CCF Core: Medium: Usable Property-Based Testing

Project Description

Testing plays a vital role in modern software development processes, contributing crucially to robustness and overall quality—as well as to overall development costs. It comes in many styles—unit testing, integration testing, performance testing, stress testing, accessibility testing, etc.—supported by many sorts of tools, with yet more advanced tools and techniques continually being developed, studied, and applied.

One such technique, *property-based testing* (PBT), has been enthusiastically taken up by the functional programming research community, since its introduction in the QuickCheck library [1] in Haskell, and is beginning to make serious inroads beyond academia. PBT is sometimes explained as "formal specification without formal verification"—a developer characterizes a the desired behavior of some piece of code at a high level, in the form of executable *properties*, which are simply Boolean-valued functions; the code is then validated against these properties by running it over and over with a large number of automatically generated test cases. PBT tools thus give programmers an efficient way of testing their software's behavior with a comprehensiveness that is often not possible with alternative tools.

PBT's combination of rich, high-level specification with easy, mostly automatic validation has proved effective at identifying subtle bugs in telecommunications software [2], replicated file [3] and key-value stores [4], automotive software [5], and a range of other real-world systems [6]. With support now available in all major programming languages, PBT has begun to make significant inroads in the broader software industry—for example, the developers of the Hypothesis library for Python estimate that it has on the order of half a million users [7].

After all these successes, one might wonder if the research community has already addressed all of the challenges that could limit PBT's adoption in the broader software industry, but it seems the answer is no. In an ongoing need-finding study with users of OCaml's QuickCheck testing tool at Jane Street Capital, we found consistent enthusiasm for PBT—participants called it it "obviously valuable" [Participant P1], built their own libraries for it when standard ones were not available in their development context [P8, P21], and suggested that "everyone" at the company should use it [P20]. But we also found challenging new research questions based on developer concerns. Specifically, the developers we have spoken with so far (a) sometimes struggle to identify all the situations where properties are readily accessible for testing, (b) they complain that it can be difficult to write random generators for data structures with complex invariants or to tune generator distributions to adequately explore the space of interesting test cases, (c) they find it hard to understand whether their properties and generators are effectively testing their software, and (d) they are disappointed by the paucity of educational and training materials for introducing newcomers to PBT.

Solutions to these challenges will demand insights from both the programming languages (PL) and human-computer interaction (HCI) communities. Chasins et al. [8] argue that PL+HCI is a "sweet spot" that marries "need-finding techniques [to] identify ... programmers' current and future pain points" with tools that "help [programmers] write safe and correct programs." We wholeheartedly agree. The team behind this proposal is uniquely positioned to advance the state of PL+HCI: PI Head is deeply embedded in the HCI community and recently co-founded the new HCI group of the University of Pennsylvania, while PI Pierce is has published in every corner of the PL world and has decades of experience in both researching and teaching about programming languages. This proposal will be the first of many PL+HCI projects that the University of Pennsylvania Computing and Information Science department undertakes.

We propose a comprehensive, interdisciplinary research program that will bring the combined power of PL and HCI approaches to bear, accelerating PBT's transition into practice.

- Our ongoing study (described in more detail below) is just the first step in our formative research. We will further explore the challenges PBT faces in industry by conducting three additional studies—two survey studies and one observation study of PBT users *in situ* (§1).
- We will address challenges in property specification by developing a conceptual framework of high-impact PBT use cases, along with tools that enhance PBT in common use-cases like model-based testing and extend that set of use cases with the ability to write properties over logged output (§2).
- We will address challenges in test-case generation with new domain-specific languages for expressing generators that enable novel algorithms for PBT tasks like generation of precondition-satisfying values, test input mutation, and example-based generator tuning. (§3).
- We will address gaps in validation of testing effectiveness by providing users with powerful new ways

of interacting with their generators and test suites that give them visibility into the bug-finding power of their tests (§4).
- We will create comprehensive educational resources for training developers and university students in best practices for PBT (§5).

We conclude the proposal with a concrete outline of our work plan (§6) and discussion of the broader impacts of the proposed work (§7)

The projects we undertake will be motivated by foundational need-finding research in the software industry, including, but not limited to, the aforementioned study with Jane Street. Some projects will have a clear PL flavor, focusing on domain-specific languages and type-based tools that will make PBT tools more powerful and flexible; but we will motivate and evaluate them with techniques from HCI. Other projects will focus on the HCI elements of PBT, building better interfaces and experiences for developers; yet their implementation will draw from our wealth of PL knowledge. In this way, we craft a holistic research agenda around PBT that is sure to be both well motivated and well founded.

Orientation: Property-Based Testing

Before discussing our motivating study, we briefly explain what PBT is and why it is important.

Popularized by QuickCheck [9] in Haskell, PBT is a form of random testing [10] where users write executable functions, usually in the same language as their System Under Test, which act as partial specifications of a function under test. For example, a user might write the following property of the `insert` function for an implementation of a binary search tree (BST):

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

This property takes only a single line to express, yet it can be used to validate a limitless number of input-output pairs. Concretely, it is a simple function that takes two arguments and returns a Boolean value: true if the property is satisfied, and false if it is not. The function `prop_insertCorrect` checks that an insert operation on a binary search tree preserves the binary ordering of the tree; it states that, given an arbitrary tree `t` and an integer `x` to insert into that tree, if the original tree satisfies the binary search tree invariant ("`isBST t`"), then it should remain a BST following the insertion of `x` ("`isBST (insert x t)`)".

Given a property like this one, the PBT tool chooses large number of inputs and checks that the property evaluates to `True` for each input; any input that causes the property to fail is reported as a *counterexample*. This proposal is on property-based *random* testing [10], in which these inputs are chosen randomly. There are other approaches, including enumerative test-case generation [11, 12, etc.] and model checking, but random generation remains the dominant approach in the PBT space. Its surprising effectiveness is often attributed to the "combinatorial" nature of larger test cases: bugs can often be exposed by test inputs with a specific combination of features, independent of whatever other features may be present—for example, by any sequence of API calls that includes calls to four particular functions in a particular order, even if these calls happens to be interleaved with other API calls. This means that testing with one big structure has the effect of testing with exponentially many smaller sub-structures. Both practical experience and theoretical arguments [13] suggest that this effect is quite powerful—i.e., we can discover small counterexamples by generating a few larger test inputs containing combinatorially many diverse sub-structures rather than systematically enumerating test inputs beginning with the smallest ones.

From the perspective of a user, applying PBT happens in a series of steps: (1) Define one or more properties that should be true of the program under test. (2) Design (or otherwise obtain) *random input generators* for the values that the properties take as input. (3) Check the properties with generated inputs, using infrastructure provided by the PBT framework. And (4) if counter-examples are found, make sense of them and determine the source of the bug. Each of these steps presents opportunities to improve the user experience of PBT, as we shall see below.

Why go to the trouble of PBT, when more straightforward example testing is the de-facto standard in the software industry? First and foremost, PBT has the potential to be much more thorough than user-defined examples. As mentioned previously, PBT has an impressive track-record uncovering bugs that other approaches to testing had failed to find [2, 3, 4, 5, 6]. But PBT is more than just thorough—it is actually more general than example-based testing. Wrenn et al. [14] point out that example-based testing of programs

like that implement relations (e.g., topological sort, which may produce one of a number of potentially correct results) is impossible to do faithfully; a property-based specification is a much better choice. PBT is also obviously preferable if formal properties already exist; this is the use-case for QuickChick [15], which implements PBT in the Coq proof assistant. Finally, PBT can act as superior documentation: participants in our foundational study (P5, P21) talked at length about properties being an ideal way to communicate what a program is supposed to do.

With so many potential advantages, it is clear that PBT should be a tool that software developers have readily at their fingertips. We plan to put it there.

Motivation: A Formative Study of PBT in Industry

The technical goals of this proposal are grounded in early findings from an ongoing need-finding study asking *How can the research community make PBT more valuable for software developers?* In this study, we have conducted semistructured interviews with PBT stakeholders at Jane Street Capital—both developers who have had exposure to PBT as well as developers of PBT tools.

Interview studies are a common tool for early-stage need finding in HCI research, yielding highly informative stories about people, the tasks they need to accomplish, and the tools they use to accomplish them. The goal is to identify a broad set of opportunities for improving PBT tools, grounded in concrete, detailed, authentic stories from developers.

Jane Street is among of the world's largest financial market-makers; its 1700 employees include around 700 software developers. A number of features make it an attractive setting for this study. Most importantly, PBT is already well established at Jane Street, so there is a large population of people with well-informed opinions on its benefits and challenges. Also, Jane Street famously builds almost all of its software in OCaml, a mostly functional programming language with strong support for rich static typing and modularity, plus a well-engineered PBT tool. This unified ecosystem allows us to control for a number of potentially confounding factors: all of the developers have access to the same tools, libraries, language-level programming abstractions, house coding rules, social conventions, etc.

There is a danger that findings from a study at a single firm with a homogeneous OCaml-based testing ecosystem may not generalize outside of this ecosystem. Part of our proposed work is to carry out further user studies targeting a broader and more diverse community (§1). However, Jane Street itself is quite diverse in the sorts of software it builds, including real-time trading, quantitative algorithms, networked systems, and hardware description code [16], and our study participants come with a wide variety of experiences and requirements.

Our interview instrument—the script used to loosely structure the interviews—and our analysis process were developed and tested through a prototype study with seven users of the Hypothesis PBT toolkit, recruited over Twitter. This analysis yielded a conceptual framework of barriers to usage of PBT, which we published recently at the SPLASH HATRA workshop, one of the key venues for emerging work on human aspects of formal methods tools and type systems [17].

At the time of writing, the full complement of 30 interviews and a preliminary round of analysis have been completed; full-scale analysis will begin in 2023.

Organizing Themes The main theoretical output from this qualitative study will be a collection of *Themes* summarizing and categorizing our observations about the ways Jane Street developers use PBT, what they need from it, and how the research community be able to help. While deep analysis of the interview transcripts remains to be done, several themes are already apparent in our real-time notes, and these form the backbone of the present proposal.

A first set of themes revolves around the **specifications** that developers test. Since PBT is often described as a kind of lightweight formal method, one might imagine that a central challenge would be coming up with the right specifications. Indeed, an earlier pilot study of PBT users in Python [18] that we ran to prepare for the Jane Street study concluded just that. But at Jane Street itself we actually heard very little about difficulties coming up with properties; rather, most participants described applying PBT in *High-Leverage Scenarios* where properties were already available or straightforward to invent. For example, P9 pointed out that PBT is particularly easy to apply when one has "a really good abstraction with a complicated implementation." Several participants (P3, P15, P20, P22), when asked to speculate, guessed that 80–100% of

Jane Street developers write programs like this, where properties are easy to find, and where PBT is relatively easy to apply. This suggests that an effective way to approach education and documentation around PBT would be to focus on "opportunistic" applications of PBT as an easy on-ramp.

Our interviews also identified several *Opportunities for Better Leverage*—situations where PBT is not easy *yet*, but could be with a little more research effort. For example, P7 spoke about trying to test a poorly abstracted program (which are notoriously resistant to PBT) by writing properties about output it prints to the terminal; we plan to operationalize this approach. Furthermore, more than three quarters of study participants had used a particular kind of PBT, commonly called *Model-Based Testing*. P3, an author of PBT tools at Jane Street, considered better automation and tooling around model-based testing to be one of the most significant ways improve PBT and make it easier to pick up and use.

We discuss plans for work that addresses *specification themes* in §2.

Another set of common themes in the interviews concerned the **generation** of random inputs for property-based testing. Many participants spoke highly of the *Derived Generators* that can be inferred from the OCaml type system (P5 called OCaml's tools for this "[expletive] amazing" and P30 called them a "game changer"). These generators are already quite good, but they could be better: participants identified deficiencies both small (e.g., API quirks) and large (e.g., derived generators cannot enforce semantic preconditions). When derived generators failed, participants fell back to *Bespoke Generators*, which are far more flexible but proportionally more time consuming to design and work with. For example, P20 found important bugs with a bespoke generator for XML documents, but reported spending "at least a day" writing it. Improving the abstractions available for writing bespoke generators would greatly improve both the experience of using PBT and its potential for bug-finding.

If a generated input is deemed a *counter-example* for a property, highlighting a bug in the code, it needs to be easy to use that input to determine the root cause of the bug. The interviews made clear that *Shrinking*, the process of finding the smallest possible input that provokes a given bug, is a critical part of the PBT process. P8 and P21, who each needed to implement their own PBT libraries, both made sure to include shrinkers in their implementations. Unfortunately, getting shrinking right, especially for values that must satisfy preconditions, is difficult.

We discuss plans for work that addresses *generation themes* in §3.

A final category of common themes concerned the **validation** of testing effectiveness. As with any sort of testing, when doing PBT it is difficult to know when "enough is enough." One standard approach is to simply test up to a time limit, but participants' approaches to *Budgeting Time for PBT* were anything but standardized. Most of our interviewees run PBT testing just for a few seconds per property, expecting that the bulk of the necessary testing is done in hundreds or up to around a thousand examples, but some run it for much longer. Interestingly, the opposite was true about a close cousin of PBT, fuzz testing. Participants P14 and P20, who used both PBT and fuzzing tools, ran the fuzzers for far longer (P14 ran the fuzzer for over a year!).

But even with arbitrary time budget, it is still difficult to know if PBT will find all of the bugs in a program. If the generator spends too much time producing uninteresting or invalid values, testing could still fail to find bugs after a *very* long time. For this reason, *Evaluation of Effectiveness* often featured in our conversations, and participants asked explicitly for tools to help them evaluate their generators and properties. There are many forms such feedback could take, from code coverage (requested by P9 and P25) to input space coverage measurements (requested by P10, P16, and P16) and even strategies like *mutation testing* [19].

We discuss plans for work that addresses *validation themes* in §4.

All of these themes will also inform our plans to improve education around PBT. Education projects are discussed in §5.

1 Foundation: Understanding Needs and Opportunities

The final results from the in-progress user study should paint a clear picture of the benefits and challenges of PBT in the specific context of Jane Street and other organizations with similar characteristics. But to fully understand the potential impact of PBT across the software industry—and the factors that may limit its adoption—we need to cast a wider net. In this section, we describe three planned studies, drawing on mixed methods in the service of producing a comprehensive and actionable agenda for future research, in this

research project and beyond—two written surveys, one to assess the generality of these needs and obstacles and one to identify potential for adoption of PBT tools (§1.1), and an observation study to understand particular tasks involved in PBT to guide the design of new algorithms and interactive tools (§1.2).

1.1 Generalizing the Jane Street Findings

Our ongoing Jane Street study has already revealed a number of opportunities to improve tools for property-based testing. To identify others and to better understand which of these opportunities are most important for the research community to explore, we will conduct two surveys with broader samples of developers. These surveys aim to (1) determine which obstacles observed in the interview study represent widely experienced pain points with PBT tools and (2) characterize the potential benefits of better tools to the software industry as a whole.

Validation survey. The main survey we plan to conduct aims to check that the things we learned from Jane Street generalize more broadly. Our interview study has revealed a number of issues with existing PBT tools, but it is not clear how broadly these issues are experienced by developers, or their relative severity. To find out, we will ask developers which of the issues we have identified are ones they have experienced, and the severity of those issues in their experience. Respondents will also be asked to report on other issues they encountered in their use of PBT tools. To provide clear usage scenarios to guide future work, respondents will be asked to write brief anecdotes elaborating on the most severe issues they had.

Respondents will be recruited broadly, from three sources. First, we will recruit users of Hypothesis, a widely used Python-based PBT framework (see the attached letter of support). Second, we will distribute the survey at Jane Street, hoping to reach a broader set of developers than we were able to interview. And third, we will recruit users from social media by posting survey announcements from the PIs' Twitter and Mastodon accounts, various mailing lists, and on discussion boards for conferences like "Yow!" where we have been invited to speak [20].

Impact survey. Another more speculative survey we hope to conduct asks how broadly PBT may eventually be able to reach. To get a sense of this, we survey "proximal" users of PBT—that is, developers who do not use PBT currently, but who experience use cases where PBT methods would be particularly useful. Particular attention will be given to those whose work requires the writing of validation code and public APIs with some definition of types. We will recruit a broad sample of participants across development contexts (professional, open source, educational) by working with our industry contacts and by recruiting over social media. Admittedly, this survey seems more difficult to "get right," and the design will evolve over time as we gain a better understanding of the situations where PBT is most effective (§5.1), but if done right this could provide invaluable feedback to PBT researchers about how far their work could reach if done well.

1.2 Observing PBT in Practice

Anecdotes related in interviews do not generally, in and of themselves, provide enough information to inform the design of effective tools—they beg questions like: (1) How much time are participants willing to devote to a task like creating a generator or debugging a counterexample when in the middle of a programming task? (2) How much space is available on a developer's screen (amidst other tools like code editors and terminals) for interacting with new tools? (3) What are developers' current strategies for solving the problems they describe (for example, what representations of generated data currently seem most helpful for programmers trying to under the distributions of data generated by a generator)?. As a basis for designing tools with a substantial novel interface component (see §4), we will observe developers undertaking the respective PBT tasks we aim to support, on the order of a small handful of developers (i.e., 2–10 observation sessions). These observation sessions will allow us to evolve singular anecdotes from the interviews into a more comprehensive picture of what kinds of designs will be viable for helping programmers.

Together, the studies described in this section provide a firm foundation many of the activities in this proposal, and for future research on PBT more generally. We will produce a validated and prioritized set of developer needs for PBT tools and a detailed set of requirements for how tools should be (re)designed to best satisfy the most important of those needs.

2 Specification: Widening the On-Ramp

The projects in this section address the *specification themes* discussed in the Motivation section. In §2.1 we describe a plan to significantly expand the scenarios where PBT has leverage, improving its applicability to poorly abstracted code. Then, in §2.2 we provide tools to apply PBT automatically in one specific scenario: model-based testing. Together, these projects will greatly improve the on-ramp to PBT and clarify its place in the software industry.

2.1 Properties Over Logs

One common problem raised by Jane Street developers (and even more loudly in our earlier pilot study [17]) was that code may sometimes not be abstracted in a way that is amenable to testing. Indeed, any kind of testing of software units—with PBT techniques or otherwise—requires "units" to test! But PBT is especially sensitive to issues like poorly encapsulated global state, which may impact the repeatability of testing, and leaky or confusing abstraction boundaries, which make it difficult to cleanly state properties or specifications.

One Jane Street developer (P7) described their experience testing a trading system (we'll call it System A) with messy abstraction boundaries. In this case, System A was hard to test because it was primarily used as a sub-component of a larger System B: System B would take in simple data, and then at some point pass a large, complex data structure to System A, which would do some work and hand results back to System B. PBT of System A was therefore hard, because generating valid examples of the large data structure that System A takes as input required deep knowledge of the internals and types of System B that the developers of System A did not have access to, and because the interface was too complex to write straightforward properties about.

One might imagine trying to test System A through System B by writing properties over intermediate values in System A. One might imagine trying to write and test properties like

```
sublist_of processed (next (processed))      (* 1. Processing is monotonic. *)
msg.length < 100 ==> never (overflow = true) (* 2. Capacity at least 100 bytes. *)
eventually (processed.length = msg.length)   (* 3. The whole message is processed. *)
```

which demonstrate that internal variables like `processed`, `msg`, and `overflow` evolve correctly over time. These properties could theoretically be tested without ever running System A on its own: we can generate inputs to System B, have System B drive System A, and simply monitor the variable values to check the desired properties.

The tricky part comes with how to actually implement this testing. One might hope to do it through debug assertions, but that quickly becomes untenable. Checking the property (1) with debug assertions would require noisy code to saves the previous value every time `processed` is updated, and the programmer would need to use macros or some other kind of meta-programming to remove that code when building in release mode. Even worse, there may be no way to check properties (2) or (3) at all! Doing so would require making an assertion at the "end" of the computation, which may not actually be possible if System B is driving System A from the outside.

Instead, we propose a framework for writing properties over *logged values* that can capture the interesting relationships between past and future variable values that the above properties demand. Concretely, we will insert lightweight logging annotations and provide a language for writing properties over those logs. In order to express concepts like `next`, `never`, and `eventually` the property language will need some logical connectives that are not standard in PBT. In particular, we need a temporal logic like *linear temporal logic* (LTL) so that properties can capture a notion of time.

Using LTL with PBT is not unheard of—Quickstrom testing framework [21] has already shown that LTL properties can be used for specifying and testing graphical user interfaces—but LTL properties at this level of generality have not yet been attempted. We plan to compile the user's LTL formulae to properties over traces that respect the structure of the log and ignore updates to irrelevant variables. Then, the developer can provide inputs to some system containing the sub-system under test, and the test harness will apply the trace properties in real time. In this way, the user can get feedback about the invariants that should hold of the intermediate values of their programs, making PBT available in a huge range of programs where it was previously too difficult to apply.

2.2 Models for Modules

One finding that has surprised us from the Jane Street study is that it is *very* common for developers to build (or already to have) a *model implementation* of the code they are testing and check that the two versions of the code agree on randomly generated sequences of API calls. This is a well-documented approach to PBT [22], but it is not supported as well as it could be by existing tooling. For example, interviewees described setting up *very similar* model-based PBT harnesses for at least three different projects. Each harness is specific to the problem at hand, so the hard work of setting up these tests has no hope of being reused from one project to another, but the logic behind the harnesses is ripe for automation.

The APIs that the Jane Street developers tested came in the form of ML-style *modules*. The module systems of OCaml and other ML-based languages are theoretically interesting and extremely expressive, and they are objects of study in their own right [23]. We propose new, type-based automation for model-based testing of these kinds of modules, building both theoretical and practical tools for understanding the ways that their APIs can be exercised.

We will work from first-principles, building a model of ML-module *traces*. This model will need to capture the various initial inputs that need to be generated, the output values that should be checked for agreement, and, most importantly, all of the ways that outputs from one call can flow to inputs of another. Since these values are all typed, this model will likely need to be dependently typed to maintain all necessary invariants.

Concretely, the tools we build will take a model signature, for example

```
module Map : sig
  type ('k, 'v) t
  val get : ('k, 'v) t -> 'k -> 'v option
  val put : ('k, 'v) t -> 'k -> 'v -> ('k, 'v) t
end
```

and generate a series of calls like

```
let m = put m 1 2; get m 1; let m = put m 1 3; m get 1
```

that manipulate the data structure and query its contents. When testing two different implementations of `Map`, the developer can verify that the two implementations produce the same results for each of the operations in the sequence. The automated tooling could also insert checks for internal invariants, ensure that one operation executes at least as quickly as another, and insert other checks that are simple to specify but tedious to implement manually.

With the prevalence of ML-derived languages used in practice today, and with ML modules' similarities to other tools for defining interfaces, this project has the potential to significantly increase the usability of model-based PBT.

Section §5.1 incorporates the ideas from this section along with many more into a comprehensive list of PBT patterns that developers can use to decide if PBT is an easy choice for testing their code.

3 Generation: Better Tools for Random Inputs

The Jane Street study also highlighted several *generation themes*. Generators were regularly cited as one of the most challenging parts of the PBT process, and many asked for better tools for constructing generators that reliably address their testing goals. In this section, we describe a series of projects to develop new and better ways to express random data generators that give developers more automatic ways to test their code more thoroughly.

Context: Why is Random Generation Hard? As discussed in the Orientation, PBT relies heavily on *random data generators*. The inputs produced by the generator exercise the developer's properties and give confidence that they hold—provided the inputs are interesting enough.

Preconditions. Many properties that developers want to test have *preconditions* (or *validity conditions* or *input constraints*) that restrict the set of inputs that should be used for testing. This comes up often when

testing data structures with invariants that must hold in order to apply the operations under test. Testing such properties can be problematic, because many preconditions are difficult to satisfy randomly; if the developer is not careful their testing may waste most of its time budget generating and discarding invalid inputs.

Coverage of Realistic Inputs. Worse, even among valid inputs, not all are equally realistic. Often generated inputs feel fabricated, since they are not constructed with prior knowledge of the kinds of inputs that the program is likely to see.

Distributional Concerns. Worse yet, even with validity and realism accounted for, there are other ways for generators to under-perform. In particular, the spaces of inputs that generators target is often very large, and the best generators explore as much of the space as possible, as quickly as possible.

The PBT literature partially addresses these concerns, with solutions solving one problem at the expense of another and trading off between programmer effort and generator power. The existing solutions fall on a spectrum from automatic to manual. The automatic approaches use proxies for validity and general "interestingness" of inputs: some, like *fuzzers* [24], optimize readily available metrics like code coverage, others ask users to provide metrics [25], and naturally some use machine learning to infer proxies for validity [26, 27]. These approaches are easy to apply and can obtain good distribution coverage, but they are rarely sufficient for testing properties with complex preconditions. Slightly more manual approaches are based on declarative representations of validity conditions: for preconditions that are primarily structural, *grammar-based fuzzing* provides a compelling solution [28, 29, 30, 31, 32], and for more complex, semantic preconditions, some have proposed using SMT-solvers [33, 34, 35] to automatically seek out valid inputs. These tools are much better at satisfying up to moderately-complex properties, but some are still out of reach. The semi-automatic category also includes tools for *example-based tuning*, a process that improves realism of inputs by mimicking user-provided examples [36]; the existing tools successfully generate more realistic inputs, but are again limited in the preconditions they can satisfy.

Context: Monadic Generators The most manual, but also the most flexible, solutions use hand-written generators, written in a convenient domain-specific language (DSL). In Haskell, where PBT was first popularized, many such DSLs are implemented using *monads* [37], an elegant design pattern for expressing effectful (in this case, random and stateful) computations in a pure, stateless underlying language. While monadic DSLs are not actually necessary to express generators in non-pure languages, some libraries (e.g., in OCaml) still use monadic abstractions to build their generator DSLs.

Monadic generators can implement random data producers of arbitrary complexity (e.g., it is possible to write a monadic generator for Haskell programs [38]), so they are often more expressive than other representations like grammar-based generators. Yet monadic generators are syntactically constrained in a way that isolates the probabilistic code and prevents usage errors (like passing the wrong random seed around). As we will see, the constrained nature of monadic generators also makes them the perfect candidates for sophisticated manipulations and interesting generalizations.

Context: Free Generators In order to improve monadic generation along the dimensions listed above, it helps to re-frame generators as *parsers of randomness*. A generator operates by making a series of random choices, but we can equivalently think of it as being provided some random sequence of choices and then simply following those choices to produce a value. This perspective has been used in a few implementations of PBT systems [39, 40]; we made it formal in our paper, *Parsing Randomness* [41].

The paper introduces *free generators*, which generalize the standard monadic generator abstraction, demonstrate a formal link between parsing and random generation, and enable new algorithms for generation that improve generation modulo validity constraints. Free generators are written the same way as standard monadic generators, but they are more like generator "plans" or syntax trees.[1] This means that a single free generator can be be *interpreted* in multiple different ways. In *Parsing Randomness* we prove that any free generator can be interpreted either as a standard monadic generator or as a source of random choice strings with a parser over those strings; this formalizes the relationship between generators and parsers.

[1] For experts: Free generators are implemented using *freer monads* [42], which have been used to great effect in recent years to capture the structure of effectful computations (cf. ITrees [43]). Freer monads represent monadic computations syntactically by reifying the monad operations (`return` and `>>=`) as data constructors. Critically, this is all implemented within the language (no macros or AST manipulation). See [41] for more details on the free generator representation.

Furthermore, since free generators are uninterpreted syntax trees, they can be manipulated programmatically. Free generators admit a version of a Brzozowski derivative [44] that can be used incrementalize generation. We showed that free generator derivatives enable an algorithm called *Choice Gradient Sampling*, which uses repeated derivatives to guide a generator to inputs that are more likely to be valid with respect to a given precondition. This work is an important step towards better automation for generators with complex preconditions.

3.1 Reflective Generators

Continuing on from the free generator work, we have begun to look at even more powerful generalizations of the standard monadic approach to random generation. One such project concerns monadic generators that can be run *backward*.

If a generator parses a sequence of choices into a value, then running the generator backward should take a value and produce a sequence of choices that would produce that value. With this in mind, we propose *reflective generators*, an extension of monadic generators that can be run backward to "reflect" on the choices that they made when producing an input. The machinery that makes reflective generators work is quite complex,[2] but, like free generators, their syntax is still quite close to that of normal monadic generators.

But reflective generators are useful for more than just generation! We plan to explore (at least) the following two example applications.

Validity-Preserving Mutation. Many automated testing algorithms (especially fuzzing algorithms [24]) *mutate* values to explore the behavior of the program in a space "around" those values. This can be difficult in PBT scenarios, where values are subject to complex validity constraints, since mutation often produces invalid values. Reflective generators can help: we can (1) reflect on the choices that lead to a particular value, (2) mutate those choices, and (3) re-run the generator with the new choices *while correcting any choices that would lead to an invalid value.* Figure 1 shows the way this algorithm is able to mutate a binary search tree, while maintaining validity, using no BST-specific code beyond the reflective generator itself.

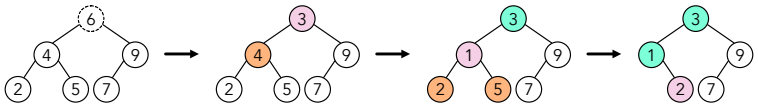

Figure 1: Validity-preserving mutation of a binary search tree, maintaining the BST invariant.

Example-Based Tuning. Earlier we pointed out that good generators produce "realistic" inputs; one way to ensure this is to tune the generator so it produces values that are similar to some user-supplied values deemed realistic. Existing tools make good use of this example-based approach to tuning [36], but they do not work with generators as powerful as monadic generators. We implement a similar algorithm using reflective generators: we can (1) again, reflect on the choices that lead to a set of realistic values, and (2) run the generator with *new choice weights* informed by the choices that we saw.

Both of these applications have the potential to significantly improve testing effectiveness—example-based tuning helps users generate more realistic inputs, while validity-preserving mutation enables more automated approaches to improving generator distributions—and we get *both* by upgrading our existing generators to reflective ones. We will see some additional use cases for reflective generators in the following sections.

3.2 Reflective Fuzzers

Fuzzers like AFL [24] use principles that are similar to the ones behind PBT: they leverage randomized testing to quickly exercise as many program behaviors as possible. One might then expect that the fuzzing and PBT share a significant amount of literature, but in reality they do not. Often the communities seem to "talk past" one another.

[2]Reflective generators are both monads and *partial profunctors*, implementing bidirectional programming in the style of Xia et al. [45]. This approach to bidirectional programming is related to lenses [46], but it hides much of the complexity of bidirectional program composition in the bind operation of the monad. The result is an elegant programming experience where both directions of the computation can be written at once, in a type-safe way.

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

There is already some work that attempts to bridge the gap between PBT and fuzzing. For example, the FuzzChick library in Coq [47] uses code coverage as guidance for PBT and the HypoFuzz library uses a similar approach in Python [48]. These projects are demonstrably powerful, but neither benefits from the years of expertise poured into industrial-strength fuzzers; Crowbar does [40]. Crowbar uses AFL [24], one of the best-established fuzzers, to generate random bit-strings that are later parsed into program inputs. (We plan to use AFL++ [49], which has supplanted the original AFL project.)

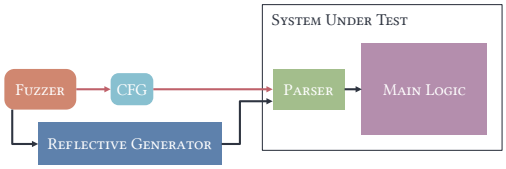

Figure 2: Overview of gradually constrained fuzzing.

Our system will start with a setup similar to Crowbar, but with a much greater focus on the generators; in particular, we will use reflective generators to gain significant testing power. The setup is shown in Figure §2. We start with a classic fuzzing setup, attempting to make the system under test crash by passing it a variety of semi-random inputs. Normally, the fuzzer is working against the parser, in the sense that the parser's job is to reject invalid inputs and the fuzzer's job is to "get past the parser." The CFGs used by grammar-based fuzzers help a bit, but they cannot generate inputs satisfying complex context-sensitive constraints. As in Crowbar, we avoid this adversarial relationship and subsume grammar-based generation with a powerful generator that is powerful enough to satisfy the parser by construction; in our case, that generator is reflective.

Why use a reflective generator? First, we should clarify why any kind of monadic generator is preferable to grammar-based options. Monadic generators can straightforwardly generate context-free structures, and they can often do so automatically with the help of type information [50]. If this is all that the precondition requires, then there is no harm in using a monadic generator, rather than a grammar-based one. But the beauty of a monadic generator is that it can be made far more powerful, incrementally, as the developer's testing needs change. The developer can start off thinking that they need only consider the structure of their inputs, but they can later add more semantic guarantees if they determine that their testing is ineffective.

Focusing on reflective generators specifically, one compelling benefit is that their backward interpretation can be used to help seed the fuzzer. Most fuzzers ask for a number of *seeds*, input examples that the fuzzer can start from, in order to ensure that the fuzzer does not spend ages exploring inputs that have no hope of working out. Normally these seeds are easy enough for the user to write down, since they are simply program inputs, but now that we are asking the fuzzer to generate sequences of choices it becomes much more error-prone (and tedious) to produce seeds by hand. This is one great use for a backward interpretation. The user can write down their seeds—either as values in the program, or as text that can be parsed by the program's parser—and then the reflective generator can reflect on the choices that produce those seeds.

The reflective generator also provides validity-preserving mutation. Guided by heuristics, the system can opt to supplement the fuzzer's mutation schedule with mutations that are obtained by the reflective generator's validity-preserving mutation (which is more targeted than the mutators provided by the fuzzer). We expect this to have a significant impact on performance, especially in contexts where preconditions are relatively sparse and therefore hard for AFL++ to mutate correctly.

Our ultimate goal is a grand unification of PBT and fuzzing generator tooling: the fuzzing literature provides battle-tested heuristics for coverage-guided generation, the PBT literature provides powerful tools like reflective generators for refining generator performance incrementally, and both communities benefit from generators with better distributions.

3.3 Reflective Shrinkers

A more modest application of reflective generators uses them to implement validity-preserving *shrinking* of values to find smaller counterexamples and speed up debugging. On its face, this feels similar to validity-preserving mutation: Can we reflect on choices, shrink the choices, and then re-run the generator with the smaller choices? Likely yes! But there are complications. When mutating, it is often fine if the mutated value is accidentally quite different from the original value, since the mutator is trying many values and any that catch a bug are equally good. But when shrinking, it is often very important to get another input that is both smaller and, ideally, provokes the *same bug*. These nice properties are likely within reach, but care will need to be taken to ensure that shrinkers behave as expected.

3.4 Benchmarking

The many papers in the PBT literature demonstrate effectiveness with case studies, showing that certain bugs in certain systems are caught more quickly with one too over another. For theoretical advances, this is often sufficient to demonstrate that the paper is worth publishing, but this kind of evaluation can be hard to interpret from the perspective of a would-be user. With all of the new approaches to generation that we are proposing in this document, and considering our goals around usability, we want to do better.

We will to develop and popularize a robust empirical evaluation framework for generators and other PBT techniques. Our first contribution will be an infrastructure for easily and extensibly running experiments. By "easily," we mean that we will take on the burden of collecting data and analyzing the results, exposing to the user library functions for their particular instantiations as needed. We will evaluate a given tool based on (1) the degree to which it is able to achieve high code coverage quickly, and (2) the speed with which it finds bugs that have been pre-seeded in example programs. By "extensibly," we mean that in addition to the two languages (Haskell and OCaml/Coq), multiple frameworks (QuickCheck, SmallCheck, QuickChick, etc.), and numerous workloads that we plan to support on release, we will design the infrastructure so that users can easily add new things along each dimension.

Our second contribution will codify a library of case-studies and examples as *benchmarks for PBT*. Similar suites of benchmarks already exist in the fuzzing literature [51], but those benchmarks are not organized around the particular challenges that PBT tools face. In particular, few of the benchmarks deal with the kinds of complex preconditions that PBT tools are built to handle. We want to establish a set of challenging tasks that can serve as a north star for future improvements to PBT generators and bug-finding strategies (including our own!).

Designs for this project are currently being discussed with Leonidas Lampropoulos and his group at the University of Maryland. PI Pierce has a long history of successful projects with Prof. Lampropoulos [34, 13, 52, 53, 47, etc.].

4 Validation: Understanding Testing Effectiveness

One of the unique challenges in creating usable property-based testing is providing adequate support for evaluating test results. Testing with properties is fundamentally different from conventional unit testing tools in many ways. First, it becomes a task in and of itself to understand individual inputs. This because the inputs are not written by the developer, but rather generated automatically, and furthermore, they can be of unbounded structural complexity. Second, a developer needs to assess the quality of their tests. The quality of a property-based test depends in part on the extent to which it covers execution pathways through the code, though it also depends on the quality of the inputs. Namely, are the inputs complex enough, and distributed in such a way that they are likely to trigger practically important bugs? Third, developers need to decide what to do with failed test cases, and may need to spend time migrating failed test cases into regression suites. Finally, developers need to make difficult decisions around how much computation time to budget for exercising the property-based tests.

Each of these points of departure from conventional testing methodology introduces new challenges into the testing process. And they require new approaches to tool design to help developers reason about complex distributions of complex inputs. In this section, we describe a sequence of research projects we will undertake to bring about usable developer tooling for property-based testing. These projects will contribute

new paradigms for tools that help developers understand program failures involving complex inputs (§4.1), assess whether their generators are generating sufficient, appropriate inputs (§4.2) and whether those inputs sufficiently exercise their code (§4.3), and migrate failed property-based tests into regression tests (§4.4). These projects will be pursued using human-computer interaction methodology, integrating these tools into contemporary development environments for functional programming. The result of this work will be a comprehensive, innovative set of design primitives for helping developers get work done in the challenge setting of reasoning about tests with an overabundance of input-output examples.

4.1 Understanding Failures

A developer can better understand a failure if the input that triggered it is simple. We propose rapid, incremental, interactive shrinking of complex inputs into simpler inputs. Shrinking is a well known technique in the PBT literature [54, 55], and §3.3 presents a novel approach that makes shrinking automatic when a reflective generator is available for a given type. But the shrinking process is often opaque and difficult to control. Our novel shrinking can be done *interactively*.

We will design interfaces that allow developers to shrink inputs by pruning the input's contents and structure in an interactive object viewer. The trick in interactive shrinking will be to point out to developers the *valid* ways in which an input can be pruned. These valid options for pruning can be collected as metadata for the input as they are generated by a reflective generator; in other words, many choices that the generator makes correspond to an aspect of the input that could be pruned. These pruning points will be made visible in the object viewer. As a developer prunes the input, they will receive continuous feedback as to whether it still causes a test to fail or not, to help the developer determine whether the simplified input in fact continues to trigger the same error. Developers will also be alerted if the execution path through the code has changed as a result of shrinking the input. In some circumstances, this will indicate an undesirable change in the shrunken input, and in other cases, such changes in execution paths may be permissible (e.g., if the change in the input has led to a reduction in the number of cycles through a loop).

With complex and simple inputs alike, we propose that tools should provide better support for helping developers identify areas of the code that are likely causes of undesired program behavior. We propose to make it clearer why and input fails as follows. First, we will generate additional inputs that are very close to the counterexample that are pass the test. Then, we will execute the program up to the point where the traces of the programs begin to diverge. Finally, we will drop the programmer into a debugging environment where they can query the state of the program and step through the remainder of the execution. PI Head has prior work designing debugging tools that help programmers understand trace divergences in an educational setting [56].

4.2 Evaluating Data Distributions

The work in §3 will significantly improve the tools that users have at their disposal when designing and using random generators, but reflective generators do not solve all of the problems developers may run into when writing generators. One common challenge is understanding when a random generator falls short of producing useful input data distributions. of values; many participants in our study complained that they did not really know what their generators were doing. In the worst case, the generators may generate totally unhelpful inputs (e.g., degenerate values like empty lists, or unrealistic values like strings of exclusively special characters), but even when some of the inputs are reasonable, the distribution may not hit interesting values frequently enough to make testing efficient. The participants asked for tools to help them catch mistakes that hamper their testing efforts and make it easier to quickly adapt and tune their generators to their specific needs.

Of course, understanding generator distributions is challenging. The values produced by generators are often highly structured and have complex shapes (e.g., lists, trees, and other algebraic data types). As a concrete example, consider a list of log events with type:

```
type log = {id : int; payload : string} list
```

Such values cannot simply be plotted on a chart, and even spot-checking them visually may be hard if the values are large. Ideally, a developer would have a way to answer questions like: Are the logs long enough? Do the messages have a reasonable distribution of payload lengths? Are the payloads realistic? etc.

To help developers answer this question, we propose to build a tool that gives developers comprehensive visualizations of their generators' distributions from directly within their code editor. The tool will address the challenges with visualizing generator distributions using a novel combination of tried-and-true features can support the task of understanding the quality of input data distributions. This includes (1) live, realtime displays of generated values; (2) the ability to drill-down from aggregate displays to individual values; (3) extensibility with lightweight hooks for customizing aggregate data displays and previews of individual points; (4) live feedback on what code is covered by the generated inputs.

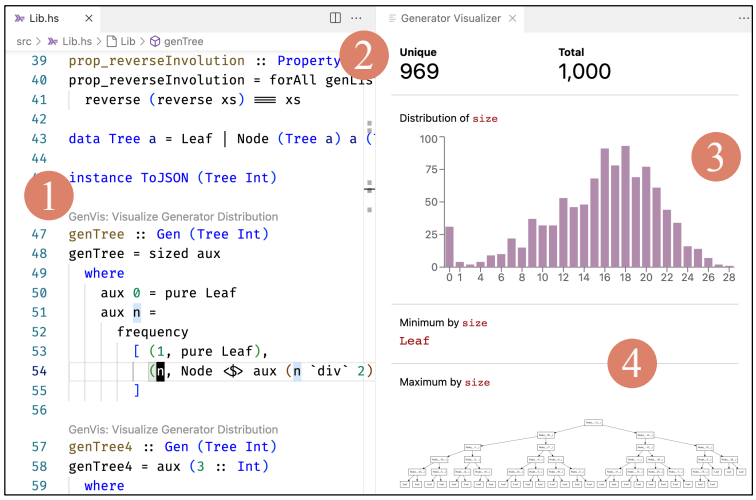

Figure 3: A mockup of our tool. (1) A *code lens* that the developer clicks to start the visualization. (2) High-level statistics about the generated values. (3) Charts displaying various data features. (4) Example values, rendered as graphs.

Our tool starts by sampling a developer's generator and obtaining a set of values to visualize. To get around the problem of arbitrarily structured data, the system extracts and plots relevant *features* of a given datatype, that will be extracted using functions in the surrounding programming environment and validated by the user. In the case of the `log` type above, we would consider simple functions like `length`, field accessors like `head`, `id`, and `payload`, filters like `is_empty`, and even aggregators like `maxBy` and `avgBy`. Then the tool will then use lightweight type-based program synthesis to compose and combine these functions to get features. It may choose to show the `length` of a log, but also the `maxBy (fun l -> length l.payload)` (the maximum payload length), and even pairs of features like these (which could be viewed as a two dimensional feature). If there are features that the user notices should be extracted, but that the system cannot come up with itself (e.g., `ids_unique`) the user can write it themselves and add it to the interface.

With the features extracted, the tool will then select various charts and graphs to visualize the data, depending on its type. We are not yet certain exactly which charts will be most helpful, but we will start with things like histograms for discrete numerical data, pie charts for categorical data, scatter plots for two dimensional numerical data, etc. Of course, developers would struggle to grok dozens of charts representing their data's features, so we will also have a way for users to specify which visualizations are interesting (and which to hide). We will base these interactions based on the literature around visualization recommendation, specifically Voyager [57, 58]. This interaction might be very quick (maybe a few seconds to find a single representative chart) or slow and careful (many minutes spent curating a dashboard of results to help with tuning), depending on how much time and energy the developer has to put into their generator.

Another important aspect of understanding a generator's distribution is looking at example values. We will therefore also present the user with views into some of the generated values, which they can use to spot-check the generated data. As we have mentioned, the shape and scale of these inputs may be problematic—showing the user simple print-outs of the examples will not be good enough. Thus, we propose potential ways of visualizing examples that may become part of our tool. When the input *size* is the main barrier to understandability, we can provide the user with the values represented as JSON objects, and give them ways to manipulate those objects *in situ*. This may involve exploring the data via mouse interactions (e.g., clicking to expand or collapse sections) or even via a REPL with commands for filtering and drilling into the structure. This will be a huge help when scale is a bottleneck, but what if the user really needs to know about the data's *shape*? In this case, we can give developers views of their data structures as DOT graphs [59], which we can generate via meta-programming and render for the user. The graph visualizations will only work for smaller inputs (e.g., less than 100 nodes), but for certain data structures (e.g., trees) they may prove invaluable for understanding biases in the data.

When complete, this project will be the first live tool for generator feedback and analysis; in the next

section, we take it one step farther.

4.3 Tuning Data Distributions

The tool proposed in the previous section will give developers unprecedented visibility into their generator distributions, but what should they do if the distribution is not the right one? Tuning generators is a challenging task that would be much easier with the support of tools.

We propose extending the above visualization tool to include *bidirectional interactions*: the developer will be able to select parts of the visualization that seem wrong and edit them, causing changes to flow back into their editor as modifications to the generator. These interactions include things like (1) selecting points in a scatter plot to filter out; (2) dragging a bar on a bar chart up or down, to indicate that a certain feature value should be more or less represented; or (3) manipulating the sections of a pie chart to indicate a better categorical distribution.

Some of these interactions can be implemented via *filtering*. Generators already support operations that filter their outputs via predicates, so this is a natural choice. If the user selects a range of values, as in interaction (1), we can simply ask the generator to discard any values that do not fall in that range. However, filtering generators this way can be problematic: if the filter is too broad, the generator will slow down significantly as it wastes time generating inputs that it will have to discard. We can protect users by reporting a *discard rate* and warning them when that rate reaches a certain threshold, but a more robust to altering the generator would be better.

Luckily, we may be able to both alleviate concerns around discard rates and implement more interactions with the help of reflective generators (introduced in §3). Recall that reflective generators can operate in reverse, reflecting on the generator choices that produce a particular value. This means that if the user expresses a desire like "category X should be better represented" (maybe by dragging a bar or a pie slice as in interactions (2) or (3)), reflective generators can determine which choices often result in values of that category and raise the weights on those choices. And this mode of interaction can just as easily be applied to de-prioritize certain choices, meaning that in some situations re-weighting with reflective generators can obviate the need for expensive filters.

Ultimately we see this project and the previous one as a comprehensive toolkit for generator customization. The understanding and tuning generators is too often a cryptic process that users are inclined to avoid, to the detriment of their testing success. With interactive tools, developers will be far more likely to build useful generators that they can understand, and that they can customize as they see fit.

4.4 Counterexamples as Regression Tests

After a developer has identified a failure in their property-based tests, they often wish to turn that failure into a regression test, to ensure that later changes to the code will not reintroduce the failure. One pain point experienced by informants in our interview study was that it required considerable work to transform a failure that was already detected by their PBT tools into a regression test, despite the fact that much of the work involved in doing so felt mechanical.

We plan to develop usable tooling for transforming failed PBT tests into single regression tests. What is important to note about this project is that creation of regression tests is *mostly*, but not entirely mechanical. In reality, the creation of regression tests will likely require judicious incorporation of the developer's input at key decision points. This is particularly the case for specifying acceptance criteria. For instance, consider a property that checks that a list insertion function never produces an empty list. In the event of a failure, a developer may want to produce a regression test checking the exactness of the result for the failed input (i.e., checking on a particular concrete list that should have been produced by the insertion) rather than simply checking that the output list is non-empty. The act of writing regression tests involve several such choices, including… Finish me.

The projects in this section must all be designed in concert with real users evaluated in terms of their impact and usability. We will design these tools using *human-centered design*: we will show prototypes to potential users, get feedback, redesign, and repeat, until we are satisfied that our design will solve users' problems effectively. When evaluating the designs, we will conduct short user studies, comparing a users'

experiences with our tools to their performance on the same tasks without those tools. Designing and evaluating in this way will increase our potential for impact and keep our work grounded in the needs of real users.

5 Education: Advancing PBT in the Broader Culture

Needs introduction.

5.1 When to Specify It!

PBT is often described as a kind of lightweight formal method, and one might therefore imagine that a central challenge of using PBT would be coming up with the right specifications. Indeed, our pilot study concluded just that. But at Jane Street we actually heard very little about the challenge of coming up with properties; instead, most participants described applying PBT in scenarios when properties were already available or quite easy to imagine. This suggests that, while PBT educators should certainly spend some time teaching developers how to write specifications for arbitrary programs, they should spend even more energy helping developers quickly recognize situations where PBT is a particularly natural fit because properties are obvious!

The study with Jane Street provides a solid start to a comprehensive list of high-leverage use-cases for PBT. We have already identified seven software patterns where properties are relatively straightforward to find and PBT is an obvious choice for testing: 1. Code that is already (semi-)formally specified. 2. Functions that round trip. 3. Pure data structures. 4. Modules with invariants. 5. Related versions of the same code. 6. Programs that may fail catastrophically. 7. Stateful APIs with well-understood contracts. These patterns are already quite broad in their scope, and include many scenarios that real software developers regularly find themselves in, but they are likely not exhaustive. Thus, we will continue to gather examples of high-leverage use-cases: we will survey research papers and experience reports, examine open-source software projects, and leverage our connections with various PBT communities (QuickCheck/Haskell, Hypothesis/Python, Quickcheck/OCaml, etc.) to ensure that we have a comprehensive understanding of the best uses for PBT.

Then, we will compile a survey paper (with an accompanying talk) entitled *When to Specify It!* in homage to John Hughes's *How to Specify It!* [60], a lovely tutorial on the many different kinds of properties that one can write for a given piece of code. Clearly it is important to ask *How*, but our developer interviews suggest that it may be even more important to have a clear sense of *When* to use PBT. Our paper on *When to Specify It!* will combine examples from developer interviews with ones from our our years of experience studying and applying PBT into the definitive guide for the most impactful opportunities to apply PBT.

5.2 Interactive Property Specification

Preliminary findings from the Jane Street study suggest that developers sometimes have difficulty imagining which properties to test, even when they believe their software would benefit from property-based testing. One area we are interested in exploring is how programmers can work with their PBT tools to decide on a set of significant properties to test. Prior research demonstrates that automated tools can extract specifications of a program's behavior [61, 62, 63]. We are interested in the non-trivial research problem of integrating such techniques into usable developer tools. We see the research as addressing several challenges:

Generated properties should be important. Any non-trivial program can be characterized by an overwhelmingly large number of properties, many of which describe only incidental aspects of the program's behavior that do not need to be tested. How can tools produce those properties that developers would want to have tested? We believe that this is a problem that can be best solved with a mixed-initiative approach [64], where properties are determined by judiciously incorporating both developer input and automated techniques.

Developer could guide a specification mining tool to extracting relevant properties through input mechanisms such as (1) identifying regions of code that are likely to lead to an adverse behavior such as an exception or a logical error; (2) providing unit test cases that test a special case of a generalized property; and (3) indicating aspects of interest on input and output data during exploration in a debugging REPL.

Generated properties should be readable. While prior research [63] has shown promise for automatically generating specifications of program behavior, we expect that users of PBT systems would benefit from

having properties generated in the language of their property-based testing tools. Furthermore, there are some variants about systems that may be so complex that they require significant comprehension time for users (i.e., those involving a large number of clauses). In such cases, a tool may way to generate simpler variants of properties first, and allow developers to refine them on their own.

Tools should help developers revise properties. If a generated property is too relaxed, the tool should request that developer provides a counterexample that should trigger a failure, and then regenerate the property. If a generated property is too strict, a tool should allow a programmer to mark a counterexample that was generated by the PBT tool as spurious, i.e., not indicating an actual failure of the program. In each of these cases, the property generator may have generate multiple properties for a developer to review, each of which may satisfy the refinements that a developer has provided.

Plan of work: We will iteratively design and develop developer tools as an extension to the VSCode editor. In our first iterations, we will generate candidate properties using QuickSpec [63], a tool that builds on Haskell's QuickCheck library to synthesize a series of plausible properties about a given function. Future iterations will include tools like [65]. The interactions described above will be designed and tested first for simple programs, and then on successively more complex programs from our benchmark set. PI Head has extensive experience designing and developing developer tools involving program analysis [66, 67] and program synthesis [68] components, and extending the VSCode environment [69].

5.3 PBT in the Undergrad Curriculum

Write me.

6 Plan of Work

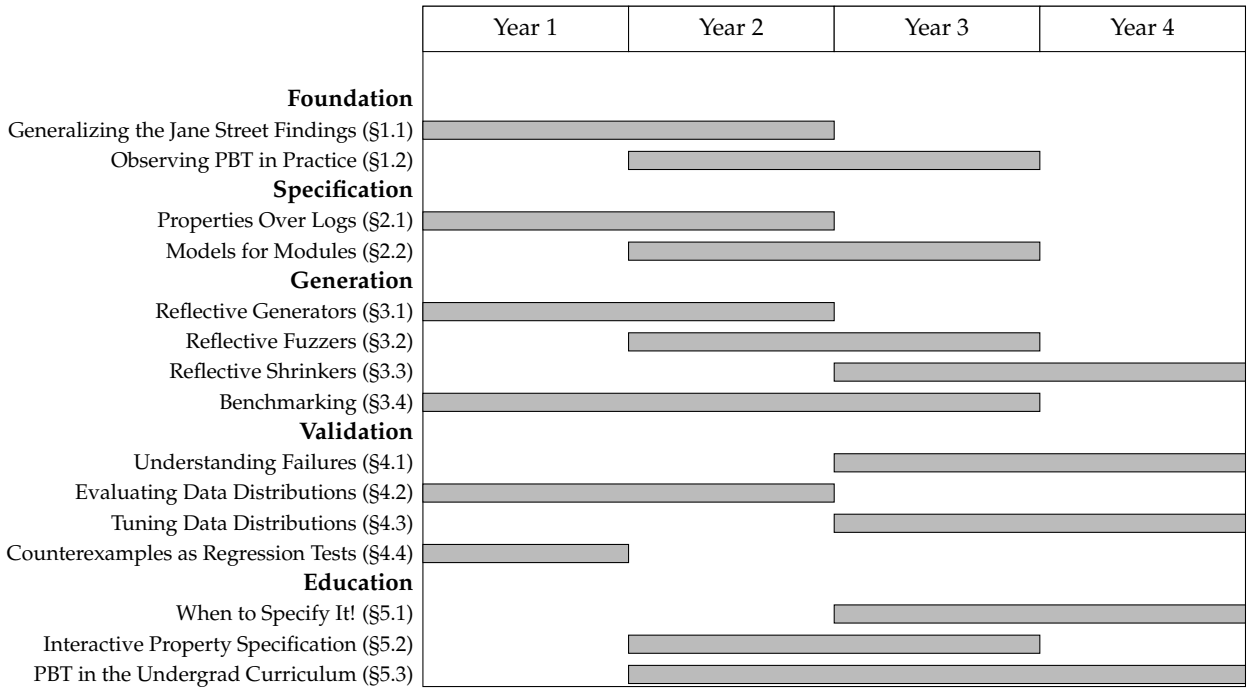

Figure 4: Plan of work.

Write me.

7 Broader Impacts

Write me.

8 Results from Prior NSF Support

Write me.

References Cited

[1] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," in *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, ser. ICFP. ACM, 2000, pp. 268–279. [Online]. Available: http://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/quick.pdf

[2] T. Arts, J. Hughes, J. Johansson, and U. Wiger, "Testing telecoms software with quviq quickcheck," in *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, 2006, pp. 2–10.

[3] J. Hughes, B. Pierce, T. Arts, and U. Norell, "Mysteries of dropbox," 2014.

[4] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, J. V. Geffen, and A. Warfield, "Using lightweight formal methods to validate a key-value storage node in amazon s3," in *SOSP 2021*, 2021. [Online]. Available: https://www.amazon.science/publications/using-lightweight-formal-methods-to-validate-a-key-value-storage-node-in-amazon-s3

[5] T. Arts, J. Hughes, U. Norell, and H. Svensson, "Testing autosar software with quickcheck," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2015, pp. 1–4.

[6] J. Hughes, "Experiences with quickcheck: testing the hard stuff and staying sane," in *A List of Successes That Can Change the World*. Springer, 2016, pp. 169–186.

[7] Z. H. D. (current Hypothesis maintainer), Personal communication, 2022.

[8] S. E. Chasins, E. L. Glassman, and J. Sunshine, "PL and HCI: better together," *Communications of the ACM*, vol. 64, no. 8, pp. 98–106, Aug. 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3469279

[9] J. Hughes, "Quickcheck testing for fun and profit," in *International Symposium on Practical Aspects of Declarative Languages*. Springer, 2007, pp. 1–32.

[10] R. Hamlet, "Random testing," *Encyclopedia of software Engineering*, vol. 2, pp. 971–978, 1994.

[11] C. Runciman, M. Naylor, and F. Lindblad, "Smallcheck and lazy smallcheck: automatic exhaustive testing for small values," in *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, A. Gill, Ed. ACM, 2008, pp. 37–48. [Online]. Available: https://doi.org/10.1145/1411286.1411292

[12] R. M. Braquehais, "Tools for discovery, refinement and generalization of functional properties by enumerative testing," October 2017. [Online]. Available: http://etheses.whiterose.ac.uk/19178/

[13] H. Goldstein, J. Hughes, L. Lampropoulos, and B. C. Pierce, "Do judge a test by its cover," in *Programming Languages and Systems: 30th European Symposium on Programming, ESOP 2021, Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021*, ser. Lecture Notes in Computer Science, vol. 12648, 2021, pp. 264–291. [Online]. Available: https://link.springer.com/chapter/10.1007%2F978-3-030-72019-3_10

[14] J. Wrenn, T. Nelson, and S. Krishnamurthi, "Using relational problems to teach property-based testing," *The art science and engineering of programming*, vol. 5, no. 2, 2021.

[15] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce, "Foundational Property-Based Testing," in *Interactive Theorem Proving*, ser. Lecture Notes in Computer Science, C. Urban and X. Zhang, Eds. Cham: Springer International Publishing, 2015, pp. 325–343.

[16] Y. Minsky, "Signals and threads," Web, Jane Street, LLC, 2022. [Online]. Available: https://signalsandthreads.com/

[17] H. Goldstein, J. W. Cutler, A. Stein, B. C. Pierce, and A. Head, "Some problems with properties," 2022.

[18] ——, "Some Problems with Properties," vol. 1, Dec. 2022. [Online]. Available: https://harrisongoldste.in/papers/hatra2022.pdf

[19] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Mutation Testing Advances: An Analysis and Survey," *Advances in Computers*, Jan. 2018. [Online]. Available: http://dx.doi.org/10.1016/bs.adcom.2018.03.015

[20] "(When) Will Property-Based Testing Rule the World? | SkillsCast." [Online]. Available: https://skillsmatter.com/skillscasts/17525-when-will-property-based-testing-rule-the-world

[21] L. O'Connor and O. Wickström, "Quickstrom: property-based acceptance testing with LTL specifications," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, Jun. 2022, pp. 1025–1038. [Online]. Available: https://doi.org/10.1145/3519939.3523728

[22] J. Hughes, "Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane," in *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, ser. Lecture Notes in Computer Science, S. Lindley, C. McBride, P. Trinder, and D. Sannella, Eds. Cham: Springer International Publishing, 2016, pp. 169–186. [Online]. Available: https://doi.org/10.1007/978-3-319-30936-1_9

[23] D. MacQueen, "Modules for standard ML," in *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, ser. LFP '84. New York, NY, USA: Association for Computing Machinery, Aug. 1984, pp. 198–207. [Online]. Available: http://doi.org/10.1145/800055.802036

[24] M. Zalewski, "AFL quick start guide," http://lcamtuf.coredump.cx/afl/QuickStartGuide.txt, Apr. 2018.

[25] A. Löscher and K. Sagonas, "Targeted property-based testing," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 46–56. [Online]. Available: https://doi.org/10.1145/3092703.3092711

[26] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 50–59.

[27] S. Reddy, C. Lemieux, R. Padhye, and K. Sen, "Quickly generating diverse valid test inputs with reinforcement learning," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 1410–1421. [Online]. Available: https://doi.org/10.1145/3377811.3380399

[28] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, 2008, pp. 206–215.

[29] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 445–458.

[30] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming," in *European Symposium on Research in Computer Security*. Springer, 2016, pp. 581–601.

[31] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.

[32] P. Srivastava and M. Payer, "Gramatron: Effective grammar-aware fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 244–256.

[33] K. T. Dewey, *Automated Black Box Generation of Structured Inputs for Use in Software Testing*. University of California, Santa Barbara, 2017.

[34] L. Lampropoulos, D. Gallois-Wong, C. Hritcu, J. Hughes, B. C. Pierce, and L. Xia, "Beginner's Luck: a language for property-based generators," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, 2017, pp. 114–129. [Online]. Available: http://dl.acm.org/citation.cfm?id=3009868

[35] D. Steinhöfel and A. Zeller, "Input invariants," *ESEC/FSE 2022*, 2022.

[36] E. Soremekun, E. Pavese, N. Havrikov, L. Grunske, and A. Zeller, "Inputs from hell learning input distributions for grammar-based test generation," *IEEE Transactions on Software Engineering*, 2020.

[37] E. Moggi, "Notions of computation and monads," *Information and computation*, vol. 93, no. 1, pp. 55–92, 1991.

[38] M. H. Pałka, K. Claessen, A. Russo, and J. Hughes, "Testing an optimising compiler by generating random lambda terms," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. New York, NY, USA: Association for Computing Machinery, May 2011, pp. 91–97. [Online]. Available: http://doi.org/10.1145/1982595.1982615

[39] D. R. MacIver, Z. Hatfield-Dodds *et al.*, "Hypothesis: A new approach to property-based testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, 2019.

[40] S. Dolan and M. Preston, "Testing with crowbar," in *OCaml Workshop*, 2017.

[41] H. Goldstein and B. C. Pierce, "Parsing Randomness," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA2, pp. 128:89–128:113, Oct. 2022. [Online]. Available: https://doi.org/10.1145/3563291

[42] O. Kiselyov and H. Ishii, "Freer monads, more extensible effects," *ACM SIGPLAN Notices*, vol. 50, no. 12, pp. 94–105, 2015.

[43] L.-y. Xia, Y. Zakowski, P. He, C.-K. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic, "Interaction trees: representing recursive and impure programs in coq," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, 2019.

[44] J. A. Brzozowski, "Derivatives of regular expressions," *Journal of the ACM (JACM)*, vol. 11, no. 4, pp. 481–494, 1964.

[45] L.-y. Xia, D. Orchard, and M. Wang, "Composing bidirectional programs monadically," in *European Symposium on Programming*. Springer, 2019, pp. 147–175.

[46] J. N. Foster, "Bidirectional programming languages," Ph.D. dissertation, University of Pennsylvania, 2009.

[47] L. Lampropoulos, M. Hicks, and B. C. Pierce, "Coverage guided, property based testing," Apr. 2019, to appear in PACMPL / OOPSLA 2019.

[48] Z. Hatfield-Dodds, "HypoFuzz." [Online]. Available: https://hypofuzz.com/

[49] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++} : Combining Incremental Steps of Fuzzing Research," 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[50] A. Mista and A. Russo, "Deriving compositional random generators," in *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*, 2019, pp. 1–12.

[51] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A Ground-Truth Fuzzing Benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 49:1–49:29, Jun. 2021. [Online]. Available: http://doi.org/10.1145/3428334

[52] L. Lampropoulos, M. Hicks, and B. C. Pierce, "Coverage guided, property based testing," *PACMPL*, vol. 3, no. OOPSLA, pp. 181:1–181:29, 2019. [Online]. Available: https://doi.org/10.1145/3360607

[53] L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce, "Generating good generators for inductive relations," *PACMPL*, vol. 2, no. POPL, pp. 45:1–45:30, 2018. [Online]. Available: http://doi.acm.org/10.1145/3158133

[54] J. Hughes, "QuickCheck Testing for Fun and Profit," in *Practical Aspects of Declarative Languages*, ser. Lecture Notes in Computer Science, M. Hanus, Ed. Berlin, Heidelberg: Springer, 2007, pp. 1–32.

[55] T. Arts, "On shrinking randomly generated load tests," in *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, ser. Erlang '14. New York, NY, USA: Association for Computing Machinery, Sep. 2014, pp. 25–31. [Online]. Available: http://doi.org/10.1145/2633448.2633452

[56] R. Suzuki, G. Soares, A. Head, E. Glassman, R. Reis, M. Mongiovi, L. D'Antoni, and B. Hartmann, "Tracediff: Debugging unexpected code behavior using trace divergences," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2017, pp. 107–115.

[57] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer, "Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 1, pp. 649–658, Jan. 2016, conference Name: IEEE Transactions on Visualization and Computer Graphics.

[58] K. Wongsuphasawat, Z. Qu, D. Moritz, R. Chang, F. Ouk, A. Anand, J. Mackinlay, B. Howe, and J. Heer, "Voyager 2: Augmenting Visual Analysis with Partial View Specifications," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. Denver Colorado USA: ACM, May 2017, pp. 2648–2659. [Online]. Available: https://dl.acm.org/doi/10.1145/3025453.3025768

[59] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphviz— Open Source Graph Drawing Tools," in *Graph Drawing*, ser. Lecture Notes in Computer Science, P. Mutzel, M. Jünger, and S. Leipert, Eds. Berlin, Heidelberg: Springer, 2002, pp. 483–484.

[60] J. Hughes, "How to specify it!" *20th International Symposium on Trends in Functional Programming*, 2019.

[61] G. Ammons, R. Bodik, and J. R. Larus, "Mining specifications," *ACM Sigplan Notices*, vol. 37, no. 1, pp. 4–16, 2002.

[62] T.-D. B. Le and D. Lo, "Deep specification mining," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 106–117.

[63] K. Claessen, N. Smallbone, and J. Hughes, "Quickspec: Guessing formal specifications using testing," in *International Conference on Tests and Proofs*. Springer, 2010, pp. 6–21.

[64] J. E. Allen, C. I. Guinn, and E. Horvtz, "Mixed-initiative interaction," *IEEE Intelligent Systems and their Applications*, vol. 14, no. 5, pp. 14–23, 1999.

[65] C. Smith, G. Ferns, and A. Albarghouthi, "Discovering relational specifications," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Paderborn Germany: ACM, Aug. 2017, pp. 616–626. [Online]. Available: https://dl.acm.org/doi/10.1145/3106237.3106279

[66] A. Head, E. L. Glassman, B. Hartmann, and M. A. Hearst, "Interactive extraction of examples from existing code," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–12.

[67] A. Head, F. Hohman, T. Barik, S. M. Drucker, and R. DeLine, "Managing messes in computational notebooks," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019, pp. 1–12.

[68] A. Head, E. Glassman, G. Soares, R. Suzuki, L. Figueredo, L. D'Antoni, and B. Hartmann, "Writing reusable code feedback at scale with mixed-initiative program synthesis," in *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*, 2017, pp. 89–98.

[69] A. Head, J. Jiang, J. Smith, M. A. Hearst, and B. Hartmann, "Composing flexibly-organized step-by-step tutorials from linked source code, snippets, and outputs," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–12.