JFP, **??** pages, 2022. © Cambridge University Press 2022
doi:10.1017/xxxxx

Etna: An Evaluation Platform for Property-Based Testing

ALPEREN KELES
University of Maryland, College Park, MD, USA

JESSICA SHI
University of Pennsylvania, Philadelphia, PA, USA

NIKHIL KAMATH
University of Maryland, College Park, MD, USA

TIN NAM LIU
University of Pennsylvania, Philadelphia, PA, USA

CEREN MERT
University of Maryland, College Park, MD, USA

HARRISON GOLDSTEIN
University of Pennsylvania, Philadelphia, PA, USA

BENJAMIN C. PIERCE
University of Pennsylvania, Philadelphia, PA, USA

LEONIDAS LAMPROPOULOS
University of Maryland, College Park, MD, USA

Abstract

Property-based testing is a mainstay of functional programming, boasting a rich literature, an enthusiastic user community, and an abundance of tools — so many, indeed, that new users may have difficulty choosing. Moreover, any given framework may support a variety of strategies for generating test inputs; even experienced users may wonder which are better in any given situation. Sadly, the PBT literature, though long on creativity, is short on rigorous comparisons to help answer such questions.

We present Etna, a platform for empirical evaluation and comparison of PBT techniques. Etna incorporates a number of popular PBT frameworks and testing workloads from the literature, and its extensible architecture makes adding new ones easy, while handling the technical drudgery of performance measurement.

To illustrate its benefits, we use Etna to carry out several experiments with popular PBT approaches in Rocq, Haskell, OCaml, Racket, and Rust, allowing users to more clearly understand best practices and tradeoffs.

1 Introduction

Haskell's QuickCheck library popularized *property-based testing* (PBT), which lets users test executable specifications of their programs by checking them on a large number of inputs. In fact, QuickCheck made PBT so popular that Claessen and Hughes's seminal paper [2000] is the most cited ICFP paper of all time by a factor of two, according to the ACM Digital Library. PBT tools can now be found in languages from OCaml (Cruanes, 2017; Dolan, 2017) and Scala (Nilsson, 2019) to Erlang (Arts et al., 2008; Papadakis and Sagonas, 2011) and Python (MacIver, 2016), not to mention proof assistants like Rocq (Lampropoulos and Pierce, 2018), Agda (Lindblad, 2007), and Isabelle (Bulwahn, 2012*a*).

Many aspects of PBT impact its effectiveness, from the properties themselves (Hughes, 2019) to counterexample minimization (Maciver and Donaldson, 2020), but arguably the most crucial one is the algorithm for generating test inputs. Papers citing QuickCheck often retain its distinctive style of *random* test-case generation, but many other options have been explored. In particular, *enumerative* PBT has also become a staple in the functional programming community (Runciman et al., 2008; Braquehais, 2017), and tools for *feedback-based* PBT are gaining ground (Lampropoulos et al., 2019; Dolan, 2017; Löscher and Sagonas, 2017). Each of these approaches comes with benefits and tradeoffs, and choosing one over another can make a big difference on testing effectiveness.

Even after selecting a generation style — say, random PBT — one may be left with quite a few options of *framework*, each with its own unique style. In Haskell, for example, both QuickCheck and Hedgehog (Stanley, 2019) are quite popular. And even after selecting a framework — say, QuickCheck — there are yet more options for choosing a specific generation strategy. Tools like generic-random (Xia, 2018) and DraGEN (Mista and Russo, 2021) can derive QuickCheck generators from type information, offering a quick and accessible entrypoint to PBT, but their effectiveness suffers when inputs need to satisfy more complex semantic constraints. Alternatively, one can write a *bespoke* generator that is "correct by construction," producing only *valid* test inputs. Such bespoke generators can sometimes become quite sophisticated (Palka et al., 2011; Midtgaard et al., 2017; Hritcu et al., 2016). And there are other options: for example, QuickChick, Rocq's variant of QuickCheck, can derive specialized generators for free from specifications expressed as inductive relations (Paraskevopoulou et al., 2022). Nuances of the properties under test may make strategies more or less preferable, and considerable experience may be required to make a good choice.

Moreover, even after selecting a particular way of using the tool — say, writing a bespoke generator — there are *yet more* options: a given generator can typically be tuned to produce different sizes and shapes of data. For example, QuickCheck generators can be parameterized both globally by a size parameter and locally by choices like numeric weights on the arguments to various combinators.

In the existing literature, there are plenty of performance evaluations for individual PBT tools, but a dearth of *comparisons* across the various available design dimensions. New tools are typically evaluated on just one or two case studies, often showcasing incomparable measures of effectiveness. So how is a PBT user supposed to make sense of all these options?

How is a tool designer supposed to measure success? How can we turn PBT from an art to a science?

Answering these questions is the goal of this paper. Our contributions are:

- We present Etna, an extensible platform for evaluating and comparing generation techniques for PBT, with generic support for measuring performance and presenting results (§2).[1]
- We populate Etna with six testing *workloads* from the literature presenting a range of bug-finding challenges, with PBT *frameworks* in Haskell, Rocq, OCaml, Racket, and Rust, and with various *strategies* for using each framework (§3).
- We report on our experiences using Etna to make observations about PBT performance. Some of these observations lend empirical weight to commonly held beliefs, while others suggest improvements to existing processes and tools (§4, §5, and §6).
- We extend Etna with support for cross-language experimentation with popular PBT frameworks in Haskell, Rocq, OCaml, Racket, and Rust (§7), enabling, for the first time, precise comparisons of generator efficiency and effectiveness across languages.

We discuss related and future work in §8.

2 Platform Design

The central purpose of Etna is to give researchers, library authors, and expert PBT users an extensible platform for experimenting with their testing strategies. In this section, we outline our design principles and the rationale behind them. Then we describe the Etna architecture and finish with a discussion and depiction of communicating with Etna as a user. Before all that, however, we'll begin by providing background on the generation strategies themselves.

2.1 Background: Property-Based Testing and Generation Strategies

A key difference between approaches to PBT is how each deals with *preconditions*. Consider *binary search trees*, where each node value is greater than everything to its left and less than everything to its right. In Haskell syntax:

```
data Tree k v = Leaf | Node (Tree k v) k v (Tree k v)
isBST  :: Tree k v -> Bool
insert :: k -> v -> Tree k v -> Tree k v
```

What properties should we expect to hold for operations on BSTs such as `isBST` and `insert`? Hughes thoroughly answers this question in his guide to writing properties of pure functions [2019]. For instance, one desirable property is that if we insert a key into a valid BST, then it should remain a valid BST:

[1] Etna is publicly available at https://github.com/alpaylan/etna-cli.

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

Here ==> encodes a *precondition*. That is, the `insert` function is only exercised when the binary tree `t` satisfies the `isBST` predicate; otherwise, the property is vacuously true.

There are many ways to generate data for properties like this. A simple approach is to straightforwardly follow the structure of the types to generate arbitrary trees and filter out the ones that are not BSTs. While simplistic, this approach works well in some circumstances. In fact, for the BST example, such *type-driven* approaches can find all bugs introduced in Hughes's guide to writing properties of pure functions [2019]. But this generate-and-filter approach breaks down with "sparse" preconditions that are harder to satisfy randomly; for instance, valid red-black trees are harder to generate at random than valid BSTs, so type-driven strategies work less well (see §4 and §5). For yet sparser preconditions, such as C programs with no undefined behaviors (Yang et al., 2011), such an approach is hopeless. On the other end of the spectrum, users can write *bespoke generators*: programs that are manually tailored to produce the desired distribution. Such programs can be extremely effective in finding bugs when the inputs satisfy the precondition by construction, but they can also be extremely difficult to write. A well-crafted such generator can in fact be a significant research result: such is the case for many well-typed term generators in the last decade (Palka et al., 2011; Midtgaard et al., 2017; Hoang et al., 2022; Frank et al., 2024). Naturally, there are also approaches in the middle. For instance, some use the structure of the precondition to produce valid data directly (Bulwahn, 2012*b*; Claessen et al., 2014; Fetscher et al., 2015; Lampropoulos et al., 2017, 2018), while others leverage feedback to guide generation towards valid or otherwise interesting inputs (Löscher and Sagonas, 2017; Löcher and Sagonas, 2018; Lampropoulos et al., 2019).

2.2 Design Principles

Etna is desinged to help researchers and framework developers quickly experiment with different options for PBT data generation. During Etna's development, we focused on a few key design principles, centered around usefulness, extensibility, and maintainability.

2.2.1 Evaluate for the ground truth, not for proxy metrics.

How do we measure the effectiveness of a generator? The software testing literature offers two main answers: *code coverage* and *mutation testing*. Code coverage is popular, but problematic: higher coverage does not always translate to better bug finding (Gopinath et al., 2014; Klees et al., 2018). We instead choose mutation testing (Jia and Harman, 2011), which measures the effectiveness of testing by artificially injecting mutations to the system under test and checking if testing is able to detect them. Mutations in the literature (Hazimeh et al., 2020; Klees et al., 2018; Zhang et al., 2022; Hritcu et al., 2016) fall on a spectrum from manually sourced to automatically synthesized. We opt for manual sourcing, allowing us to more readily maintain *ground truth* and ensure that every mutant violates some aspect of the property specification. Etna supports a terse syntax for incorporating these mutants into the systems under test. In §3, we detail the systems evaluated in this paper.

2.2.2 Use minimal, but precise interfaces.

A key challenge during Etna's development is that it needs to gather data about the testing effectivenes of a wide variety of existing frameworks, each of which reports such data in an ad-hoc non standardized manner: Most framework only report the number of inputs that were generated before a counterexample was found; very few offer timing statics; none offer a detailed breakdown of generation, testing, or minimization time. Similarly, most frameworks report the number of inputs that fail to satisfy a precondition as discards; some (like Rust's QuickCheck and Racket's RackCheck) do not. Most frameworks allow for setting a limit on the number of tests; very few allow for setting such time limits. Moreover, any printing of such data is optimized for human readability—with little to no consideration to how machine-readable this output is.

To tame this diversity, we settled on a set of metrics for PBT frameworks that are easily measurable, and on a precise output format that developers can adhere to: JSON defined using a schema that frameworks can validate themselves on. We built adaptors to that schema for frameworks across multiple languages (§7).[2]

2.2.3 Every Etna capability should be available to use manually

Etna acts as an orchestration mechanism that invokes testing tools, parses their results, and performs analysis on them. Such orchestration is, even with the best of intentions, fragile as a result of loosely coupled independently developed systems working together. In turn, as we found out from experience, any opaqueness in this process can result in unrecoverable failures. The final guiding principle of Etna is to ensure not just that such opaqueness doesn't exist, but that users can also reproduce the steps of any Etna experiment manually, if they so wish.

2.3 Terminology

Our mutation-testing based evaluation is built upon **tasks**: a mutant-property pair where the mutant causes the property to fail. As any given program can give rise to multiple tasks — it might need to satisfy multiple properties or be subjected to multiple mutants — we organize tasks into **workloads**. Each workload comes with several components: data type definitions; variant implementations of functions using these types; and a property specification of these functions.

We call a PBT paradigm at the level of a library a **framework**, which should contain functions for (a) constructing properties, (b) constructing generators, and (c) running tests. For instance, QuickCheck, QuickChick, SmallCheck and LeanCheck are all examples of frameworks. And we call a PBT paradigm at the level of how to use a framework to write generators a **strategy**. Examples of such strategies include type-based random generation, manually written bespoke generation, or exhaustive enumeration of the input space.

[2] `https://github.com/alpaylan/etna-cli/blob/main/PROTOCOL.md`

2.4 *Using* Etna

Etna is designed to be an extensible platform that flexibly accommodates new workloads, strategies, frameworks, and languages, built with inspiration from modern package managers such as Cargo [2018]. At its core is an experiment driver that provides three main pieces of functionality: (a) toggling between variant implementations in a directory of workloads; (b) compiling and running each strategy on each task; and (c) analyzing the results.

Users interact with Etna by creating such *experiments*: projects that can host multiple tests, pull workloads from Etna and modify them as the users wish, holding the raw data produced by experimentation, the analysis results and figures.

To replicate one of the existing experiments, the users can either reuse the experiment repository that holds the tests and experimentation scripts for this paper or create a new experiment from scratch, potentially reusing existing workload/strategy pairs.

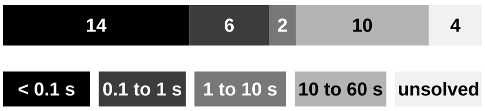
To evaluate a new generation strategy if the framework is already supported, they only need to extend one of the existing experiment scripts to include their new strategy. If the framework is not supported, they need to first implement an adaptor for the framework that provides the information the CLI requires, following the existing examples of Haskell, Rocq, OCaml, Racket, or Rust. The adapters follow a schema available at the Etna repository.

Finally, to contribute a new workload, users can implement the system under test just as they would ordinary code in a supported language. Then, they can then encode mutants via special comment syntax embedded within the implementation. For example, consider the following implementation of `insert`, together with a triggerable bug, in Haskell syntax:

```
insert k v E = T E k v E
insert k v (T l k' v' r)
  {-! -}
  | k < k' = T (insert k v l) k' v' r
  | k > k' = T l k' v' (insert k v r)
  | otherwise = T l k' v r
  {-!! insert_becomes_singleton -}
  {-!
  | k < k' = T (insert k v l) k' v' r
  | otherwise = T l k' v' (insert k v r)
  -}
  {- !-}
```

The correct (i.e. uncommented) implementation of `insert` ensures that the search tree invariant is maintained: every key in the left subtree of a node is smaller than its root, and every key in the right subtree is greater. In specially marked comments, a mutant is specified, which triggers a bug if enabled by not considering the case where the key being inserted is already present in the tree.

2.5 Analysis and Presentation



Though Etna supports customizable experiments, we choose a standard set of defaults for the experiments in this paper. We run each strategy on each task for a set amount of trials (10 unless otherwise specified) and with a set timeout (60 seconds). We then measure if the strategy was able to *solve the task*, i.e. find the injected bug in all trials within the given time frame. Multiple trials account for the non-determinism of random generation strategies, and results are simple averages unless indicated otherwise.

Our first attempts at presenting this data were hard to interpret: what does it mean, for example, if one strategy takes an average of two seconds and the other an average of three? Rather than present a slew of raw numbers, we wanted a data representation that captures a user's experience of interacting with PBT tools, so that visual differences in the representation correspond to tangible differences in performance. The figure above demonstrates our solution: a *task bucket* chart. For every strategy we classify tasks ranging from "solved instantly" to "unsolved", depicted with progressively lighter shades. For example, for the strategy/workload combination in the figure, 14 tasks are solved very quickly (the darkest shade) while four are not solved at all (the lightest).

In case a task bucket chart does not show enough detail, especially in head-to-head comparisons, we also support statistical analyses like Mann–Whitney U tests[3] (see §4.1).

3 Populating the Platform

We have integrated a number of PBT frameworks and workloads into Etna, both for our own use in §4 - §7 and for potential users to use and compare against.

3.1 Languages and Frameworks

Haskell is an obvious starting point: as the language that hosts QuickCheck, it is the lingua franca of PBT research. We focus on three Haskell frameworks: QuickCheck, of course; SmallCheck (Runciman et al., 2008), a competitor to QuickCheck that does enumerative testing; and LeanCheck (Braquehais, 2017), a more modern enumerative framework.

Our second language of choice is Rocq. While Haskell is blessed with many PBT frameworks, PBT in Rocq is built on a single framework: QuickChick (Lampropoulos and Pierce, 2018). However, QuickChick is a rich ecosystem that supports a variety of different strategies for input generation (Lampropoulos, 2018; Lampropoulos et al., 2018, 2019), so there is plenty to study and compare.

The third language we focus on is OCaml. Like Haskell, OCaml users can reach of a variety of random testing frameworks, from QuickCheck variants (QCheck (Cruanes, 2017) or base_quickcheck (Street, 2019)) to AFL-powered fuzzers like Crowbar (Dolan, 2017)).

[3] The Mann–Whitney U test is a nonparametric test that compares data samples from two different distributions. We use it here because it makes no assumptions about the distributions being compared.

For these three languages, we perform intra-language experiments comparing different generation strategies (§4, §5, §6). We will also show how to perform cross-language experiments with Etna (§7), using strategies both from these languages, as well as Racket (using its RackCheck framework (Popa, 2021)) and Rust (using (Gallant, 2014)). Etna's extensible design means that adding new languages is straightforward; we discuss languages that we plan to add to the platform in §8.

3.2 Workloads

Our initial set of workloads is drawn from three application domains that are of practical interest to the functional programming community and that have featured prominently in the PBT literature. These workloads feature in the following sections' experiments, although not every workload is used for every experiment. A detailed description of each workload, together with a list of properties and associated mutants, can be found in the repository.

Data Structures. The first workload focuses on a functional data structure that is ubiquitous in the literature: binary search trees. Multiple PBT papers have focused on BST generation, including John Hughes's *How to Specify It!* [2019], an extended introduction to specifying properties using QuickCheck. Our BST workload ports the mutations and properties from that paper. The second workload focuses on another popular functional data structure, red-black trees, including self-balancing insertion and deletion operations that are notoriously easy to get wrong. RBTs have also been studied in the PBT literature (Lampropoulos et al., 2017; Runciman et al., 2008; Mista and Russo, 2019; Klein and Findler, 2009). Our RBT workload combines the BST mutants with additional mutants that focus on potential mistakes when balancing or coloring the tree.

Lambda Calculi and Type Systems. The third workload centers around a DeBruijn index based implementation of the simply typed lambda calculus with booleans. Bespoke generators for producing well-typed lambda terms is a well studied problem in the literature (Palka et al., 2011; Midtgaard et al., 2017), while the mutations for STLC included in our case study are drawn from the appropriate fragment of a System F case study (Goldstein et al., 2021), dealing mostly with mistakes in substitution, shifting, and lifting. For a more complicated fourth workload $F_<$: revolving around calculi and type systems, we turn to the full case study of Goldstein et al. (2021) and extend it with subtyping. This allows for significantly more complex errors to be injected (such as those dealing with type substitution, shifting, or lifting). Bespoke generators for System F have been the subject of recent work (Goldstein et al., 2021; Hoang et al., 2022) and can be straightforwardly extended to handle subtyping. The fifth workload involves a parser and pretty-printer for Lu, a language based on Lua; the implementation of Lu was drawn from a Haskell course at the University of Pennsylvania. We specify correctness through a round-trip property: printing a valid Lu expression and then parsing it should result in the original expression.

Security. The sixth and final workload focuses on a security domain: information flow control. The IFC case study, introduced by Hritcu et al. (2013, 2016), explores the effectiveness of various bespoke generators for testing whether low-level monitors for abstract machines enforce noninterference: differences in secret data should not become publicly visible through execution. Violations in the enforcement policies are introduced by systematically weakening security checks or taint propagation rules, exploring all possible ways of introducing such violations.

4 Experiments: Haskell

We next report on our experience using Etna to probe different aspects of testing effectiveness. Our first set of observations are on the PBT frameworks and strategies available in Haskell.

4.1 Comparing Frameworks

In the first experiment, we assess the "out of the box" bug-finding abilities of three Haskell frameworks — QuickCheck, SmallCheck, and LeanCheck. We examine four strategies. For the *bespoke* strategy, we manually write a QuickCheck generator that always produces test inputs that satisfy the property's precondition. This serves as a "topline" for the other strategies: a high-effort generator that solves all of the tasks easily. The other three strategies — one per framework — are all *naive*. The QuickCheck strategy uses the generic-random library to derive its generator automatically, with constructors chosen at each step with uniform probability and a size parameter that decreases on recursive calls to ensure termination. For the enumerative frameworks, SmallCheck and LeanCheck, we use combinators that follow the type structure.

We evaluate these strategies against four workloads: BST, RBT, STLC, and $F_{<}$:.

Results. We visualize the results of this experiment in Figure 1. Some points to note:

The bespoke strategy outperforms the naive strategies along multiple axes. For example, looking at the naive QuickCheck strategy (the others are similar), the bespoke strategy solved all tasks, while the naive strategy failed to solve 43 tasks. Among tasks that both strategies solved, using a Mann–Whitney U test with $\alpha = 0.05$, we find that the bespoke strategy's average time to solve a task was (statistically) significantly lower in 83 out of 124 tasks and the average valid inputs to solve a task were lower for 89 out of 124 tasks. That is, the bespoke strategy found more bugs, more quickly, and with better quality tests.

Between the two enumeration frameworks, LeanCheck substantially outperforms SmallCheck on these workloads. LeanCheck had an 82% solve rate, while SmallCheck's was only 35%. On one BST task, LeanCheck found the bug in about a hundredth of a second on average, while SmallCheck required 26 seconds. One reason for these differences may be that SmallCheck attempts to enumerate larger inputs much earlier. In the first thousand binary trees, SmallCheck produces trees with up to ten nodes, while LeanCheck only reaches four nodes. Unsurprisingly, it is harder for larger trees to satisfy the BST invariant — only 1% of these thousand SmallCheck trees are valid, compared to 13% of the LeanCheck trees. And across all workloads, we can calculate the rate at which they enumerate test inputs, by aggregating over the tasks that they both solved and dividing by the total number of tests by the total time spent. We find that LeanCheck produces over a hundred times more tests per second than SmallCheck.

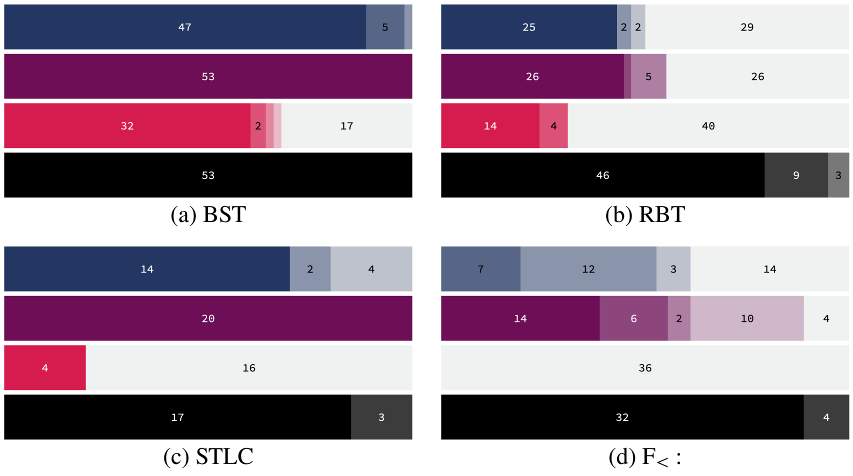


Fig. 1: Effectiveness of Haskell generation strategies on four workloads.
🟦 = Naive QuickCheck, 🟪 = Naive LeanCheck,
🟥 = Naive SmallCheck, ⬛ = Bespoke QuickCheck.

LeanCheck also outperforms naive QuickCheck. It is illuminating to consider failed tasks that were *partially solved*: the bug was found in at least one trial and not found in at least one trial. There is one such task for LeanCheck and 14 for QuickCheck. For LeanCheck's partially solved task, the inputs required are the same for each trial, but the time fluctuates between 55 and 65 seconds. That is, this is a situation where a task nears — and sometimes exceeds — what LeanCheck can reach with the one minute time limit. QuickCheck's partially solved tasks are also interesting. Of the 13 that LeanCheck solves but QuickCheck does not, 10 are partially solved by QuickCheck. This suggests that there are situations where a deterministic approach may be more reliable than a random alternative: LeanCheck solves these tasks consistently and relatively quickly, while QuickCheck sometimes takes less than a second, sometimes nearly a minute, and sometimes times out.

Similarly, in STLC, naive LeanCheck solves three more tasks within the first bucket than the bespoke strategy. Upon closer inspection, these are tasks that the bespoke strategy sometimes solves in under 100 inputs but sometimes requires over 10,000 inputs, leading to an average slightly above the 0.1 second threshold; as before, LeanCheck does not experience this variability.

A note about memory usage. LeanCheck is documented[4] to be memory intensive, especially when run for prolonged periods of time, as we do here. Our experiments using LeanCheck were conducted on a server with plenty of memory, allowing us to complete trials without issues. Future work might consider the relative space complexities of different frameworks.

[4] https://github.com/rudymatela/leancheck/blob/master/doc/memory-usage.md

4.2 Exploring Sized Generation

We next explore the sensitivity of bug-finding to various parameters, starting with input size.

A significant part of generator tuning is ensuring that the generated inputs are well sized. Conventional wisdom in random testing posits that there is a "combinatorial advantage" to testing with large inputs, since they can exercise many program behaviors at once; tools like *QuickCover* (Goldstein et al., 2021) capitalize on this notion to make testing more efficient. But are large inputs *always* better? We used our BST workload to investigate.

We conducted this experiment on the QuickCheck framework, using a bespoke strategy to focus attention on the quality of the distribution of *valid* inputs. We used a generator from Hughes (2019), which generates a list of keys and then inserts each key into the tree, because it gives precise control over final tree sizes. We choose the keys for a n-node tree from a range of integers 1 to $2n$. This range is large enough to allow for sufficient variety in shape and content but not so large that a randomly generated key is unlikely to be in the tree.

We then measured the bug-finding effectiveness of the generator at different sizes n. Thanks to Etna's flexibility, we could vary the size in the script and otherwise treat this experiment as we would any other where we wanted to compare several strategies.

Results. Figure 2 plots the size of the tree versus the average number of inputs to solve a task; each trace represents one task. Some noteworthy traces, highlighted in black, are discussed below.

Larger trees can be worse *for bug-finding, for properties that rely on dependencies between their inputs.* We found that, for BST, small trees were generally sufficient to find bugs, and performance got significantly worse for some tasks as trees got larger.

For example, task #1, which has the steepest upward curve, involves a mutant where the delete function fails to remove a key unless that key happens to be the root. The property takes one tree and two keys as inputs and checks that removing the keys in either order leads to the same result. Together, these mean that the task is only solvable when one key `k` is the root of the tree and the other key `k'` becomes the root after deleting `k`. The probability of satisfying this condition decreases as the size of the tree increases, so larger trees take more inputs to solve this task.

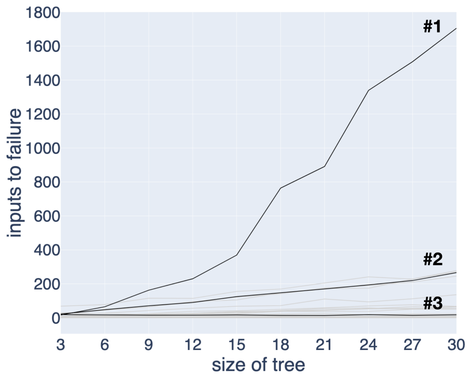


Fig. 2: Number of generated inputs (averaged over 100 trials) to solve each BST task, as input size increases from three to 30 nodes.

Task #2 is a similar story. It takes a tree and two key-value pairs; this time, the task is only solvable when the two keys are the same (and the two values are different), a probability that is inversely proportional to the size of the tree. These two tasks demonstrate situations where the inputs to a property need to be related in a mutant-specific way, and large trees are less likely to satisfy this dependency relationship.

Not all tasks with dependencies between their inputs are harder to solve with larger trees. Unlike #1 and #2, the curve for task #3 is mostly flat, even though it has a similar dependency. The mutant here causes the `union` operation to fail by occasionally preferring the wrong value if both trees contain the same key; the property takes a key `k` and two trees and checks that `k` exists in the union of the trees when it exists in either tree. Since this mutant causes problems with keys that appear in both trees, the property only fails when `k` is in the input trees. That is, there *is* a dependency between the inputs, but this dependency does *not* scale with the size of the tree.

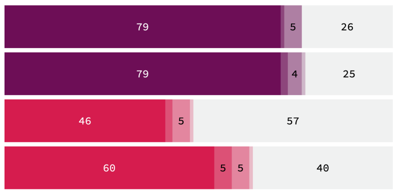
Discussion. We have seen that larger inputs sometimes not only fail to provide a combinatorial advantage but in fact can provide a dependency disadvantage. The size of the main input — in this case, the tree — cannot be evaluated in a vacuum. Instead, the particulars of the mutant and property can lead to dependencies between the property inputs that must be satisfied in order to detect the mutant. Our size exploration is thus a cautionary tale: PBT users should not naively expect that larger inputs are better, especially for properties with multiple inputs.

This exploration suggests a few recommendations for improving both testing frameworks and individual users' choices of properties. *(1) Do not treat property inputs as independent.* The difficulties with the above properties arise, in part, because QuickCheck automates generation of multiple inputs by assuming that each input can be generated independently — but treating inputs independently can lead to unintuitive testing performance. Frameworks like Hedgehog explicitly avoid introducing a generator typeclass so as to force users to build generators by hand; our results lend credence to that design choice. *(2) Think carefully about properties with multiple inputs.* Testers should prefer properties with fewer inputs where possible. When this is infeasible, testers should think carefully about potential interactions between their property's inputs and write generators that take those interactions into account.

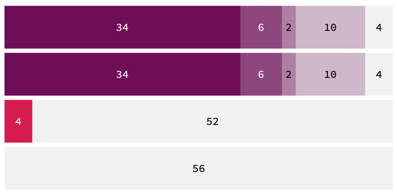
4.3 Enumerator Sensitivity

Papers about enumeration frameworks sometimes speak of enumeration as a kind of exhaustive testing — validating the program's behavior within a "small scope" (Andoni et al., 2002). But realistic testing budgets often mean that exhausting all inputs up to some interesting size or depth is not possible: enumeration is *expensive*. Thus, the actual performance of enumeration frameworks like SmallCheck and LeanCheck is impacted by the specific order in which values are enumerated. In this section we examine some factors that, perhaps unexpectedly, impact bug-finding performance.

There are many axes along which order could vary. We have explored two: the order of the inputs to each property and the order of constructors in an algebraic data type. We conduct this experiment on SmallCheck and LeanCheck, using the BST and RBT workloads where many of their properties have multiple inputs, including a combination of `Trees` and `Int` keys. One enumeration strategy uses the default properties, with the trees passed in first, and one uses properties with the trees last — for example, `(Tree, Tree, Int)` vs. `(Int, Tree, Tree)`.



(a) Enumerator performance on the BST and RBT workloads when the trees are at the start of the properties (top rows) versus when they are at the end (bottom rows).



(b) Enumerator performance on the STLC and FSUB workloads when the constructor enumeration order aligns with the definition of the data type (top rows) versus when the orders are reversed (bottom rows).

Fig. 3: 🟪 = Naive LeanCheck, 🟥 = Naive SmallCheck.

Results. We count the number of tasks that are solved by the same framework under exactly one of the two orderings. The results are shown in Figure 3.

For LeanCheck, the tree-last strategy solved one additional task that the tree-first strategy did not (completing in about 38 seconds instead of timing out at 60). For SmallCheck, the tree-last strategy solved 17 more tasks than tree-first, taking between 0.002 and 7 seconds. The low end is especially remarkable: simply by enumerating `(Int, Tree, Tree)`s rather than `(Tree, Tree, Int)`s, SmallCheck finds a counterexample almost immediately instead of timing out.

Discussion. A deeper dive into the enumeration frameworks to explore these differences fully would be worthwhile, but what jumps out even from these simple experiments is the question of *sensitivity*. The potentially pivotal role of enumeration order in the success or failure of these strategies means that users of these enumerative frameworks need to be careful of configuration settings that would be immaterial in their random counterparts. As a meta point, we put the tree data types at the front of each property as a matter of convention; it was not until much later that we realized the inadvertent effect on the performance of the enumerators!

5 Experiments: Rocq

After focusing on the multi-framework landscape of Haskell in the previous section, we now turn our attention to the single-framework but multi-strategy landscape in Rocq. As discussed in §3.1, PBT in Rocq revolves around QuickChick (Lampropoulos, 2018), which, in addition to the type-based and bespoke strategies that we explored in Haskell, provides two additional options: a *specification-driven* strategy that derives correct-by-construction generators from preconditions in the form of inductive relations (Lampropoulos et al., 2018) and a *type-driven fuzzer* strategy that combines type-based generation with mutation informed by AFL-style branch coverage to guide the search toward interesting parts of the input space (Lampropoulos et al., 2019).

Both papers exemplify the lack of performance comparisons across approaches discussed in the introduction. First, Lampropoulos et al. (2018) is evaluated in a toy IFC example, where only the throughput of generators is measured against that of a bespoke generator; there is no measurement of the effectiveness of the strategy in finding bugs. On the other

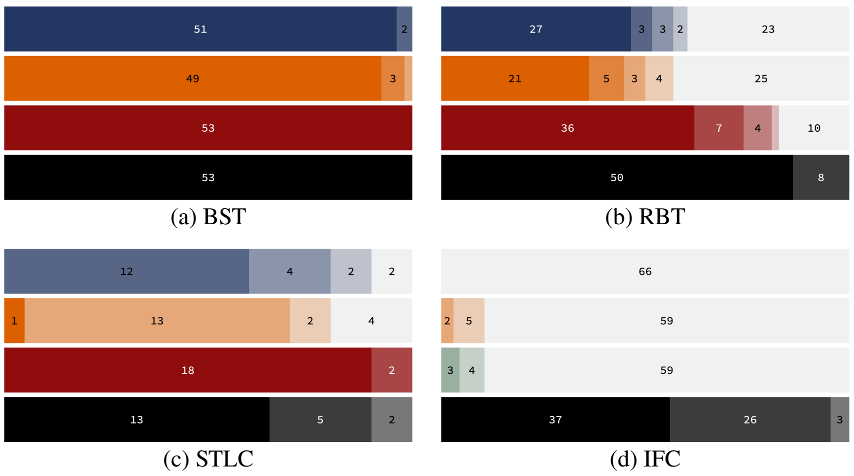


Fig. 4: Effectiveness of Rocq generation strategies on four workloads.
🟦 = Type-based generator, 🟧 = Type-based fuzzer,
🟥 = Specification-based generator ((a) - (c) only), 🟩 = Variational fuzzer ((d) only),
⬛ = Bespoke generator.

hand, FuzzChick (Lampropoulos et al., 2019) is evaluated in the more realistic IFC workload of Hritcu et al. (2016) that we will reuse later in this section, with systematically injected mutations that break the enforcement mechanism of a dynamic monitor. Still, multiple aspects of their strategies were left unevaluated, including their performance on any other workload.

5.1 Comparison of Fuzzers, Derived Generators, and Handwritten Generators

We aim to fill the evaluation gaps described above. How do QuickChick's newer strategies compare with the more established bespoke and type-based ones? In particular, are they effective at uncovering bugs across disparate workloads?

We again use the BST, RBT, and STLC workloads, along with a more complex case study, IFC, pulled from the FuzzChick paper. For the first three case studies, inductively defined specifications are widely available (e.g. in Software Foundations (Pierce, 2018)); for IFC, such specifications do not exist, so the specification-driven generators of Lampropoulos et al. does not apply.

Results. In Figure 4, we visualize the results of the experiments with a task bucket chart. Results for the simple BST workload (Figure 4a) establish a baseline level of confidence for all four methods, as they are all able to solve most tasks quickly. Indeed, most of the tasks are solved by all methods within 0.1 seconds (the darkest color), with the exception of the *type-based fuzzer*, which falls short on a few tasks.

Specification-derived strategies are on par with bespoke ones. In the harder RBT workload, with its much more complex invariant, there is a clear performance gap between type-driven strategies (type-based generator and type-based fuzzer) and precondition-driven methods (specification-based generator and bespoke generator). Precondition-driven methods are able to solve more tasks under 0.1 seconds than type-driven methods are able to solve within a 60 second timeout. The type-based generator fails to solve 23 tasks, and the type-based fuzzer fails to solve 25. The bespoke generator solves all tasks in under ten seconds, and the specification-based generator solves all but 10 tasks. We see a similar pattern in the STLC workload, with the precondition-driven methods outperforming the type-driven ones.

Fuzzers exhibit more variance but outperform type-driven methods for sparse preconditions. For the IFC workload, the only precondition-driven strategy is the bespoke generator, which emerges as a clear winner: noninterference is a property with a *very* sparse precondition, and type-based methods are basically unable to generate valid inputs. For this particular workload, we included another fuzzing variant borrowed from the original paper that introduced FuzzChick (Lampropoulos et al., 2019) to strengthen the connection to the existing literature: rather than generating a pair of input machines completely at random and then fuzzing the pair (as in the type-based fuzzer approach), we generate one input machine and copy it to create a pair that is indistinguishable by default. The two fuzzers, *type-based fuzzer* and *variational fuzzer*, have a clear advantage over the pure type-based generation approach: the ability to guide generation allows fuzzers to discover parts of the input space that naive type-based generation are simply unable to reach.

Yet fuzzers are not reliable in this sense, as Figure 5 shows: if we include *partially solved* tasks, fuzzers outperform their generator counterparts. This further clarifies the picture painted by the first set of comparisons. Fuzzers may get stuck following program paths that will not lead to interesting revelations, but sometimes discover paths that a traditional type-based generator could never hope to reach. In particular, roughly 30 tasks are solved *at least once* through 10 runs (Figure 5), but less than 10 tasks are fully solved (Figure 4d).

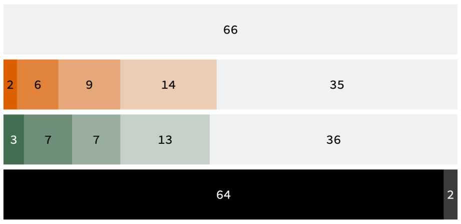


Fig. 5: Tasks solved within the timeout in one or more trials.
Empty = Type-based generator.
🟧 = Type-based fuzzer,
🟩 = Variational fuzzer,
⬛ = Bespoke generator.

Another interesting observation is that even though fuzzers typically spend more time per generated input, as the underlying types are more complex and large, mutating the input takes less time than generating a new one. For IFC, the type-based generator takes four times longer per input than the type-based fuzzer.

5.2 Validation and Improvement of Fuzzers

Despite its minimal evaluation, the conclusion of Lampropoulos et al. (2019) seems to hold — that is, FuzzChick shows promise compared to type-based approaches, but has a long way to go before catching up with the effectiveness of precondition-driven ones. This led us to wonder, *could we further improve the performance of FuzzChick using* E*tna?*

We focused on two different aspects of fuzzing: *size* and *feedback*. FuzzChick's generation strategy started small but quickly increased to quite large sizes, relying on the idea of "combinatorial advantage" discussed in §4.2 — i.e., that larger inputs contain exponentially many smaller inputs and are therefore more effective for testing. As we saw there, that is not always the case. After realizing this, we switched to a more gently increasing size bound which led to significant improvements in terms of throughput, positively impacting our bug-finding ability.

With respect to feedback, by using Etna to evaluate FuzzChick across multiple workloads we were able to identify, isolate, and fix a bug that caused it to saturate the seed pool with uninteresting inputs. FuzzChick (like Zest (Padhye et al., 2019)) keeps two seed pools: one for valid and one for invalid inputs. FuzzChick's bug applied to the latter one, and was hidden from its authors as the variational fuzzer strategy they employed readily gives access to valid inputs (which are prioritized).

Results. Figure 6 demonstrates the bug-finding capabilities of the original (top) and tuned (bottom) versions of FuzzChick across the new workloads. The tuned version clearly outperforms the original in all cases—and is what was used in the previous section.

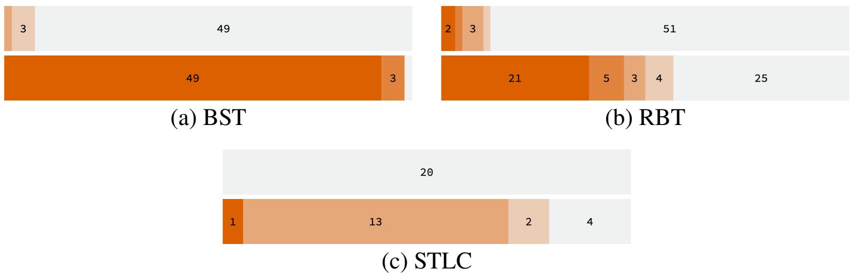
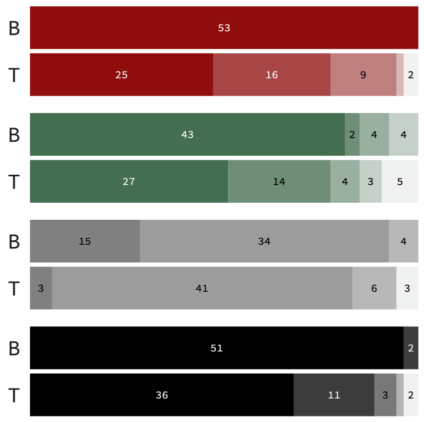


Fig. 6: Comparison of the original FuzzChick (top) with the tuned one (bottom).

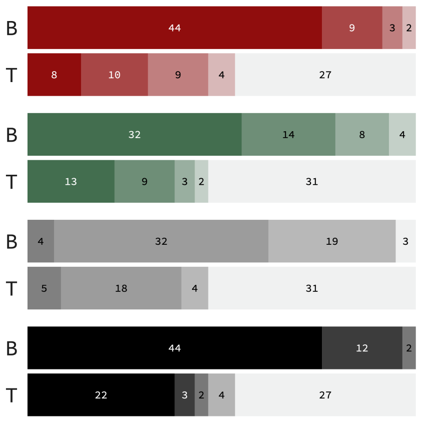
Hardening QuickChick's Implementation. In our experimentation with Etna, we stress tested some of QuickChick's features in ways that occasional user interactions could not hope to reach. One particular bug stood out: The main fuzzing loop of FuzzChick is written in Gallina and uses a natural number fuel to satisfy the termination checker. That natural number is extracted as an OCaml integer for efficiency purposes. However, when extracted, a pattern match becomes a call to an eliminator:

```
Fixpoint loop fuel ... :=    let rec loop fuel ... = (fun f0 fS n ->
  match fuel with              if n = 0 then f0 () else fS (n-1))
  | O => (* base *)              (fun () -> (* base *))
  | S fuel' => (* rec *)         (fun fuel -> (* rec *))
  end.                           fuel
```

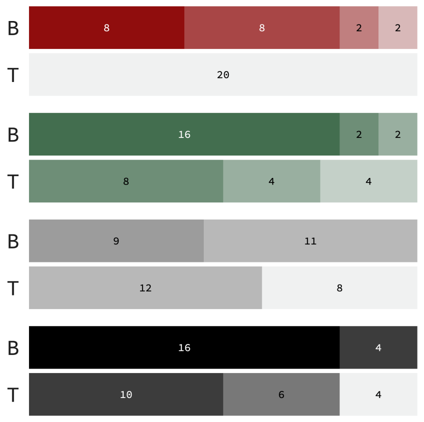
Can you spot the problem? The extracted version is no longer identified by the OCaml compiler as tail recursive... which means that when Etna used large fuel values, it led to stack overflows!



(a) BST



(b) RBT



(c) STLC

Fig. 7: Effectiveness of OCaml generation strategies on three workloads. The first and second buckets for each framework represent the bespoke and type-based generators respectively. 🟥 = QCheck, 🟩 = Crowbar (Random), ▨ = Crowbar (AFL), ⬛ = Base_quickcheck.

6 Experiments: OCaml

Whereas QuickChick is the defacto property-based testing framework in Rocq, programmers in OCaml have a choice of frameworks: QCheck, offering a standard QuickCheck-like monadic API for writing generators; Crowbar, offering fuzzing capabilities as a wrapper around AFL; and Base_quickcheck, leveraging the Core standard library replacement.

For the workloads, we ported the three basic ones from the previous sections—BST, and RBT, and STLC—to OCaml. As neither of the three frameworks provides machinery for automatically deriving either type-based (in the style of Haskell's generic-random) or specification-based (in the style of Rocq's QuickChick) generators, we also ported a type-based and a bespoke generator to each framework.

Controlling for Size As our experiments earlier in the paper showed, the size of generated inputs can have a significant impact on the effectiveness of testing. However, the three OCaml frameworks offer vastly diverse default size distributions. In particular, the one for Base_quickcheck was similar to the effective ones from §4.2, so we left it as is. For QCheck, to avoid generating a distribution that is heavily bimodal (i.e. with many trees containing one or two nodes and most others containing several thousand) or where the range of integers is too large (to provoke bugs that rely on collisions), we used the variants of QCheck's integer generators that focus on smaller integer ranges. Finally, for Crowbar, the default behavior of its `list` generator (which we use as a basic building block in multiple strategies) produces very small lists[5]. As a result, we re-implemented a list generator using the rest of Crowbar's API to construct longer lists.

Figure 7 shows the results in bucket-chart form, where we used type-based and bespoke generators for each of the frameworks, using Crowbar both with its purely random and AFL-powered backends. For these workloads, the Core-library powered Base_quickcheck bespoke generators outperform the other frameworks in almost all situations. The best generator that we could write using Crowbar's interface, performed the worst out of the strategies we tried. However, that does not mean that Crowbar as a framework is less effective: rather, just as the FuzzChick case (§5.1), if one takes the effort to handcraft bespoke generators that satisfy a property's precondition by construction, coverage-guided fuzzing only adds overhead for minimal gain.

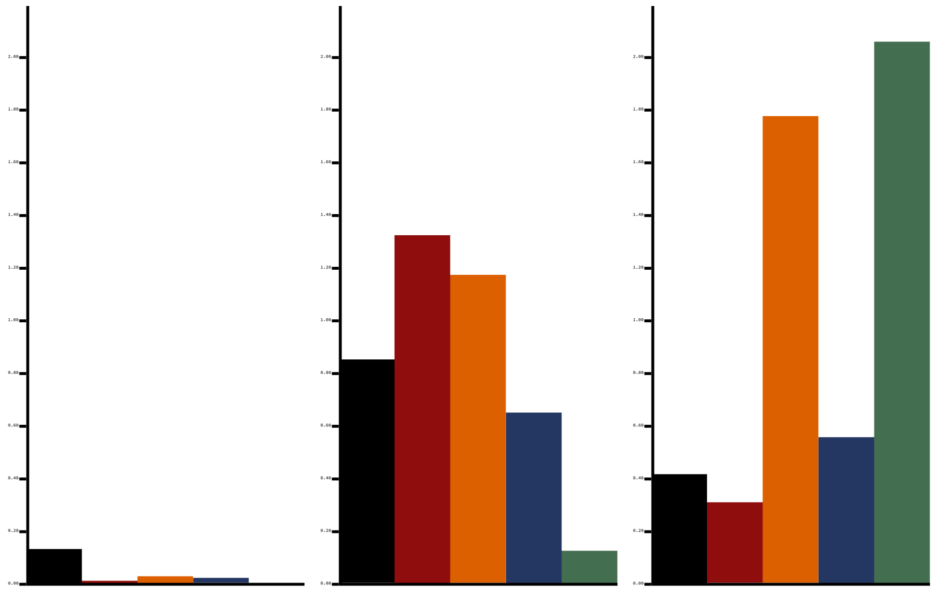
7 Experiments: Cross-Language Comparison of PBT Frameworks

As we demonstrated throughout this paper, Etna allows for running complex experiments that can provide powerful insights to property-based testing practitioners or framework developers. However, the scope of the experiments we have provided thus far has been limited to the level of a single programming language. Given two testing frameworks or generation strategies within the same language, we can measure and compare their bugfinding performance across the different Etna workloads implemented for that language.

Yet, such inter-language experimentation does not encompass the current practice of property-based testing, where generation strategies in one language are used to test systems in another. For one example, a specification-derived generator for well-typed System F terms written in Rocq was used to test a higher-order blockchain language implemented in OCaml (Hoang et al., 2022); for another, a bespoke generator for file-system interactions written in Erlang were used to test Dropbox's Python-based file synchronization service (Hughes et al., 2016).If we want Etna to enable prototyping of and experimentation with effective testing strategies in practice, we need to be able to compare the performance of such strategies across languages.

To that end, we developed support to perform cross-language experiments in Etna, decoupling generation of inputs and testing of properties. On the generation side, each aspiring Etna user must implement their generation strategy, just like before, but instead

[5] Given an input size, it will generate an empty list 50% of the time, or a cons cell whose tail is recursively generated with the size parameter halved. In practice, that means vanishingly few lists of length more than 5 will be generated even for large input sizes.



(a) BST (b) RBT (c) STLC

Fig. 8: Generation time to failure for bespoke generators written in different languages. ⬛ = Haskell – QuickCheck, 🟥 = Rocq – QuickChick, 🟧 = OCaml – QCheck, 🟦 = Racket – Rackcheck, 🟩 = Rust – QuickCheck.

of linking it directly with a framework-dependent way of executing a test, they need to only output a list of serialized inputs together with the time it took to generate each one. On the property end, we have created *runners* for the BST, RBT, and STLC workloads that can read serialized inputs from the command line to test their related properties with. Etna can then perform its analysis in a language-agnostic manner: it can give precise, fine-grained feedback about the performance of each generation strategy (without aggregating generation, execution, or shrinking times together).

As a beneficial side effect of decoupling strategies from workloads, the extensibility of Etna is greatly improved. Integrating a new language within Etna no longer *requires* porting all of its workloads in yet another language—although that is still an option. Instead, workloads only need to be implemented once, in any language that can deserialize inputs to interface with Etna's API. And strategies written in a previously unsupported language need only implement a generator and a serializer to use the existing workloads for experimentation. Moreover, the decoupled approach to generation and testing allows for quick validation of ports of strategies and workloads to new languages: if running the same generator produces different results in the cross-language mode than inter-language mode, that points towards an error in the implementation of the workload or the strategy.

To demonstrate Etna's new capabilities, we pit bespoke generators written in Haskell, Rocq, Ocaml, Racket, and Rust against the three workloads. The results are shown in Figure 8. As we are dealing with tuned bespoke generators, bucket charts are not ideal for discerning differences—all generators find basically all bugs, quickly. Instead, the top of the figure shows the *total generation time to failure* per-framework per-workload.

Despite all bespoke generators implementing the same strategy in principle, there are slight differences in efficiency: the way each framework manipulates the size of generated inputs across runs and the performance characteristics of the language both affect the end result. Still, all strategies are relatively close, with Rust's QuickCheck, for example, being simultanously the fastest in the BST and RBT workloads, and the slowest in STLC (we conjecture because of excessive creation of expensive closure's in binds).

Another takeway from Figure 8 is the difficulty of the workloads themselves: BST is comprised of 52 tasks, and the slowest generator (Haskell's QuickCheck) still finds all 52 injected bugs in 130ms; on the other hand, STLC is comprised of only 20 tasks, but the fastest generator (Rocq's QuickChick) takes 300ms, where the slowest takes just over 2 seconds.

Moving forward, armed with cross-language evaluation capabilities and only needing to implement runners once, we hope to rapidly expand the number of workloads and frameworks available for experimention.

8 Related and Future Work

The future directions we imagine for Etna are inspired by related work in the literature. Thus, we discuss both related and future work together in this section.

Etna's name, referencing every crossword-puzzler's favorite Italian volcano, was inspired by two existing benchmark suites in the fuzzing space: LAVA (Dolan-Gavitt et al., 2016) and Magma (Hazimeh et al., 2020). Both provide a suite of workloads that can be used to compare different fuzzing tools: LAVA's workloads consist of programs with illegal memory accesses that are automatically injected, while Magma relies on real bugs forward-ported to the current versions of libraries. More recently, FixReverter (Zhang et al., 2022) offered a middle ground, generalizing real bug-fixes into patterns and applying them to multiple locations in a program. Etna is different from these suites in a few ways. First, Etna aims to be a platform for exploration and evaluation rather than a rigid set of benchmarks. Thus, we do not claim that Etna's workloads are complete — instead, we intend for users to add more over time. Additionally, evaluating fuzzing is quite different from evaluating PBT, since PBT is expected to run for less time on programs with higher input complexity. This means that Etna's measurement techniques and workload focus must necessarily be different from LAVA's or Magma's. Still, there are ideas worth borrowing from these suites: fuzzing benchmarks generally record code-coverage information, which we plan for Etna to eventually offer as well.

Besides LAVA and Magma, there is a massive literature of Haskell and Rocq papers from which we will continue to draw both workloads and frameworks. With the help of the community, we hope Etna will eventually include frameworks like: Luck (Lampropoulos et al., 2017), a language for preconditions from which generators can be inferred; FEAT (Duregård et al., 2012), an enumerator framework focusing on uniformity; tools for deriving better Haskell generators (Mista and Russo, 2021, 2019); and specification-driven enumerators for QuickChick (Paraskevopoulou et al., 2022).

Outside of the currently supported languages and frameworks, there are yet more opportunities for growth.We will solicit framework maintainers and researchers to add support for other languages such as Scala (SciFe (Kuraj and Kuncak, 2014; Kuraj et al., 2015) and ScalaCheck (Nilsson, 2019)), Erlang (QuviQ (Arts et al., 2008) or PropEr (Papadakis and Sagonas, 2011)), or Isabelle (Bulwahn, 2012*a*,b).

Finally, the presentation back end of ETNA is fit-for-purpose, but we intend to do further research into the best possible ways to visualize PBT results. Consulting experts in human-computer interaction, we plan to use tools like Voyager (Wongsuphasawat et al., 2017) to explore which kinds of outcome visualizations real users of ETNA want. At the very least, integrating ETNA into a Jupyter notebook (Jupyter, 2023) and providing hooks into a powerful graphics engine like Vega-lite (Satyanarayan et al., 2017) would make it easier for users to experiment with visualizations.

9 Conclusion

We designed ETNA to meet a concrete need in our research — we needed a clear way to convince ourselves and others that the PBT tools we build are worth pursuing. ETNA provides that, with an extensible suite of interesting workloads and the infrastructure necessary to validate and refine designs against them. In §4 - §7, we originally set out to answer straightforward questions about whether X is better than Y, and while we did get feedback about general trends, we also uncovered some unexpected nuances of the testing process. PBT-curious readers may have further questions building upon and extending beyond our explorations. ETNA is there for you!

Acknowledgments

This work was supported by the NSF under award #1955610 *Bringing Python Up to Speed* and under #2145649 *CAREER: Fuzzing Formal Specifications*.

References

Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. 2002. Evaluating the ”Small Scope Hypothesis”. (10 2002).

Thomas Arts, Laura M. Castro, and John Hughes. 2008. Testing Erlang Data Types with QuviQ QuickCheck. In *7th ACM SIGPLAN Workshop on Erlang* (Victoria, BC, Canada). ACM, 1–8. https://doi.org/10.1145/1411273.1411275

Rudy Matela Braquehais. 2017. *Tools for Discovery, Refinement and Generalization of Functional Properties by Enumerative Testing*. Ph.D. Dissertation. University of York.

Lukas Bulwahn. 2012a. The New Quickcheck for Isabelle - Random, Exhaustive and Symbolic Testing under One Roof. In *2nd International Conference on Certified Programs and Proofs (CPP) (Lecture Notes in Computer Science, Vol. 7679)*. Springer, 92–108.

Lukas Bulwahn. 2012b. Smart Testing of Functional Programs in Isabelle. In *18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) (Lecture Notes in Computer Science, Vol. 7180)*. Springer, 153–167.

Koen Claessen, Jonas Duregård, and Micha l H. Pa lka. 2014. Generating Constrained Random Data with Uniform Distribution. In *Functional and Logic Programming (Lecture Notes in Computer Science, Vol. 8475)*. Springer, 18–34. https://doi.org/10.1007/978-3-319-07151-0_2

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 268–279. https://doi.org/10.1145/351240.351266

Simon Cruanes. 2017. QuickCheck Inspired Property-Based Testing for OCaml. https://github.com/c-cube/qcheck/.

Stephen Dolan. 2017. Property Fuzzing for OCaml. https://github.com/stedolan/crowbar.

Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. 110–121. https://doi.org/10.1109/SP.2016.15

Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: Functional Enumeration of Algebraic Types. In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark) *(Haskell '12)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2364506.2364515

Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *24th European Symposium on Programming (Lecture Notes in Computer Science, Vol. 9032)*. Springer, 383–405.

Justin Frank, Benjamin Quiring, and Leonidas Lampropoulos. 2024. Generating Well-Typed Terms that are not "Useless". In *Proceedings of the ACM on Programming Languages (PACMPL), Volume POPL*. https://doi.org/10.1145/3632919

Andrew Gallant. 2014. quickCheck: Automated property based testing for Rust (with shrinking). https://github.com/BurntSushi/quickcheck.

Harrison Goldstein, John Hughes, Leonidas Lampropoulos, and Benjamin C. Pierce. 2021. Do Judge a Test by its Cover: Combining Combinatorial and Property-Based Testing. In *Proceedings of the European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-030-72019-3_10

Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code Coverage for Suite Evaluation by Developers. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 72–82. https://doi.org/10.1145/2568225.2568278

Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3, Article 49 (Dec. 2020), 29 pages. https://doi.org/10.1145/3428334

Tram Hoang, Anton Trunov, Leonidas Lampropoulos, and Ilya Sergey. 2022. Random Testing of a Higher-Order Blockchain Language (Experience Report). In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*. https://doi.org/10.5281/zenodo.6778257

Catalin Hritcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. Testing Noninterference, Quickly. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2544174.2500574

Catalin Hritcu, Leonidas Lampropoulos, Antal Spector-Zabusky, Arthur Azevedo Amorim, Maxime Denes, John Hughes, Benjamin C. Pierce, and Dimitrios Vytiniotis. 2016. Testing Noninterference, Quickly. In *Journal of Functional Programming (JFP)*. https://doi.org/10.1017/S0956796816000058

John Hughes. 2019. How to Specify It! - A Guide to Writing Properties of Pure Functions. In *Symposium on Trends in Functional Programming*. https://doi.org/10.1007/978-3-030-47147-7_4

John Hughes, Benjamin C. Pierce, Thomas Arts, and Ulf Norell. 2016. Mysteries of Dropbox: Property-Based Testing of a Distributed Synchronization Service. In *International Conference on Software Testing, Verification and Validation (ICST)*.

Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. https://doi.org/10.

1109/TSE.2010.62

Project Jupyter. 2023. Project Jupyter. https://jupyter.org

George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

Casey Klein and Robert Bruce Findler. 2009. Randomized Testing in PLT Redex. In *Workshop on Scheme and Functional Programming (SFP)*.

Ivan Kuraj and Viktor Kuncak. 2014. SciFe: Scala framework for efficient enumeration of data structures with invariants. In *Proceedings of the Fifth Annual Scala Workshop*. ACM, 45–49. https://doi.org/10.1145/2637647.2637655

Ivan Kuraj, Viktor Kuncak, and Daniel Jackson. 2015. Programming with Enumerable Sets of Structures. In *OOPSLA*. https://doi.org/10.1145/2814270.2814323

Leonidas Lampropoulos. 2018. *Random Testing for Language Design*. Ph.D. Dissertation. University of Pennsylvania.

Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li yao Xia. 2017. Beginner's Luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, (POPL)*. https://doi.org/10.1145/3009837.3009868

Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage Guided, Property Based Testing. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*. https://doi.org/10.1145/3360607

Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2018. Generating Good Generators for Inductive Relations. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3158133

Leonidas Lampropoulos and Benjamin C. Pierce. 2018. *QuickChick: Property-Based Testing In Coq*. Electronic textbook.

Fredrik Lindblad. 2007. Property Directed Generation of First-Order Test Data. In *8th Symposium on Trends in Functional Programming (Trends in Functional Programming, Vol. 8)*. Intellect, 105–123.

Andreas Löcher and Konstantinos Sagonas. 2018. Automating Targeted Property-Based Testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 70–80. https://doi.org/10.1109/ICST.2018.00017

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

David Maciver and Alastair F. Donaldson. 2020. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer (Tool Insights Paper). In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:27. https://doi.org/10.4230/LIPIcs.ECOOP.2020.13

David R. MacIver. 2016. Hypothesis: Property-Based Testing for Python. https://hypothesis.works/.

Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-Driven QuickChecking of Compilers. *Proc. ACM Program. Lang.* 1, ICFP, Article 15 (aug 2017), 23 pages. https://doi.org/10.1145/3110259

Agustín Mista and Alejandro Russo. 2019. Generating Random Structurally Rich Algebraic Data Type Values. In *Proceedings of the 14th International Workshop on Automation of Software Test* (Montreal, Quebec, Canada) *(AST '19)*. IEEE Press, 48–54. https://doi.org/10.1109/AST.2019.00013

Agustín Mista and Alejandro Russo. 2021. Deriving Compositional Random Generators. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages* (Singapore, Singapore) *(IFL '19)*. Association for Computing Machinery, New York, NY, USA, Article 11, 12 pages. https://doi.org/10.1145/3412932.3412943

Rickard Nilsson. 2019. ScalaCheck: Property-Based Testing for Scala. https://scalacheck.org/.

Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

Michal H. Palka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test* (Waikiki, Honolulu, HI, USA) *(AST '11)*. ACM, New York, NY, USA, 91–97. https://doi.org/10.1145/1982595.1982615

Manolis Papadakis and Konstantinos F. Sagonas. 2011. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang, Tokyo, Japan, September 23, 2011*. 39–50. https://doi.org/10.1145/2034654.2034663

Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. 2022. Computing Correctly with Inductive Relations. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3519939.3523707

Benjamin C. Pierce. 2018. *Software Foundations*. Electronic textbook.

Bogdan Popa. 2021. Rackcheck: Property-Based Testing for Racket. https://docs.racket-lang.org/rackcheck/index.html/.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In *1st ACM SIGPLAN Symposium on Haskell*. ACM, 37–48. https://doi.org/10.1145/1543134.1411292

Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (Jan. 2017), 341–350. https://doi.org/10.1109/TVCG.2016.2599030

Jacob Stanley. 2019. Hedgehog: Release with Confidence. https://hackage.haskell.org/package/hedgehog/.

Jane Street. 2019. Random Testing Framework for OCaml. https://opensource.janestreet.com/base_quickcheck/.

The Cargo Team. 2018. Cargo: Rust's Package Manager. https://doc.rust-lang.org/cargo/.

Kanit Wongsuphasawat, Zening Qu, Dominik Moritz, Riley Chang, Felix Ouk, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2017. Voyager 2: Augmenting Visual Analysis with Partial View Specifications. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, Denver Colorado USA, 2648–2659. https://doi.org/10.1145/3025453.3025768

Li-yao Xia. 2018. A quick tour of generic-random. https://hackage.haskell.org/package/generic-random-1.5.0.0/docs/Generic-Random.html.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. 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*. 283–294. https://doi.org/10.1145/1993498.1993532

Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. 2022. FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3699–3715.