing for their system under test, we propose incorporating that information into Tyche for better interaction support. Recall that in §5.7.1, we used mutation testing, via the Etna tool, as a ground truth for test suite quality; mutation testing checks that a test suite can find synthetic bugs or "mutants" that are added to the test suite. Etna is powerful, but its output is not interactive: there is no way to explore the charts it generates, nor can you connect the mutation testing results with the other visualizations that Tyche provides. Thus, we hope to add optional visualizations to Tyche, inspired by Etna, that tell developers how well their tests catch mutants.

Longitudinal Comparisons of Testing Effectiveness Informal conversations with potential industrial users of Tyche suggest that developers want ways to compare visualizations of the same system at different points in time—either short term, to inspect the results of changes—or longer term, to understand how testing effectiveness has evolved over time. These comparisons would make it clear if changes over time have improved test quality, or if there have been significant regressions.

Interestingly, the design of the online evaluation study accidentally foreshadowed a design that may be effective: allowing two instances of Tyche, connected to different instances of the system under test, to run side-by-side so the user can compare them. Since developers were able to successfully compare two distributions side-by-side with Tyche in the study, we expect they will also be able to if presented the same thing in practice. This is simple to implement and provides good value for little conceptual overhead on developers.

[18]Unit-test frameworks could also report simple OpenPBTStats output with one line per example-based test, enabling per-test coverage visualization for almost any test suite.

5.8.3 *Improving Control in Tyche*

Tyche is currently designed to support *existing* developer workflows and provide insights into test suite shortcomings. But participants in the formative research (P1, P4) did speculate about some ways that Tyche could help developers to adjust their random generation strategies after they notice something is wrong.

Direct Manipulation of Distributions When a developer notices, with the help of Tyche, that their test input distribution is sub-par, they may immediately know what distribution they would prefer to see. In this case, we would like developers to be able to change the distribution via *direct manipulation*—i.e., clicking and dragging the bars of the distribution to the places they should be, automatically updating the input generation strategy accordingly. One potential way to achieve this would be to borrow techniques from the probabilistic programming community, and in particular languages like Dice [78]. Probabilistic programming languages and random data generators are actually quite closely related, and the potential overlap is under-explored. Alternatively, *reflective generators* [58] can tune a PBT generator to mimic a provided set of examples. If a developer thinks a particular bar of a chart should be larger, a reflective generator may be able to tune a generator to expand on the examples represented in that bar.

Manipulating Strategy Parameters in Tyche Occasionally direct manipulation as discussed above will be computationally impossible to implement; in those cases Tyche could still provide tools to help developers easily manipulate the parameters of different generation strategies. For example, if a generation strategy takes a `max_value` as an input, Tyche could render a slider that lets the developer change that value and monitor the way the visualizations change, resembling interactions already appearing in HCI programming tools (e.g., [65; 92]). Of course, running hundreds of tests on every slider update may be slow; to speed it up, we propose incorporating ideas from the literature of self-adjusting computation [1], which has tools for efficiently re-running computations in response to small changes of their inputs.

5.8.4 *Tyche for Education*

So far we have designed Tyche to help experienced software developers gain a better understanding of their testing effectiveness, but it could just as easily help beginners to understand what is going on "under the hood" when they are learning PBT for the first time. This is especially important because prior work has argued that teaching PBT can be a stepping stone to teaching formal methods more broadly [99; 127].

To reap these benefits, we plan to make Tyche available as a learning aid when teaching PBT to undergraduate students in a large university course. We will track students' use of the tool and collect feedback about how Tyche affects students' perception of testing effectiveness. Ultimately we hope that having tools like Tyche around from the beginning of one's PBT journey will give developers better-calibrated instincts with which to gauge their testing effectiveness.

5.8.5 *Tyche Beyond PBT*

The ideas behind Tyche may also have applications beyond the specific domain of PBT. Other automated testing techniques—for example fuzz testing ("fuzzing")—could also benefit from enhanced understandability. Fuzzing is closely related to PBT, and the fuzzing community has some interesting visual approaches to communicating testing success. One of the most popular fuzzing tools, AFL++ [39], includes a sophisticated textual user-interface giving feedback on code coverage and other fuzzing statistics over the course of (sometimes lengthy) "fuzzing campaigns." But current fuzzers suffer from the same usability limitations as current PBT frameworks, hiding information that could help developers evaluate testing effectiveness. We would like to explore adapting Tyche and expanding OpenPBTStats to work with fuzzers and other automated testing tools, bringing the benefits of our design methodology to an even broader audience.

CHAPTER 6

Property-Based Testing Into the Future

I believe that in the coming years, the need for accessible, high-assurance software validation will only increase. Software development will continue to accelerate (e.g., with the rise of technologies like generative AI), but, if trends continue [86], that speed will be at the expense of correctness. Eventually we may hit a tipping point, where we are no longer willing to accept low quality software that bleeds money and puts users at risk, but nor are we willing to return to writing every line of code by hand. In this new world, powerful approaches to testing will be more important than ever.

With this in mind, I envision a number of directions for future work that, I will explore with the help of many collaborators. These directions target the different parts of the PBT process—specification, generation, and evaluation—and then come together into a unified framework for usable property-based testing.[19]

6.1 SPECIFICATION

As discussed in Chapter 2, PBT is easiest to apply in "high-leverage" scenarios where specifications are easily accessible. Indeed, this is how I would recommend that developers get started with incorporating PBT into their engineering process. But as one's expertise with PBT grows and their need for assurance increases, it is natural to look for other opportunities to bring PBT's power to bear. The PBT community needs to support a broader range of specifications and make specification tools more accessible to developers.

When trying to write properties, one important barrier that developers face is unclear software boundaries: PBT is hard to apply to programs with poorly encapsulated global state or with leaky or overly complex abstraction boundaries. A potential way around this limitation is to change where the specifications are applied; rather than write a property that aligns with function or module boundaries, the user could instead write properties about the way the program's internal behavior evolves over time. For example,

[19]Some of the ideas and explanations in this chapter come from a funded NSF Medium proposal with the same name as this dissertation. I contributed to the proposal in roughly equal collaboration with Andrew Head and Benjamin C. Pierce; the sections I borrowed verbatim were my ideas.

properties could be written about *event logs* that track the program's execution. Properties could be written declaratively in temporal logic, following O'Connor and Wickström [130], or (since linear logics can be difficult to understand [61]), the properties could explicitly describe a program that produces an abstract log, which can then be checked against the actual log.

Another problem developers face when writing properties is simply the problem of deciding on their program's specification. Tools like GhostWriter [67] and QuickSpec [21] can be helpful for this purpose, suggesting properties to users based on the shape and behavior of their code, but neither of these tools is general enough to apply reliably in real programming workflows. One might be tempted to use generative AI to help solve this problem. Indeed, tools like Copilot [45], an in-editor AI that suggests code completions, and the GPT line of tools (e.g., [132]) are surprisingly good at writing code. However, using a large language model to generate properties directly could be disastrous: test suites are part of an application's *trusted computing base* and, as such, must be carefully and thoughtfully chosen. LLMs are prone to hallucination [86], so using them to automatically write an application's tests may result in nonsense properties that miss critical bugs. Ultimately, I would like to see the community develop hybrid approaches: a user could use AI to help them to identify complex areas of their code base, brainstorm natural language correctness conditions, etc.; and then classical tools could assist programmers in actually expressing their ideas as code. Keeping the user as an active participant in the specification process is critical—designing a specification may help with bug-finding on its own [99]—but giving users help around ideation and execution will help to drive adoption.

6.2 GENERATION

Parallel to work on specifications, PBT research needs to keep pushing the boundaries on tools for random data generation. While choice gradient sampling from Chapter 3 is a good starting point for novices, and while the reflective generator language from Chapter 4 is a power tool that can be leveraged by experts, there are still limited options for developers in the middle. As I see it, the biggest challenge in property-based testing right now is: helping users create generators that satisfy *realistic* preconditions and that produce *high quality* inputs *fast* enough to catch bugs within unit-testing time budgets.

In the next few years, I am interested in helping developers obtain better generators via program synthesis. This problem is difficult because, as discussed elsewhere in this document, ideal generators must be both *sound*—producing *only* inputs satisfying a property's precondition—and *complete*—producing *all* inputs satisfying the precondition. I have two key observations that I think will make it possible to synthesize sound and complete generators for realistic preconditions:

1. Generators do not need to be built from scratch. They can be assembled from a library of sub-generators that do some of the heavy lifting.

2. Synthesis does not need to be perfect. If a system can automatically check that a generator is sound and complete, it can use imperfect synthesis techniques and simply take any correct generator that the synthesizer can find.

These observations simplify the synthesis problem to something that should be tractable for modern synthesis techniques. At the time of writing, it is unclear what synthesis approach will work best for this task. A classical approach to synthesis, e.g., type-directed synthesis in the style of Synquid [142], would provide nice guarantees and a solid formal foundation, whereas AI approaches like neuro-symbolic synthesis might be more powerful but less predictable. Finding the right combination of these ideas will be key.

Unfortunately, even a sound and complete generator might not be optimal: the *distribution* of values produced by the generator is also key. Generator tuning is the topic of a wide range of existing work, discussed in depth in Chapters 3 and 4, but existing approaches either introduce overhead during generation or only work for a narrow subset of generators. Moving forward, I see two promising paths. First, interactive approaches to tuning could significantly improve PBT performance without requiring better automation. Extending tools like Tyche to give developers concrete feedback on how to change their generators' distributions would potentially go a long way. Second, approaches using probabilistic programming languages, which can perform inference ahead of time to match desired distributions, are currently under-explored in the PBT space.

When addressing all of these problems, I believe the future of PBT is *interactive*. Problems like constrained generation and automatic tuning are *hard*—not just technically but computationally—and it will not always be possible to fully automate these processes. But that does not mean users need to do everything themselves. By carefully leveraging interaction, I believe we can produce interfaces that are both powerful and usable.

6.3 EVALUATION

With specification and generation handled, evaluation is the last critical step of the process. I hope to continue the work on Tyche from Chapter 5, exploring new and better ways to clarify test suite quality for developers. In this realm, the open problems lie in finding better evaluation metrics to measure and better ways to communicate those metrics.

Many of the under-explored evaluation metrics have to do with "coverage," although not always code coverage. Informal conversations with software developers indicate that black-box coverage metrics like *combinatorial coverage* [56; 100] need more exploration: developers want to understand how well they cover the possibility space of program behaviors without necessarily instrumenting their whole codebase. Of course, code coverage is also a useful metric (even if it does not necessarily predict bug-finding on its own [83]), and understanding it at a glance is not always straightforward. Better birds-eye visualizations of code coverage, built into interfaces like Tyche, are also an important frontier for PBT evaluation.

It is also critical that PBT evaluation be understandable by parties beyond the original test author. Indeed, in domains with strict acceptance requirements for software, it is often necessary to persuade auditors of the value of a property-based test. Tools like Tyche can help here too, providing data that can be compiled into reports and other documents that clarify the impact of a property as part of a test suite.

6.4 USABILITY

New tools for specification, generation, and evaluation should not be developed independently. A unified environment for PBT technologies would encourage adoption and provide a platform for user-driven research. I envision an integrated development environment (IDE) that helps throughout the PBT process.

Concretely, I propose extending the basic platform provided by Tyche to incorporate a variety of interactive tools that the community develops for PBT. This can include the above ideas—specification aids, generator automation, and evaluation metrics—in addition to other advances that benefit from living in and among user code. This unified platform will give would-be users an easy way to discover tools as they are developed, opening channels for feedback and impact. And as user research on PBT progresses, a unified set of tools can serve as a controlled environment in which to study interface design.

BIBLIOGRAPHY

[1] Umut A. Acar. Self-adjusting computation: (an overview). In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '09, pages 1–6, New York, NY, USA, January 2009. Association for Computing Machinery. ISBN 978-1-60558-327-3. doi: 10.1145/1480945.1480946. URL https://dl.acm.org/doi/10.1145/1480945.1480946.

[2] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the "Small Scope Hypothesis". Accessed online, 2002.

[3] Maurício Aniche, Christoph Treude, and Andy Zaidman. How Developers Engineer Test Cases: An Observational Study. *IEEE Transactions on Software Engineering*, 48(12):4925–4946, December 2022. ISSN 1939-3520. doi: 10.1109/TSE.2021.3129889. Conference Name: IEEE Transactions on Software Engineering.

[4] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, September 2006. Association for Computing Machinery. ISBN 978-1-59593-490-1. doi: 10.1145/1159789.1159792. URL http://doi.org/10.1145/1159789.1159792.

[5] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. Testing AUTOSAR software with QuickCheck. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–4, April 2015. doi: 10.1109/ICSTW.2015.7107466.

[6] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA, 2019. Internet Society. ISBN 978-1-891562-55-6. doi: 10.14722/ndss.2019.23412. URL https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_04A-3_Aschermann_paper.pdf.

[7] Karen Barrett-Wilt. The trials and tribulations of academic publishing – and Fuzz Testing, September 2021. URL https://www.cs.wisc.edu/2021/01/14/the-trials-and-tribulations-of-academic-publishing-and-fuzz-testing/. Publication Title: UW. Madison Department of Computer Sciences.

[8] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 179–190, New York, NY, USA, August 2015. Association for Computing Machinery. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786843. URL https://dl.acm.org/doi/10.1145/2786805.2786843.

[9] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer Testing in the IDE: Patterns, Beliefs, and Behavior. *IEEE Transactions on Software Engineering*, 45(3):261–284, March 2019. ISSN 1939-3520. doi: 10.1109/TSE.2017.2776152. Conference Name: IEEE Transactions on Software Engineering.

[10] Ann Blandford, Dominic Furniss, and Stephann Makri. Analysing Data. In Ann Blandford, Dominic Furniss, and Stephann Makri, editors, *Qualitative HCI Research: Going Behind the Scenes*, Synthesis Lectures on Human-Centered Informatics, pages 51–60. Springer International Publishing, Cham, 2016. ISBN 978-3-031-02217-3. doi: 10.1007/978-3-031-02217-3_5. URL https://doi.org/10.1007/978-3-031-02217-3_5.

[11] 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. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 836–850, New York, NY, USA, October 2021. Association for Computing Machinery. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483540. URL https://dl.acm.org/doi/10.1145/3477132.3483540.

[12] Rudy Matela Braquehais. Tools for Discovery, Refinement and Generalization of Functional Properties by Enumerative Testing. Publisher: University of York, October 2017. URL http://etheses.whiterose.ac.uk/19178/.

[13] Darren Brown, Margaret Burnett, Gregg Rothermel, Hamido Fujita, and Fumio Negoro. Generalizing WYSIWYT visual testing to screen transition languages. In *IEEE Symposium on Human Centric Computing Languages and Environments, 2003. Proceedings. 2003*, pages 203–210. IEEE, 2003.

[14] Janusz A Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964. doi: 10.1145/321239.321249. URL https://doi.org/10.1145/321239.321249. Publisher: ACM New York, NY, USA.

[15] Brian Burg, Richard Bailey, Amy J. Ko, and Michael D. Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, UIST '13, pages 473–484, New York, NY, USA, October 2013. Association for Computing Machinery. ISBN 978-1-4503-2268-3. doi: 10.1145/2501988.2502050. URL http://doi.org/10.1145/2501988.2502050.

[16] Margaret Burnett, Simone Stumpf, Jamie Macbeth, Stephann Makri, Laura Beckwith, Irwin Kwan, Anicia Peters, and William Jernigan. GenderMag: A Method for Evaluating Software's Gender Inclusiveness. *Interacting with Computers*, 28(6):760–787, November 2016. ISSN 0953-5438. doi: 10.1093/iwc/iwv046. URL https://doi.org/10.1093/iwc/iwv046.

[17] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 978-1-4503-4946-8. doi: 10.1145/3133956.3134020. URL https://doi.org/10.1145/3133956.3134020. event-place: Dallas, Texas, USA.

[18] Tsung-Hsiang Chang, Tom Yeh, and Robert C Miller. GUI testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1535–1544, 2010.

[19] Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine. PL and HCI: better together. *Communications of the ACM*, 64(8):98–106, August 2021. ISSN 0001-0782, 1557-7317. doi: 10.1145/3469279. URL https://dl.acm.org/doi/10.1145/3469279.

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

[21] Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: Guessing Formal Specifications Using Testing. In Gordon Fraser and Angelo Gargantini, editors, *Tests and Proofs*, pages 6–21, Berlin, Heidelberg, 2010. Springer. ISBN 978-3-642-13977-2. doi: 10.1007/978-3-642-13977-2_3.

[22] Koen Claessen, Jonas Duregård, and Michal H. Palka. Generating constrained random data with uniform distribution. *J. Funct. Program.*, 25, 2015. doi: 10.1017/S0956796815000143. URL http://dx.doi.org/10.1017/S0956796815000143.

[23] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996. ISSN 0360-0300. doi: 10.1145/242223.242257. URL https://dl.acm.org/doi/10.1145/242223.242257.

[24] Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design. *ACM Transactions on Computer-Human Interaction*, 28(4):28:1–28:53, July 2021. ISSN 1073-0516. doi: 10.1145/3452379. URL https://doi.org/10.1145/3452379.

[25] Arthur Corgozinho, Marco Valente, and Henrique Rocha. How Developers Implement Property-Based Tests. In *Conference: 39th International Conference on Software Maintenance and Evolution (ICSME 2023)*, September 2023.

[26] Ermira Daka and Gordon Fraser. A Survey on Unit Testing Practices and Problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211, November 2014. doi: 10.1109/ISSRE.2014.11. ISSN: 2332-6549.

[27] Natasha Danas, Tim Nelson, Lane Harrison, Shriram Krishnamurthi, and Daniel J. Dougherty. User Studies of Principled Model Finder Output. In Alessandro Cimatti and Marjan Sirjani, editors, *Software Engineering and Formal Methods*, Lecture Notes in Computer Science, pages 168–184, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66197-1. doi: 10.1007/978-3-319-66197-1_11.

[28] Matthew Davis, Sangheon Choi, Sam Estep, Brad Myers, and Joshua Sunshine. NaNofuzz: A Usable Tool for Automatic Test Generation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023,

pages 1114–1126, New York, NY, USA, November 2023. Association for Computing Machinery. ISBN 9798400703270. doi: 10.1145/3611643.3616327. URL https://dl.acm.org/doi/10.1145/3611643.3616327.

[29] Kyle Thomas Dewey. *Automated Black Box Generation of Structured Inputs for Use in Software Testing*. University of California, Santa Barbara, 2017. URL https://api.semanticscholar.org/CorpusID:190004227.

[30] Zac Hatfield Dodds. Zac Hatfield-Dodds Personal Communication, 2022. Published: current maintainer of Hypothesis (https://github.com/HypothesisWorks/hypothesis). Personal communication.

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

[32] Daniel Drew, Julie L Newcomb, William McGrath, Filip Maksimovic, David Mellis, and Björn Hartmann. The toastboard: Ubiquitous instrumentation and automated checking of breadboarded circuits. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 677–686, 2016.

[33] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. Wrex: A Unifed Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *\confchi*. \pubacm, 2020.

[34] Nicolas Dubien. fast-check, 2024. URL https://fast-check.dev/.

[35] Jonas Duregård, Patrik Jansson, and Meng Wang. Feat: functional enumeration of algebraic types. *ACM SIGPLAN Notices*, 47(12):61–72, September 2012. ISSN 0362-1340. doi: 10.1145/2430532.2364515. URL https://doi.org/10.1145/2430532.2364515.

[36] Tristan Dyer, Tim Nelson, Kathi Fisler, and Shriram Krishnamurthi. Applying cognitive principles to model-finding output: the positive value of negative information. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):79:1–79:29, April 2022. doi: 10.1145/3527323. URL https://dl.acm.org/doi/10.1145/3527323.

[37] Carl Eastlund. Quickcheck for Core, October 2015. URL https://blog.janestreet.com/quickcheck-for-core/.

[38] facebook. create-react-app, 2024. URL https://github.com/facebook/create-react-app.

[39] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++} : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020. URL https://www.usenix.org/conference/woot20/presentation/fioraldi.

[40] John Nathan Foster. *Bidirectional programming languages.* Ph.D., University of Pennsylvania, United States – Pennsylvania, 2009. URL https://www.proquest.com/docview/304986072/abstract/

11884B3FBDDB4DCFPQ/1. ISBN: 9781109710137.

[41] Andrew Gallant. BurntSushi/quickcheck, February 2024. URL https://github.com/BurntSushi/quickcheck. original-date: 2014-03-09T07:29:09Z.

[42] Tony Garnock-Jones, Mahdi Eslamimehr, and Alessandro Warth. Recognising and Generating Terms using Derivatives of Parsing Expression Grammars, January 2018. URL http://arxiv.org/abs/1801.10490. arXiv:1801.10490 [cs].

[43] Jean-Yves Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45:159–192, January 1986. ISSN 0304-3975. doi: 10.1016/0304-3975(86)90044-7. URL https://www.sciencedirect.com/science/article/pii/0304397586900447.

[44] Michele Giry. A Categorical Approach to Probability Theory. In *Categorical aspects of topology and analysis*, pages 68–85. Springer, 1982. URL https://link.springer.com/content/pdf/10.1007/BFb0092872.pdf.

[45] GitHub. Copilot, 2024. URL https://github.com/features/copilot.

[46] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Transactions on Computer-Human Interaction*, 22(2):7:1–7:35, March 2015. ISSN 1073-0516. doi: 10.1145/2699751. URL https://dl.acm.org/doi/10.1145/2699751.

[47] Elena L. Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. Visualizing API Usage Examples at Scale. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 1–12, New York, NY, USA, April 2018. Association for Computing Machinery. ISBN 978-1-4503-5620-6. doi: 10.1145/3173574.3174154. URL https://dl.acm.org/doi/10.1145/3173574.3174154.

[48] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215, New York, NY, USA, June 2008. Association for Computing Machinery. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375607. URL https://doi.org/10.1145/1375581.1375607.

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

[50] Harrison Goldstein. Ungenerators. In *ICFP Student Research Competition*, 2021. URL https://harrisongoldste.in/papers/icfpsrc21.pdf.

[51] Harrison Goldstein. Parsing Randomness: Free Generators Development, October 2022. URL https://doi.org/10.5281/zenodo.7086231. Publisher: Zenodo.

[52] Harrison Goldstein. Property-Based Testing in Practice: Codebook, December 2023. URL https://zenodo.org/doi/10.5281/zenodo.10407686.

[53] Harrison Goldstein. Reflecting on Random Generation: Reflective Generators Development, 2023. URL https://zenodo.org/record/7988049. Publisher: Zenodo.

[54] Harrison Goldstein. Tyche: In Situ Exploration of Random Testing Effectiveness (Demo), October 2023. URL https://harrisongoldste.in/papers/uist23.pdf.

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

[56] Harrison Goldstein, John Hughes, Leonidas Lampropoulos, and Benjamin C. Pierce. Do Judge a Test by its Cover. In Nobuko Yoshida, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 264–291, Cham, 2021. Springer International Publishing. ISBN 978-3-030-72019-3. doi: 10.1007/978-3-030-72019-3_10. URL https://doi.org/10.1007/978-3-030-72019-3_10.

[57] Harrison Goldstein, Joseph W Cutler, Adam Stein, Benjamin C Pierce, and Andrew Head. Some Problems with Properties. In *Workshop on Human Aspects of Types and Reasoning Assistants*, volume 1, December 2022. URL https://harrisongoldste.in/papers/hatra2022.pdf.

[58] Harrison Goldstein, Samantha Frohlich, Meng Wang, and Benjamin C. Pierce. Reflecting on Random Generation. In *Proc. ACM Program. Lang.*, volume 7, pages 322–355, Seattle, WA, USA, August 2023. Association for Computing Machinery. doi: 10.1145/3607842. URL https://doi.org/10.1145/3607842.

[59] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, volume 187 of *ICSE '24*, pages 1–13, Lisbon, Portugal, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639581. URL https://doi.org/10.1145/3597503.3639581.

[60] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *Future of Software Engineering Proceedings*, FOSE 2014, pages 167–181, New York, NY, USA, May 2014. Association for Computing Machinery. ISBN 978-1-4503-2865-4. doi: 10.1145/2593882.2593900. URL https://doi.org/10.1145/2593882.2593900.

[61] Ben Greenman, Sam Saarinen, Tim Nelson, and Shriram Krishnamurthi. Little Tricky Logic: Misconceptions in the Understanding of LTL. *The Art, Science, and Engineering of Programming*, 7 (2):7, October 2022. ISSN 2473-7321. doi: 10.22152/programming-journal.org/2023/7/7. URL https://programming-journal.org/2023/7/7.

[62] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Test confessions: A study of testing practices for plug-in systems. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 244–254, June 2012. doi: 10.1109/ICSE.2012.6227189. ISSN: 1558-1225.

[63] Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. Perception and Practices of Differential Testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 71–80, May 2019. doi: 10.1109/ICSE-SEIP.2019.00016.

[64] J. A. Hartigan and B. Kleiner. Mosaics for Contingency Tables. In William F. Eddy, editor, *Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface*, pages 268–273, New York, NY, 1981. Springer US. ISBN 978-1-4613-9464-8. doi: 10.1007/978-1-4613-9464-8_37.

[65] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. Design as Exploration: Creating Interface Alternatives through Parallel Authoring and Runtime Tuning. In *\confuist*, pages 91–100. \pubacm, 2008.

[66] Zac Hatfield-Dodds. HypoFuzz, 2022. URL https://hypofuzz.com/.

[67] Zac Hatfield-Dodds. Ghostwriting tests for you — Hypothesis 6.82.0 documentation, 2023. URL https://hypothesis.readthedocs.io/en/latest/ghostwriter.html.

[68] Zac Hatfield-Dodds. Observability Tools & Hypothesis 6.99.13, 2024. URL https://hypothesis.readthedocs.io/en/latest/observability.html.

[69] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A Ground-Truth Fuzzing Benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3):49:1–49:29, June 2021. doi: 10.1145/3428334. URL http://doi.org/10.1145/3428334.

[70] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *\conflas*, pages 89–98. \pubacm, 2017.

[71] Constance Heitmeyer. On the Need for Practical Formal Methods. Technical report, Naval Research Lab, Washington, DC, January 1998. URL https://apps.dtic.mil/sti/citations/ADA465485. Section: Technical Reports.

[72] Brian Hempel, Justin Lubin, and Ravi Chugh. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, UIST '19, pages 281–292, New York, NY, USA, October 2019. Association for Computing Machinery. ISBN 978-1-4503-6816-2. doi: 10.1145/3332165.3347925. URL http://doi.org/10.1145/3332165.3347925.

[73] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. Augmenting Code with In Situ Visualizations to Aid Program Understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 1–12, New York, NY, USA, April 2018. Association for Computing Machinery. ISBN 978-1-4503-5620-6. doi: 10.1145/3173574.3174106. URL https://dl.acm.org/doi/10.1145/3173574.3174106.

[74] Fred Hohman, Andrew Head, Rich Caruana, Robert DeLine, and Steven M. Drucker. Gamut: A

Design Probe to Understand How Data Scientists Understand Machine Learning Models. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, pages 1–13, New York, NY, USA, May 2019. Association for Computing Machinery. ISBN 978-1-4503-5970-2. doi: 10.1145/3290605.3300809. URL http://doi.org/10.1145/3290605.3300809.

[75] Fred Hohman, Kanit Wongsuphasawat, Mary Beth Kery, and Kayur Patel. Understanding and Visualizing Data Iteration in Machine Learning. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, pages 1–13, New York, NY, USA, April 2020. Association for Computing Machinery. ISBN 978-1-4503-6708-0. doi: 10.1145/3313831.3376177. URL https://doi.org/10.1145/3313831.3376177.

[76] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, page 38, USA, August 2012. USENIX Association.

[77] Paul Holser. pholser/junit-quickcheck, January 2024. URL https://github.com/pholser/junit-quickcheck. original-date: 2010-10-18T22:33:36Z.

[78] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Scaling Exact Inference for Discrete Probabilistic Programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–31, November 2020. ISSN 2475-1421. doi: 10.1145/3428208. URL http://arxiv.org/abs/2005.09089. arXiv:2005.09089 [cs].

[79] John Hughes. QuickCheck Testing for Fun and Profit. In Michael Hanus, editor, *Practical Aspects of Declarative Languages*, Lecture Notes in Computer Science, pages 1–32, Berlin, Heidelberg, 2007. Springer. ISBN 978-3-540-69611-7. doi: 10.1007/978-3-540-69611-7_1.

[80] John Hughes. Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Lecture Notes in Computer Science, pages 169–186. Springer International Publishing, Cham, 2016. ISBN 978-3-319-30936-1. doi: 10.1007/978-3-319-30936-1_9. URL https://doi.org/10.1007/978-3-319-30936-1_9.

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

[82] John Hughes, Benjamin C. Pierce, Thomas Arts, and Ulf Norell. Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 135–145, April 2016. doi: 10.1109/ICST.2016.37.

[83] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, May 2014. Association for Computing Machinery. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568271. URL https://dl.acm.org/doi/10.1145/2568225.2568271.

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

[85] JetBrains. Python Developers Survey 2021 Results, 2021. URL https://lp.jetbrains.com/python-developers-survey-2021/. Publication Title: JetBrains: Developer Tools for Professionals and Teams.

[86] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38, 2023. Publisher: ACM New York, NY.

[87] joeyespo. pytest-watch, 2024. URL https://github.com/joeyespo/pytest-watch.

[88] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA, November 2005. Association for Computing Machinery. ISBN 978-1-58113-993-8. doi: 10.1145/1101908.1101949. URL https://dl.acm.org/doi/10.1145/1101908.1101949.

[89] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665, 2014.

[90] Hyeonsu Kang and Philip J. Guo. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 737–745, New York, NY, USA, October 2017. Association for Computing Machinery. ISBN 978-1-4503-4981-9. doi: 10.1145/3126594.3126632. URL http://doi.org/10.1145/3126594.3126632.

[91] Marcel R Karam and Trevor J Smedley. A testing methodology for a dataflow based visual programming language. In *Proceedings IEEE Symposia on Human-Centric Computing Languages and Environments (Cat. No. 01TH8587)*, pages 280–287. IEEE, 2001.

[92] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, pages 140–151, New York, NY, USA, October 2020. Association for Computing Machinery. ISBN 978-1-4503-7514-6. doi: 10.1145/3379337.3415842. URL http://doi.org/10.1145/3379337.3415842.

[93] Oleg Kiselyov. Typed Tagless Final Interpreters. In Jeremy Gibbons, editor, *Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, Lecture Notes in Computer Science, pages 130–174. Springer, Berlin, Heidelberg, 2012. ISBN 978-3-642-32202-0. doi: 10.1007/978-3-642-32202-0_3. URL https://doi.org/10.1007/978-3-642-32202-0_3.

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

[95] Ed Kmett. free: Haskell Package, 2023. URL //hackage.haskell.org/package/free.

[96] Ed Kmett. Control.Lens, 2024. URL https://hackage.haskell.org/package/lens-5.2.3/docs/Control-Lens.html.

[97] Pavneet Singh Kochhar, David Lo, Julia Lawall, and Nachiappan Nagappan. Code Coverage and Postrelease Defects: A Large-Scale Study on Open Source Projects. *IEEE Transactions on Reliability*, 66(4):1213–1228, December 2017. ISSN 1558-1721. doi: 10.1109/TR.2017.2727062. URL https://ieeexplore.ieee.org/abstract/document/8031982. Conference Name: IEEE Transactions on Reliability.

[98] Herb Krasner. The cost of poor software quality in the US: A 2022 report, December 2022.

[99] Shriram Krishnamurthi and Tim Nelson. The Human in Formal Methods. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods – The Next 30 Years*, Lecture Notes in Computer Science, pages 3–10, Cham, 2019. Springer International Publishing. ISBN 978-3-030-30942-8. doi: 10.1007/978-3-030-30942-8_1.

[100] D. Richard Kuhn, James M. Higdon, James Lawrence, Raghu Kacker, and Yu Lei. Combinatorial Methods for Event Sequence Testing. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, pages 601–609. IEEE Computer Society, 2012. doi: 10.1109/ICST.2012.147. URL https://doi.org/10.1109/ICST.2012.147.

[101] S. Kullback and R. A. Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951. ISSN 0003-4851. URL https://www.jstor.org/stable/2236703. Publisher: Institute of Mathematical Statistics.

[102] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. Beginner's Luck: a language for property-based generators. *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 114–129, 2017. URL http://dl.acm.org/citation.cfm?id=3009868.

[103] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017. URL https://dl.acm.org/doi/10.1145/3158133. Publisher: ACM New York, NY, USA.

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

[105] J Lawrance, Steven Clarke, Margaret Burnett, and Gregg Rothermel. How well do professional developers test with code coverage visualizations? an empirical study. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 53–60. IEEE, 2005.

[106] Daan Leijen and Erik Meijer. Parsec: A practical parser library. *Electronic Notes in Theoretical Computer Science*, 41(1):1–20, 2001.

[107] Sorin Lerner. Projection boxes: On-the-fly reconfigurable visualization for live programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–7, 2020.

[108] Vladimir I Levenshtein and others. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966. URL https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf. Issue: 8.

[109] N.G. Leveson and C.S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993. ISSN 1558-0814. doi: 10.1109/MC.1993.274940. URL https://ieeexplore.ieee.org/abstract/document/274940?casa_token=-dgV6qcuzb0AAAAA:kD7kj4RryXVlXmOwYADkyM61GsWTXi-Of9NRu9w96CVRilZZh72yDiYEvlg8Oad2Siq2LVHd. Conference Name: Computer.

[110] J. Lin. Divergence measures based on the Shannon entropy. *IEEE Transactions on Information Theory*, 37(1):145–151, January 1991. ISSN 1557-9654. doi: 10.1109/18.61115. Conference Name: IEEE Transactions on Information Theory.

[111] Andreas Löscher and Konstantinos Sagonas. Targeted property-based testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 46–56, New York, NY, USA, July 2017. Association for Computing Machinery. ISBN 978-1-4503-5076-1. doi: 10.1145/3092703.3092711. URL http://doi.org/10.1145/3092703.3092711.

[112] David R. MacIver and Alastair F. Donaldson. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer (Tool Insights Paper). In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:27, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. ISBN 978-3-95977-154-2. doi: 10.4230/LIPIcs.ECOOP.2020.13. URL https://drops.dagstuhl.de/opus/volltexte/2020/13170. ISSN: 1868-8969.

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

[114] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: unit testing with mock objects. *Extreme programming examined*, pages 287–301, 2000. Publisher: Addison-Wesley Boston, MA, USA.

[115] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for Haskell. *ACM SIGPLAN Notices*, 45(11):37–48, September 2010. ISSN 0362-1340. doi: 10.1145/

2088456.1863529. URL https://doi.org/10.1145/2088456.1863529.

[116] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. User Interaction Models for Disambiguation in Programming by Example. In *\confuist*, pages 291–301. \pubacm, 2015.

[117] Will McGrath, Daniel Drew, Jeremy Warner, Majeed Kazemitabaar, Mitchell Karchemsky, David Mellis, and Björn Hartmann. Bifröst: Visualizing and checking behavior of embedded systems across hardware and software. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 299–310, 2017.

[118] William McGrath, Jeremy Warner, Mitchell Karchemsky, Andrew Head, Daniel Drew, and Bjoern Hartmann. Wifröst: Bridging the information gap for debugging of networked embedded systems. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, pages 447–455, 2018.

[119] Patrick E McKnight and Julius Najab. Mann-Whitney U Test. *The Corsini encyclopedia of psychology*, pages 1–1, 2010. Publisher: Wiley Online Library.

[120] Microsoft. Visual Studio Code, 2024. URL https://code.visualstudio.com/.

[121] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, December 1990. ISSN 0001-0782. doi: 10.1145/96267.96279. URL https://dl.acm.org/doi/10.1145/96267.96279.

[122] Robert C Miller and Brad A Myers. Outlier finding: Focusing user attention on possible errors. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 81–90, 2001.

[123] Minsky. Testing with expectations, December 2015. URL https://blog.janestreet.com/testing-with-expectations/.

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

[125] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991. ISSN 0890-5401. doi: 10.1016/0890-5401(91)90052-4. URL https://www.sciencedirect.com/science/article/pii/0890540191900524.

[126] Aaditya Naik, Jonathan Mendelson, Nathaniel Sands, Yuepeng Wang, Mayur Naik, and Mukund Raghothaman. Sporq: An interactive environment for exploring code using query-by-example. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 84–99, 2021.

[127] Tim Nelson, Elijah Rivera, Sam Soucie, Thomas Del Vecchio, John Wrenn, and Shriram Krishnamurthi. Automated, Targeted Testing of Property-Based Testing Predicates. *The Art, Science, and Engineering of Programming*, 6(2):10, November 2021. ISSN 2473-7321. doi: 10.22152/programming-journal.org/2022/6/10. URL http://arxiv.org/abs/2111.10414. arXiv:2111.10414 [cs].

[128] Wode Ni, Joshua Sunshine, Vu Le, Sumit Gulwani, and Titus Barik. recode: A lightweight find-and-replace interaction in the ide for transforming code by example. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 258–269, 2021.

[129] Rickard Nilsson. ScalaCheck, 2024. URL https://scalacheck.org/.

[130] Liam O'Connor and Oskar Wickström. Quickstrom: property-based acceptance testing with LTL specifications. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 1025–1038, New York, NY, USA, June 2022. Association for Computing Machinery. ISBN 978-1-4503-9265-5. doi: 10.1145/3519939.3523728. URL https://doi.org/10.1145/3519939.3523728.

[131] Stephen Oney and Brad Myers. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 105–108, September 2009. doi: 10.1109/VLHCC.2009.5295287. ISSN: 1943-6106.

[132] OpenAI. GPT-4 Technical Report, 2023. _eprint: 2303.08774.

[133] Otter.ai. Otter.ai - Voice Meeting Notes & Real-time Transcription, 2023. URL https://otter.ai/.

[134] Micha\l H. Pa\lka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 91–97, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0592-1. doi: 10.1145/1982595.1982615. URL http://doi.acm.org/10.1145/1982595.1982615. event-place: Waikiki, Honolulu, HI, USA.

[135] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman. Mutation Testing Advances: An Analysis and Survey. *Advances in Computers*, January 2018. ISSN 0065-2458. URL http://dx.doi.org/10.1016/bs.adcom.2018.03.015.

[136] Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational Property-Based Testing. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 325–343, Cham, 2015. Springer International Publishing. ISBN 978-3-319-22102-1. doi: 10.1007/978-3-319-22102-1_22.

[137] Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. Computing correctly with inductive relations. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 966–980, New York, NY, USA, June 2022. Association for Computing Machinery. ISBN 978-1-4503-9265-5. doi: 10.1145/3519939.3523707. URL https://doi.org/10.1145/3519939.3523707.

[138] Goran Petrovic and Marko Ivankovic. State of Mutation Testing at Google. In *Proceedings of the 40th International Conference on Software Engineering 2017 (SEIP)*, 2018.

[139] Tomáš Petříček. Encoding monadic computations in C# using iterators, 2009. URL http://tomasp.net/academic/papers/iterators/iterators.pdf.

[140] M. Pickering, J. Gibbons, and N. Wu. Profunctor optics: Modular data accessors. *Art, Science, and Engineering of Programming*, 1(2), 2017. ISSN 2473-7321. URL https://ora.ox.ac.uk/objects/uuid:9989be57-a045-4504-b9d7-dc93fd508365. Publisher: Aspect-Oriented Software Association.

[141] Lee Pike. SmartCheck: automatic and efficient counterexample reduction and generalization. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*, Haskell '14, pages 53–64, New York, NY, USA, September 2014. Association for Computing Machinery. ISBN 978-1-4503-3041-1. doi: 10.1145/2633357.2633365. URL https://doi.org/10.1145/2633357.2633365.

[142] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program Synthesis from Polymorphic Refinement Types, April 2016. URL http://arxiv.org/abs/1510.08419. arXiv:1510.08419 [cs].

[143] Kevin Pu, Rainey Fu, Rui Dong, Xinyu Wang, Yan Chen, and Tovi Grossman. SemanticOn: Specifying content-based semantic conditions for web automation programs. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, pages 1–16, 2022.

[144] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. Quickly generating diverse valid test inputs with reinforcement learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 1410–1421, New York, NY, USA, October 2020. Association for Computing Machinery. ISBN 978-1-4503-7121-6. doi: 10.1145/3377811.3380399. URL http://doi.org/10.1145/3377811.3380399.

[145] Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. Towards making formal methods normal: meeting developers where they are, October 2020. URL http://arxiv.org/abs/2010.16345. arXiv:2010.16345 [cs].

[146] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. Exploratory and Live, Programming and Coding. *The Art, Science, and Engineering of Programming*, 3(1), 2018. Publisher: Aspect-Oriented Software Association (AOSA).

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

[148] Gregg Rothermel, Lixin Li, Christopher DuPuis, and Margaret Burnett. What you see is what you test: A methodology for testing form-based visual programs. In *Proceedings of the 20th international conference on Software engineering*, pages 198–207. IEEE, 1998.

[149] RTI. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 1, 2002.

[150] Adam Rule, Ian Drosos, Aurélien Tabard, and James D. Hollan. Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding. In *\confcscw*. \pubacm, 2018.

[151] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. *ACM SIGPLAN Notices*, 44(2):37–48, September 2008. ISSN 0362-1340. doi: 10.1145/1543134.1411292. URL http://doi.org/10.1145/1543134.1411292.

[152] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1): 341–350, January 2017. ISSN 1941-0506. doi: 10.1109/TVCG.2016.2599030. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[153] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C Pierce, and Leonidas Lampropoulos. Etna: An Evaluation Platform for Property-Based Testing (Experience Report). *Proc. ACM Program. Lang.*, 7, 2023. doi: 10.1145/3607860.

[154] Nischal Shrestha, Titus Barik, and Chris Parnin. Unravel: A fluent code explorer for data wrangling. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 198–207, 2021.

[155] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. Inputs from Hell: Learning Input Distributions for Grammar-Based Test Generation. *IEEE Transactions on Software Engineering*, 2020. doi: 10.1109/TSE.2020.3013716. URL https://doi.org/10.1109/TSE.2020.3013716. Publisher: IEEE.

[156] C Spearman. The Proof and Measurement of Association between Two Things. *American Journal of Psychology*, 15:72–101, 1904. Publisher: University of Illinois Press, etc.

[157] Prashast Srivastava and Mathias Payer. Gramatron: effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, pages 244–256, New York, NY, USA, July 2021. Association for Computing Machinery. ISBN 978-1-4503-8459-9. doi: 10.1145/3460319.3464814. URL https://doi.org/10.1145/3460319.3464814.

[158] Jacob Stanley. Hedgehog will eat all your bugs, 2017. URL https://hedgehog.qa/.

[159] Dominic Steinhöfel and Andreas Zeller. Input invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pages 583–594, New York, NY, USA, November 2022. Association for Computing Machinery. ISBN 978-1-4503-9413-0. doi: 10.1145/3540250.3549139. URL https://doi.org/10.1145/3540250.3549139.

[160] Donald Stewart, Koen Claessen, Nick Smallbone, and Simon Marlow. Test.QuickCheck — hackage.haskell.org, 2024. URL https://hackage.haskell.org/package/QuickCheck-2.14.3/docs/

Test-QuickCheck.html#v:label.

[161] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live API Documentation. In *\conficse*, pages 643–652. \pubacm, 2014.

[162] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Björn Hartmann. TraceDiff: Debugging unexpected code behavior using trace divergences. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 107–115, October 2017. doi: 10.1109/VLHCC.2017.8103457. URL https://ieeexplore.ieee.org/abstract/document/8103457. ISSN: 1943-6106.

[163] Steven L. Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, June 1990. ISSN 1045-926X. doi: 10.1016/S1045-926X(05)80012-6. URL https://www.sciencedirect.com/science/article/pii/S1045926X05800126.

[164] Vera Trnková, Jiř\'ı\ Adámek, Václav Koubek, and Jan Reiterman. Free algebras, input processes and free monads. *Commentationes Mathematicae Universitatis Carolinae*, 016:339–351, 1975. URL http://dml.mathdoc.fr/item/105628.

[165] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Elsevier, July 2010. ISBN 978-0-08-046648-4.

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

[167] Jordan Walke. React, 2024. URL https://react.dev/.

[168] April Yi Wang, Will Epperson, Robert A DeLine, and Steven M Drucker. Diff in the loop: Supporting data comparison in exploratory data analysis. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pages 1–10, 2022.

[169] Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J Ko. Falx: Synthesis-powered visualization authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2021.

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

[171] Ian Ward. JSON Lines, 2024. URL https://jsonlines.org/.

[172] Wikipedia. Name server — Wikipedia, The Free Encyclopedia, 2024. URL http://en.wikipedia.org/

w/index.php?title=Name%20server&oldid=1215654110.

[173] Wikipedia. Red–black tree — Wikipedia, The Free Encyclopedia, 2024. URL http://en.wikipedia.org/w/index.php?title=Red%E2%80%93black%20tree&oldid=1215636980.

[174] Aaron Wilson, Margaret Burnett, Laura Beckwith, Orion Granatir, Ledah Casburn, Curtis Cook, Mike Durham, and Gregg Rothermel. Harnessing curiosity to increase correctness in end-user programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '03, pages 305–312, New York, NY, USA, April 2003. Association for Computing Machinery. ISBN 978-1-58113-630-2. doi: 10.1145/642611.642665. URL https://dl.acm.org/doi/10.1145/642611.642665.

[175] J. Wing, D. Jackson, and C. B. Jones. Formal Methods Light. *Computer*, 29(04):20–22, April 1996. ISSN 1558-0814. doi: 10.1109/MC.1996.10038. Place: Los Alamitos, CA, USA Publisher: IEEE Computer Society.

[176] John Wrenn, Tim Nelson, , and Shriram Krishnamurthi. Using Relational Problems to Teach Property-Based Testing. *The art science and engineering of programming*, 5(2), January 2021. doi: 10.22152/programming-journal.org/2021/5/9. URL https://par.nsf.gov/biblio/10204213-using-relational-problems-teach-property-based-testing.

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

[178] Litao Yan, Elena L. Glassman, and Tianyi Zhang. Visualizing Examples of Deep Neural Networks at Scale. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, pages 1–14, New York, NY, USA, May 2021. Association for Computing Machinery. ISBN 978-1-4503-8096-6. doi: 10.1145/3411764.3445654. URL https://dl.acm.org/doi/10.1145/3411764.3445654.

[179] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294, 2011. doi: 10.1145/1993498.1993532. URL http://www.cs.utah.edu/~regehr/papers/pldi11-preprint.pdf.

[180] Michał Zalewski. American Fuzzy Lop (AFL), November 2022. URL https://github.com/google/AFL.

[181] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L Glassman. Interactive program synthesis by augmented examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 627–648, 2020.

[182] Valerie Zhao, Lefan Zhang, Bo Wang, Michael L Littman, Shan Lu, and Blase Ur. Understanding trigger-action programs through novel visualizations of program differences. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–17, 2021.

[183] Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. Functional programming for modular

Bayesian inference. *Proceedings of the ACM on Programming Languages*, 2(ICFP):83:1–83:29, July 2018. doi: 10.1145/3236778. URL https://doi.org/10.1145/3236778.