ABSTRACT

Title of Dissertation: DESIGNING EFFECTIVE PROPERTY-BASED TESTING FRAMEWORKS

Alperen Keles
Doctor of Philosophy, 2026

Dissertation Directed by: Associate Professor Leonidas Lampropoulos
Department of Computer Science
University of Maryland, College Park

Property-based testing (PBT) is a widely adopted random testing methodology that combines user-defined logical propositions with automated test case generation. Over the past two decades, PBT has gained significant traction; today, virtually all major programming languages host multiple competing PBT frameworks, each innovating in interface design and underlying techniques. As a result of this proliferation, the PBT landscape of today is significantly fragmented. Advances in one framework often fail to reach others. Common tips, tricks, and techniques are rarely evaluated quantitatively. Historical design choices, once built into popular libraries, can keep the ecosystem from adapting. As a result, the field still lacks a shared empirical foundation for deciding which PBT designs work best, when, and why.

This thesis takes a step toward consolidating PBT knowledge. It contributes a conceptual framework for describing property runners in the wild; a flexible property language for expressing runners distributed throughout the PBT ecosystem; ETNA, the first large-scale eval-

uation and benchmarking platform for PBT; and DIRT, a domain-specific testing approach that clarifies which challenges remain outside the reach of general-purpose consolidation. Together, these contributions make it easier to compare PBT techniques, transfer ideas across frameworks, and build the next generation of testing tools on shared evidence rather than local convention.

DESIGNING EFFECTIVE PROPERTY-BASED TESTING FRAMEWORKS

by

Alperen Keles

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2026

Advisory Committee:
Associate Professor Leonidas Lampropoulos, Chair
Professor Dana Dachman-Soled
Assistant Professor Harrison Goldstein
Assistant Professor Milijana Surbatovich
Associate Professor David Van Horn

To those with big dreams, from small worlds

Acknowledgements

It takes a village, they say, to raise a child. I say it takes a family of villages spread across the Atlantic to get a Ph.D. done. These last 5 years have been a crazy journey with many ups and downs; I have been fortunate to have friends, family and mentors across the globe that pulled me up at every stretch and made it all worth at the end.

Not a word of this thesis would be written today, if Leonidas Lampropoulos wasn't the person he is. His patience against all my attempts to infuriate him with my side projects and quests is the sole reason I have been able to keep making progress towards finishing this thesis. He has been a greater advisor than I could've imagined, or hoped for. Thank you, Leo.

I want to thank Harrison Goldstein for all his mentorship in the last 5 years. His contributions have been integral to all the works I present in this thesis, he has been a wonderful friend and mentor. I also want to thank Benjamin Pierce for influencing the way I think about science, to Mike Hicks for his comments on my proposal, to Milijana Surbatovich and David Van Horn for being on my committee.

PLUM Lab has been a wonderful place to work for the last 5 years. I thank Segev for his puns that livened up the room; Oliwia, Yi, and Caspar for the joy they brought to us in these last 2 years. Justine, Ceren, Ethan, Nikhil have all been a delight to work with. Almost every public presentation I gave started in IRB-5105 as a mild disaster, shaped by the comments of my labmates and professors into something I was proud to present. Thank you all.

It is surprising how many good things start when someone you have never met takes a chance on you. I want to thank Hande Alemdar, who took a chance on me when I wanted told her I wanted to do research with no idea what research was. I want to thank Pelin Angın

for giving me the opportunity to be part of my first research paper. I want to thank Elif Bilge Kavun, for taking a chance on me an introducing me to Marc Fyrbiak, who gave me my first internship that led me to fall in love with formal methods. I want to thank Tudor Dumitras for introducing me to University of Maryland, and Beyazıt Yalçınkaya for kickstarting my whole dream of interning and studying abroad. Many more people have taken chances on me since then, but none would be possible without those first steps.

I want to thank my Turkish friends in the US, who made those quite nights into colorful evenings of joy and celebration. Without them, this country would have been a job, they made it into a life. 3416 Tulane Drive has been my second home for 5 years, sometimes the first, even; I want to thank Utku and Ataberk for the friendship, camaraderie, and brotherhood. I want to thank Oğuz for that first night of hours of talking, making me feel back home for the first time. I want to thank Cemil, for being the rock that he is, always there for all of us; and Ismail, for taking us back to Turkey with his feasts and hospitality. I want to thank Eren for amusing my need for coffee at 2 am, the night rides and the chats along the way. I want to thank Onur for never saying no to another Sapienza, to Arda and Alptuğ for all the dining hall chats. I want to thank Egemen, Bedirhan, Selin, Tayga for being our little Arlington gang.

I want to thank all my friends afar for never feeling away, for not letting the physical distances turn into emotional ones. You have been my past, I hope you will be my future too. To Ozan and Ozan, Yiğit, Göktuğ, Enes, Nisansu, Zeynep, Mehmet, Umut and Umut, Emre, Ecem, Ahmet and Ahmet, Beyza, Hazal, Türker, Alptuğ, Alp, Eren, Elifnur, Ceren, Sude, Nilsu, Asım, Çisil, Murat, Tuğçe, Rabia; thank you for still being here after many years.

I want to thank Müge, the love of my life, for being the greatest supporter of my dreams. She has spent countless hours helping me be the person I am today, I would not be who I am

today without her. She is the compassion, the rock, the anchor I did not now I needed.

Lastly, I want to thank my parents, Hakan and Çiğdem, for raising me the person I am; and my sister Elif, for being there for me at every step. I am the product of their dreams and sacrifices, and I feel so lucky to be a part of their family.

Contents

List of Tables

List of Figures

List of Abbreviations

AFL	American Fuzzy Lop
API	Application Programming Interface
AST	Abstract Syntax Tree
BST	Binary Search Tree
CBC	Correct-by-Construction
CCP	Cumulative Count Plot
CPU	Central Processing Unit
DBAS	Deferred Binding Abstract Syntax
DBMS	Database Management System
DIRT	Database-Integrated Random Testing
DSL	Domain-Specific Language
ETD	Electronic Thesis and Dissertation
GA	Generation Action
GPU	Graphics Processing Unit
HOAS	Higher-Order Abstract Syntax
IFC	Information-Flow Control
LLM	Large Language Model
ML	Machine Learning
N/A	Not Applicable
NoREC	Non-Optimizing Reference Engine Construction
PBT	Property-Based Testing
PQS	Pivoted Query Synthesis
QPG	Query Plan Guidance
RBT	Red-Black Tree
RNG	Random Number Generator
SQL	Structured Query Language
STLC	Simply Typed Lambda Calculus
TED	Tree Edit Distance
TLP	Ternary Logic Partitioning
TSTL	Template Scripting Testing Language
WAL	Write-Ahead Log

Chapter 1
Introduction

Software is written with intent: a programmer wants a computer to do something. Software engineering is the process of refining and reformulating that intent into a form the computer can execute. Software testing is the bridge between intent and the produced program. It is the process of trying inputs, measuring behavior, and understanding what the program does separately from how it is implemented.

Different methods of software testing ask the programmer to express intent in different forms. In the simplest scenario, programmers *tinker* with the program at hand. They interact with the produced artifact, pressing buttons in a user interface, exploring different options in a command-line interface, testing a variety of inputs for a function through a REPL, and observing the results. These interactions are then *codified*, written as concrete testing instructions to be run alongside program development as part of an automated test suite. The simplest form of codification is *example-based testing*, in which the specification relates concrete user-written input-output pairs. A sorting function is tested through several assertions such as `sort([]) == []`, `sort([1, 0]) == [0, 1]`, `sort(3,2,-1) == [-1, 2, 3]`.

Another form of specification, the one that will be the focus of this thesis, is *properties*. A property is a universally quantified predicate function: a fact about the program under test that must hold for any possible input. Since properties are universal, they let users specify whole classes of behavior instead of individual examples. Once a property is written as a specification of a particular program behavior, there are different strategies for producing inputs: systematic exhaustive search, random sampling, and perhaps even user-supplied inputs. The

combination of these methods is typically gathered under the umbrella of *property-based testing*.

Property-based testing (PBT) was pioneered by Koen Claessen and John Hughes in their ICFP 2000 paper "QuickCheck: a lightweight tool for random testing of Haskell programs" [1]. The early history of random software testing is typically traced to the CS736 class at the University of Wisconsin–Madison, where Bart Miller coined the term "fuzz" testing by fuzzing UNIX utilities with random inputs to trigger faults and find bugs [2]. While fuzz testing has evolved toward increasingly feedback-guided, target-aware, and domain-specific input generation techniques [3–8], property-based testing has chosen specifications as its main focus. The original paper introduces both a language for expressing properties and a library implementation for writing random data generators. This design has been replicated in virtually all programming languages in the last 26 years. Table 2.1 gives a glimpse of QuickCheck's reach in the random testing world by presenting the community-maintained `jmid/pbt-frameworks` survey [9].

The ubiquity of the approach stems from its simplicity. The user starts with a simple universal predicate denoting the correctness of their program:

```
sort_correct :: Ord a => [a] -> Bool
sort_correct l = is_sorted (sort l) && is_permutation l (sort l)
```

The `sort_correct` predicate denotes the correctness of a sorting function, the elements of the list must not change, and the output must obey an ordering relation. The user then writes a random data generator for the input type. Since `sort_correct` is defined over arbitrary lists of orderable elements, we need a `List` generator. For this example, we can instantiate the

orderable element type with `Int`s. QuickCheck provides a library of combinators for writing generators such as `choose` and `listOf`, and the `Arbitrary` typeclass for denoting generators of certain types.

```
instance Arbitrary Int where
    arbitrary = choose (0, 100)

-- listOf is a combinator that takes a generator for a
-- type and produces a generator for lists of that type.
instance Arbitrary a => Arbitrary [a] where
    arbitrary = listOf arbitrary
```

Once the universally quantified predicate and the corresponding random data generators are in place, we can use QuickCheck's `forall` combinator for combining them into properties to be tested.

```
prop_sort_correct :: Property
prop_sort_correct = forAll arbitrary $ \xs -> sort_correct xs

quickCheck prop_sort_correct
{-
Stdout:
+++ OK, passed 100 tests.
-}
```

What happens in the background after the `quickCheck` call? The simplest form of a property-based testing runner has two stages. First, a `generator` produces values and a `checker` tests them, yielding one of three results: `PASSED`, indicating that the predicate did not reveal a bug on this input; `DISCARDED`, indicating that the input is invalid for the property; or `FAILED`, indicating that the input violates the property. Once a failing input is found, the runner starts a second stage in which the bug is minimized through shrinking, a form of delta-

debugging [10]. A `shrinker` produces candidate smaller values, for example by reducing keys in a tree or removing part of a computation. If a candidate reproduces the failure, shrinking continues recursively from that candidate until no smaller failing candidate remains, at which point the minimal value is `printed`. An abstract form of this process is depicted in Figure 1.1:

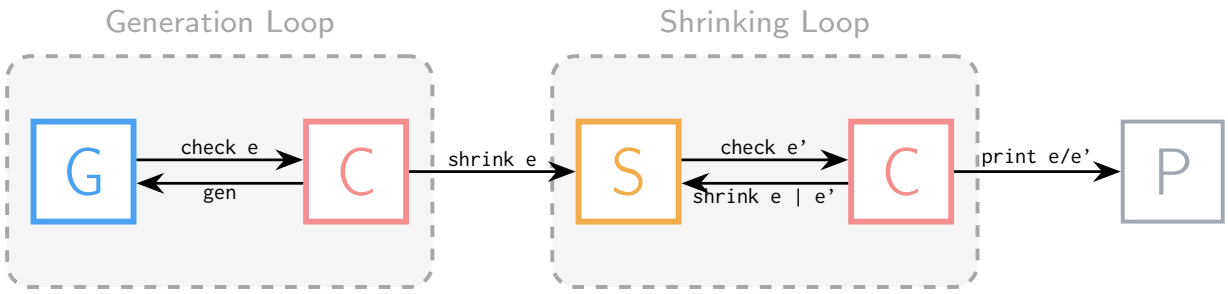


Figure 1.1: An abstract representation of QuickCheck's property runner.

The implementations presented in the PBT frameworks table (Table 2.1) all implement different versions of this same core algorithm, but vary the internal parameters of generators and shrinkers, the APIs for expressing properties, and the underlying source of randomness. Hypothesis [11], Zest [12], and Crowbar [13] parse randomness [14] into structures in their respective programming languages. Many other frameworks use QuickCheck-style RNG splitting [15], starting from an initial random seed and splitting it at every decision point.

Many of these decisions are defaults that can be overridden when users need custom configurations for their domains. Others *leak* outside the abstraction and may require significant reengineering of library internals to accommodate alternative behaviors. For instance, the decision to use randomness parsing versus RNG splitting affects one of the columns in the framework table (Table 2.1), *integrated shrinking*. In integrated shrinking [16], in contrast to the classic structural shrinking approach we mentioned, the source of randomness that is

parsed to the generated structure is shrunk instead of the structure itself. As such, a library using RNG splitting cannot accommodate integrated shrinking without reengineering of its internals.

This thesis argues that effective PBT frameworks require consolidation along two axes: programmable interfaces that make techniques easier to express, combine, and adopt; and shared evaluation infrastructure that grounds design choices in bug-finding capability and quantitative measurement.

Chapter 2 on Programmable Property-Based Testing [17] focuses on this exact problem and its many variants that cause fragmentation of the PBT landscape. Table 2.1 presents 50 frameworks in 31 languages, with up to 5 libraries in OCaml. We trace a precise design decision, the design of the property language in QuickCheck, as the predominant source of inflexibility of a PBT framework. The shallow embedded domain-specific language forces the implementation of the property runner to reside in the "library-space" as opposed to "user-space", leading to significant ecosystem fragmentation. We present an alternative property-language, a mixed-embedding, that enables flexible user-space property runners, allowing us to combine many features and advances dispersed throughout the ecosystem under a single library that users can keep on extending.

Programmable PBT addresses a central challenge: consolidating algorithmic advances within a single framework without reengineering internals, forking an existing library, or writing a new library from scratch. It lowers the cost of experimentation and helps new techniques diffuse through the ecosystem. However, fragmentation is not only a matter of competing frameworks within a language; it is also a problem of information dissemination. How can framework authors adopt a novel approach without concrete empirical evidence about its

effects in their context? A brief look at PBT papers published in the last decade reveals a core issue: quantitative evaluation of PBT is dispersed across bespoke, purpose-specific benchmarks built for individual papers and rarely maintained afterward.

In Chapter 3, I introduce ETNA [18], the first large-scale evaluation platform for PBT. We designed ETNA to be easily extensible in terms of adding new testing strategies for evaluation, new workloads as novel benchmarks, and new languages with entirely separate ecosystems from the existing ones. The purpose of ETNA is to consolidate knowledge through a common interface for quantitative evaluation: shared benchmarks across languages for comparing cross-language performance trade-offs, sampling-based evaluation modes that abstract away framework specifics and focus on generated-test quality, and reproducible experiments for sharing, verifying, and replicating existing results. We have used ETNA to carry out experiments evaluating different generators, shrinkers, and performance differences across frameworks and languages as we explain in § 3.4.2 and § 3.4.3. Similar to Programmable PBT, ETNA is a step towards accelerating scientific development of PBT.

Much of the contemporary discussion of features, generalization, performance, interface, and usage of property-based testing is based on general purpose frameworks we have outlined in Table 2.1. However, the world is full of oddities, special-purpose tools, and requirements that push outside the boxes we draw. General purpose tools grow as they redraw the lines of their approach to accommodate such extensions. In a thesis focusing on consolidation, it is important to examine what we cannot, at least for the moment, consolidate. In Chapter 4, I present and discuss a domain-specific property-based testing paradigm for databases, "DIRT: Database-Integrated Random Testing" [19]. Modern databases have enjoyed the results of fuzzing, at times with property-based testing flavored approaches, in

strengthening their reliability and correctness. However, a major challenge that is yet to be solved is the ability to evolve a property-based testing suite alongside the database, which we attempted to solve. The chapter discusses various other domain-specific tools and techniques as well as their potential inclusion to general purpose PBT frameworks.

The thesis concludes by returning to the central question running through these chapters: how can PBT move from a collection of isolated framework traditions toward a shared engineering and research discipline? The answer I develop is necessarily partial. Flexible property languages, common evaluation infrastructure, and domain-specific testing case studies do not settle every design question, but they make those questions explicit and measurable.

Chapter

2 Programmable Property-Based Testing

In the first chapter, I introduced a simple API for writing property-based tests: define a predicate, write a generator, and combine them into a `Property` to be tested by the framework. This chapter shows why that familiar interface is too rigid for many extensions that have appeared in the PBT literature. The first part of the chapter develops a conceptual framework for property runners, and explains why this framework is itself a contribution: it lets us compare runner designs that are usually buried inside unrelated libraries. The second part presents a flexible property language that makes those runners programmable. This language comes from our paper with the same title as the chapter, "Programmable Property-Based Testing" [17]. This is the result of joint work with Justine Frank, who is the main designer and implementer of the Racket Programmable PBT library; Ceren Mert, who has worked on the reimplementation of the existing ETNA workloads in Rocq and Haskell for evaluating our approach; Harry Goldstein and Leo Lampropoulos, who have been integral to understanding the problem at hand, the design of the interface as well as exposition of the ideas we presented. I have worked on the core design of the DBAS encoding, both libraries in Rocq and Racket as well as all of the experimental setup of the paper. ETNA appears in this chapter as existing evaluation infrastructure and as a source of case-study workloads. Chapter 3 presents ETNA itself in detail, including its design goals and evaluation methodology.

2.1 How to write properties?

The property I presented in the introduction was deliberately simple: single input, no preconditions, no need for any configuration changes or augmentations. From here on, I will use binary-search tree properties from How to Specify It! [20], which we will later revisit as case-study workloads. I will also use Rocq for the exposition because it is the language of our implementation. Rocq hosts QuickChick [21], one of the most faithful ports of QuickCheck, preserving much of QuickCheck's original machinery, types, and algorithms.

```
Definition insert_correct (t: Tree) (k: nat) (k': nat) (v: nat) : option bool :=
    isBST t -=>
      find k' (insert k v t) = if k =? k' then Some v else find k' t.
```

The property `insert_correct` states that for a binary-search tree, inserting an element should, if the keys are equal (which should happen rarely but sometimes), let us find the newly inserted element (which checks if any immediate insertion can fail); if the keys are not equal, insertion should not change the value associated with any other key. This property fails with a number of different bugs breaking insertion in different ways, such as inserting multiple copies of the same key with different values, essentially invalidating any update operations. For now, we will focus on how to test it rather than what those tests would reveal.

Let us initially consider how we would write the property to be tested by QuickChick:

```
Definition prop_insert_correct :=
  forAll arbitrary (fun (t: Tree) =>
  forAll arbitrary (fun (k: nat)  =>
  forAll arbitrary (fun (k': nat) =>
  forAll arbitrary (fun (v: nat)  =>
  insert_correct t k k' v)))).
```

There is a small caveat to using the `Arbitrary` typeclass: each type can have at most one typeclass instance, so we cannot specialize it for every testing scenario. We can replace `arbitrary` in the property definition with specialized generators, and can even define the `shrinkers` and `printers` ourselves using `forAllShrinkShow`:

```
Definition prop_insert_correct :=
  forAllShrinkShow gen shr pri (fun (t: Tree) =>
  forAll (choose -100 100)     (fun (k: nat)  =>
  forAll (choose -100 100)     (fun (k': nat) =>
  forAll (choose -100 100)     (fun (v: nat)  =>
  insert_correct t k k' v)))).
```

This is better, but ideally we would not generate the `k`, `k'`, and `v` if the tree `t` does not pass the precondition `isBST`. The current version still generates everything before starting to check. We could solve this by turning to `forAllShrinkShowMaybe` (which is starting to look like `AbstractPropertyFactory` at this point):

```
Definition prop_insert_correct :=
  forAllShrinkShowMaybe (
    fun _ =>
      t <- arbitrary;
      if isBST t
        then return (Some t)
        else return None
  ) shr pri (fun (t: Tree) =>
  forAll (choose -100 100)   (fun (k: nat)  =>
  forAll (choose -100 100)   (fun (k': nat) =>
  forAll (choose -100 100)   (fun (v: nat)  =>
    find k' (insert k v t) = if k =? k' then Some v
                             else find k' t)))).
```

We can make a few more changes, such as adding callbacks to `collect` statistics from the tests, and QuickChick exposes configuration options for some other aspects of testing.

Beyond those options, however, this is the flexibility the library provides. The more pressing question is, *is this enough flexibility?*

The fragmentation of the PBT landscape suggests that it is not. Looking at Table 2.1, we see significant fragmentation within languages: Haskell hosts QuickCheck, SmallCheck [22], Hedgehog [23] as well as LeanCheck [24], and falsify [25] that is not mentioned in the table. These new frameworks emerge as a response to the limits of flexibility in the existing ones. These limitations arise when user requirements go beyond the design space anticipated by library authors. The dominant design paradigm of PBT property languages fundamentally *fixes* the underlying property runner, albeit with a wide range of configurations for the user. All possible behaviors and configuration options must be programmed into the library internals with no ability to change or extend them in the user space. In the next section, I will introduce a variety of property runners from the literature that diverge from the original property runner of QuickCheck illustrated in Figure 1.1. These runners demonstrate the motivation for a programmable PBT framework.

2.2 Property Runners in Literature

The property-based testing literature contains many proposed changes to the underlying property runner, often motivated by performance or input-quality concerns. In this section, I survey a wide variety of property runners, highlighting their similarities and differences. I depict the runners using abstract representations of the dataflow between different *components*, such as `generators`, `shrinkers`, or `printers`. Crucially, existing implementations of the runners I discuss are spread across different libraries and languages. This section presents

Table 2.1: PBT framework landscape table adapted from `jmid/pbt-frameworks` [9].

Framework	Language	Gen. EDSL	Shrinking	Int. Shr.	Cov. Guidance
theft	C	(Y)	Y	(Y)	(Y)
DeepState	C / C++	(Y)	Y	Y	Y
RapidCheck	C++	Y	Y	-	-
CsCheck	C# / .Net	Y	Y	Y	-
test.check	Clojure	Y	Y	Y	-
check-it	Common Lisp	Y	Y	-	-
cl-quickcheck	Common Lisp	Y	-	-	-
QuickChick	Coq / Rocq	Y	Y	-	Y
PropCheck	Elixir	Y	Y	Y	-
StreamData	Elixir	Y	Y	Y	-
elm test	Elm	Y	Y	Y	-
propcheck	Emacs Lisp	Y	Y	Y	-
QuickCheck	Erlang	Y	Y	Y	-
PropEr	Erlang	Y	Y	Y	-
FsCheck	F# / .Net	Y	Y	-	-
FSharp-Hedgehog	F# / .Net	Y	Y	Y	-
gopter	Go	Y	Y	-	-
Rapid	Go	Y	Y	Y	-
QuickCheck	Haskell	Y	Y	-	-
SmallCheck	Haskell	Y	-[1]	-[1]	-
Hedgehog	Haskell	Y	Y	Y	-
Americium	Java / Scala	Y	Y	Y	-
junit-quickcheck	Java	Y	Y	-	(Y)[2]
QuickTheories	Java	Y	Y	Y	Y
jqwik	Java	Y	Y	Y	-

Framework	Language	Gen. EDSL	Shrinking	Int. Shr.	Cov. Guidance
fast-check	JS / TS	Y	Y	Y	-
propCheck	Kotlin	Y	Y	Y	-
Lua-QuickCheck	Lua	Y	Y	Y	-
Fox	Obj.C / Swift	Y	Y	Y	-
QCheck	OCaml	Y	Y	Y[4]	-
Crowbar	OCaml	Y	(Y)[5]	-	Y
Monolith	OCaml	Y	(Y)[5]	-	Y
Base_quickcheck	OCaml	Y	Y	-	-
Popper	OCaml	Y	Y	Y	-
quickcheck	Prolog	Y	Y	-	-
Hypothesis	Python	Y	Y	Y	(Y)
TSTL	Python	-[6]	Y	Y	Y
R-Hedgehog	R	Y	Y	Y	-
racket-quickcheck	Racket	Y	-	-	-
PropCheck	Ruby	Y	Y	Y	-
PBT	Ruby	Y	Y	-	-
Rantly	Ruby	Y	Y	-	-
quickcheck	Rust	Y	Y	-	-
proptest	Rust	Y	Y	Y	-
Scala-Hedgehog	Scala / JVM	Y	Y	Y	-
ScalaCheck	Scala / JVM	Y	Y	-	-
scalaprops	Scala / JVM	Y	Y	-	-
Echidna	Solidity / EVM	-	Y	Y	Y
qcheck	Standard ML	Y	Y	-	-
SwiftCheck	Swift	Y	Y	-	-

Column notes.

Gen. EDSL = support for a generator DSL;

Shrinking = automatic reduction of failing inputs;

Int. Shr. = shrinking integrated with generation;

Cov. Guidance = coverage-guided exploration.

Source notes.

[1] Contrary to QuickCheck and descendants, SmallCheck's generators enumerate values starting from the smallest ones (up to some bound). SmallCheck therefore encounters a minimal counterexample first and does not require shrinking.

[2] JQF and its underlying Zest engine support multiple forms of feedback guidance beyond code coverage.

[4] QCheck supports integrated shrinking in generators from the QCheck2 module; the original QCheck module does not.

[5] Crowbar and Monolith both use AFL, which trims each test input as part of its core genetic algorithm. In addition, they support test-case reduction via `afl-tmin`, which is unaware of (and may break) OCaml typing.

[6] TSTL uses an external DSL.

our attempt to put them within a single conceptual framework.

2.2.1 Simple Generational Property Runner

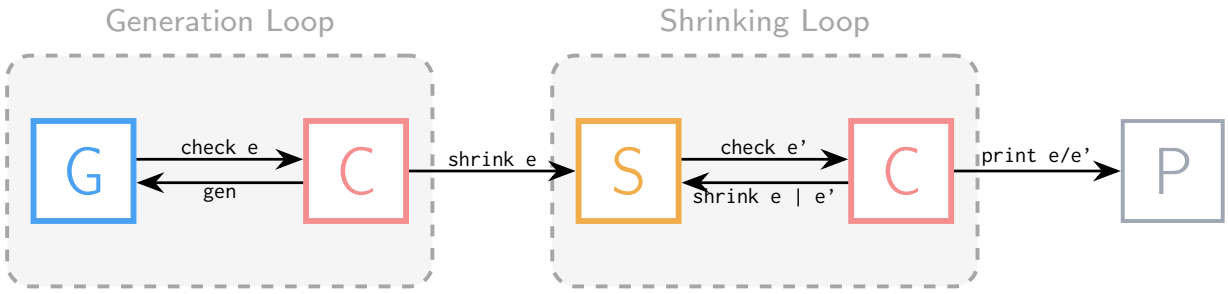


Figure 2.1: An abstract representation of QuickCheck's property runner.

I briefly discussed the quintessential property runner of Figure 2.1 in the introduction as proposed by QuickCheck [1]. Looking a bit more closely, the runner consists of two stages: the first is a generate-and-check stage that repeatedly generates random inputs and tests them against the stated property until a maximum limit on the number of tests is reached, or until an input falsifying the property—i.e., a bug—is found. If such a bug is found, the second stage of the runner is a shrink-and-check stage that repeatedly tries smaller inputs (produced by a user-provided shrinking function) until a local minimum—an input that can no longer be shrunk—is reached.

In principle, such a runner could be implemented using two straightforward modular loops that could be composed together in sequence. However, QuickCheck's design has the shrinking behavior built into the way properties are executed: when a test input is generated, a tree of potentially smaller counterexamples is lazily generated along the way, and is used for minimization. As such, changing the shrinking behavior of QuickCheck, for example to introduce integrated shrinking as we discuss next, necessitates deep changes to the property

representation and the property runner, and as a consequence often a new framework.

2.2.2 Integrated Shrinking Property Runner

Integrated shrinking emerges as a solution to a prevailing problem of type-based shrinking: generators are usually tuned to only produce inputs that satisfy implicit or explicit validity constraints, and type-based shrinking most of the time will not take such constraints into account, leading to minimized but invalid counterexamples. A well-known instance of this issue manifested in the line of work on CSmith and C-Reduce [26,27], where C programs were generated as to not exhibit undefined behavior, but shrinking based on C program grammar alone is likely to introduce undefined behavior during minimization, producing uninteresting counterexamples unless the shrinking process is guarded by external undefined-behavior detection tools.

Integrated shrinking solves this problem by removing the shrinker from the equation altogether, instead reusing the generators themselves to carry out the minimization. Generation can be thought of as a process where random bytes are parsed into structured test cases [14]; in integrated shrinking, rather than minimizing the structured output of generators, frameworks minimize the randomness that goes into them. As a result, any validity constraints that are built into a generator are preserved by construction during shrinking.

This method of integrated shrinking, depicted in Figure 2.2, can be found in many frameworks such as Python's Hypothesis [16] and Haskell's Hedgehog [23] and Falsify [28].

Traditional and integrated shrinking offer an interesting trade-off. Integrated shrinking removes the burden of writing shrinkers (and especially ones that preserve any input invariants) altogether, while QuickCheck's traditional approach can often lead to substantially

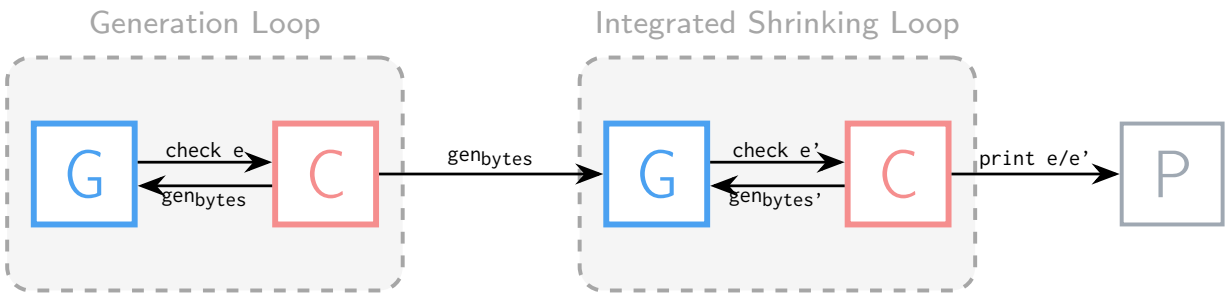


Figure 2.2: An abstract representation of the integrated shrinking property runner.

smaller inputs, and it can be substantially faster in performance (§ 3.4.3). In other words, different situations call for different approaches. Unfortunately, baking in the way inputs are minimized to the way properties are represented and executed means that to take advantage of a different approach users might have to switch frameworks, or even languages entirely.

2.2.3 Coverage-Guided (Fuzzing) Property Runner

Coverage-guided fuzzing, popularized by AFL [4], is an effective random testing technique that has been independent of the PBT literature until recently. The effectiveness of fuzzing has led to multiple attempts to incorporate it into property-based testing: from early ones like OCaml's Crowbar [13], Rocq's FuzzChick [29], and Java's JQF [30], to more recent ones like HypoFuzz [31]. Crucially, each such attempt was, by necessity, either a completely new framework (Crowbar, JQF), or a major redesign of an existing one (QuickChick for FuzzChick, Hypothesis for HypoFuzz) that required significant changes to internals in order to even offer an alternative to its existing testing strategy.

The coverage-guided fuzzing property runner significantly differs from the two loops I introduced, as shown in Figure 2.3. First, coverage-guidance requires coverage-instrumentation, so the checker must be instrumented, allowing the runner to keep track of the branches that

were covered during execution. This information is used for guiding the testing process, which brings us to the second major difference, `mutations`. Historically, property-based testing has overwhelmingly focused on generating data from scratch instead of AFL-style byte mutations. This emphasis favors correct-by-construction generators over mutators that gradually search their way toward valid inputs. However, coverage guidance requires a bias that naive generation from scratch cannot easily provide. Coverage-guided fuzzing supplies that bias by choosing inputs to mutate based on their contributions to overall coverage.

The runner keeps track of a *seed pool*: a corpus of inputs that led execution down interesting, previously unseen paths. Inputs are then selected from this pool, mutated randomly, until a bug (or, more commonly simply a crash) is found.

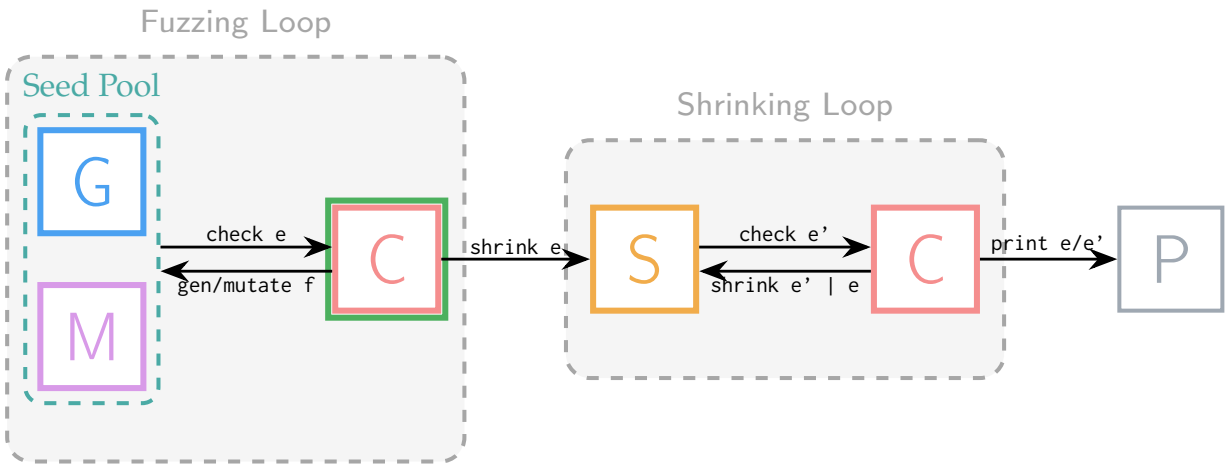


Figure 2.3: An abstract representation of the FuzzChick property runner.

There is an additional artifact of the rigidity of the existing frameworks in changing the property runner, the inflexibility makes it difficult to experiment with new ideas. For example, the search strategy of a fuzzing loop (how seeds are selected for mutation) has seen significant research interest [5–7]. However, when adapting property-based testing to incorporate fuzzing, FuzzChick reused AFL's choices [29] instead of experimenting with new

options in the higher-level setting; in part, because each option in such an experimentation would effectively require a new framework.

2.2.4 Custom-Feedback Guided (Targeted) Property Runner

There are other forms of feedback than coverage information that can be used to guide input generation. The literature is rife with domain-specific metrics that can be used to produce interesting inputs: in SlowFuzz [32] and PerfFuzz [33], the goal of the testing campaign is to discover algorithmic complexity vulnerabilities that could lead to Denial of Service attacks in systems, which is achieved by maximizing instruction counts and execution path lengths; in SQLancer [34], the goal is to find bugs in SQL engines, achieved by maximizing the diversity of the generated query plans [35]. It is also possible to consolidate such different feedback forms under a single compositional design, as demonstrated by FuzzFactory [8] and Target [36], which allowed users to define and combine different feedback functions for guiding input generation. Figure 2.4 depicts an abstract representation of this targeted property runner, where the feedback is a separate function that is computed over the input instead of a value obtained via instrumentation.

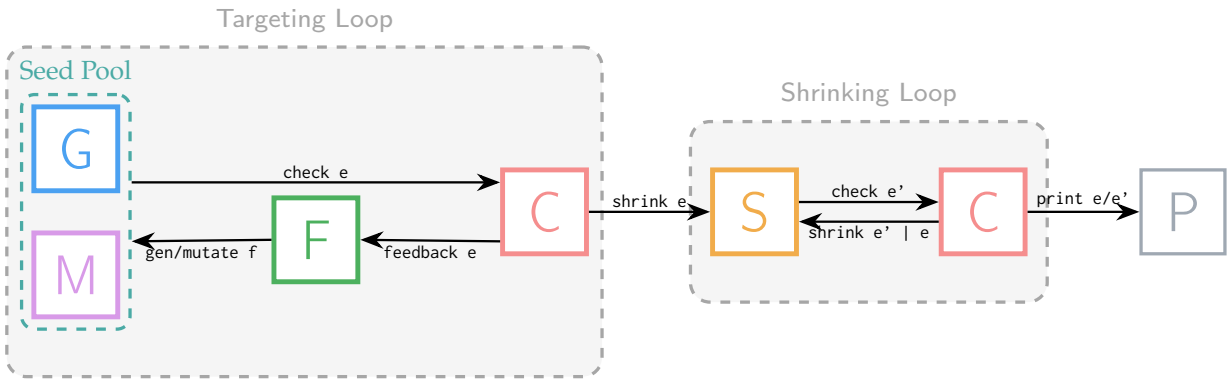


Figure 2.4: An abstract representation of the targeted property runner.

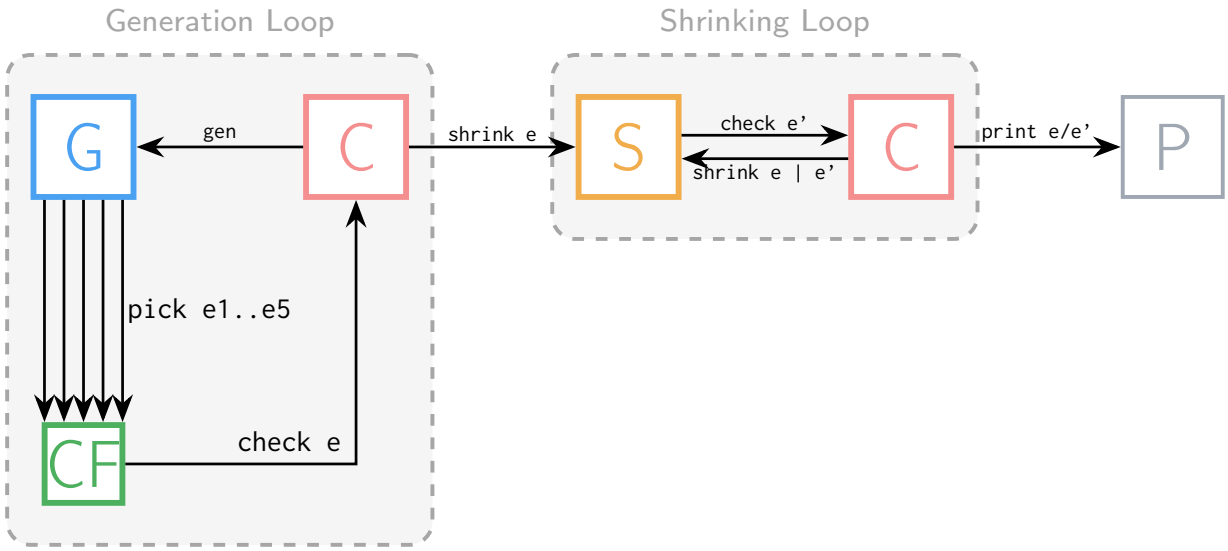


Figure 2.5: An abstract representation of the combinatorial property runner.

2.2.5 Combinatorial Property Runner

There are yet different forms of feedback that can be useful in testing, which do not rely on dynamic feedback obtained during execution of a test, but rather on static features of generated inputs. Combinatorial coverage offers one such alternative [37,38] by performing *online generator thinning*, generating several inputs at each iteration of the generate-and-check loop and only using the input that covers the maximum amount of constructor interactions. Such an approach, depicted in Figure 2.5, is especially useful when executing a test can be costly (e.g. when testing compilers), and does not rely on mutation.

2.2.6 Parallel Property Runner

The other runners explored so far have focused on obtaining a better distribution of inputs by guiding the generator towards interesting parts of the search space. By contrast, QuickerCheck [39] improves the standard loop by exploiting parallelism: letting users run

more tests during the same time span. In prior work, parallelization is made possible by building a "swarm" [40] of testing campaigns, each member focusing on a different subset of operations or capabilities of the underlying system. QuickerCheck instead builds a parallel runner where each thread cooperatively works within the same campaign. Running independent QuickCheck campaigns naively can cause each worker to repeatedly explore the same small-sized part of the input space. QuickerCheck avoids this by sharing a growing size counter across workers. In such a parallel runner, depicted in Figure 2.6, different threads share a common atomic size counter for ensuring parallel progress on each thread while keeping contention on shared resources to a minimum. The paper presents near-linear speed-up for up to 6 threads due to the ability of almost perfect responsibility sharing across threads without contention.

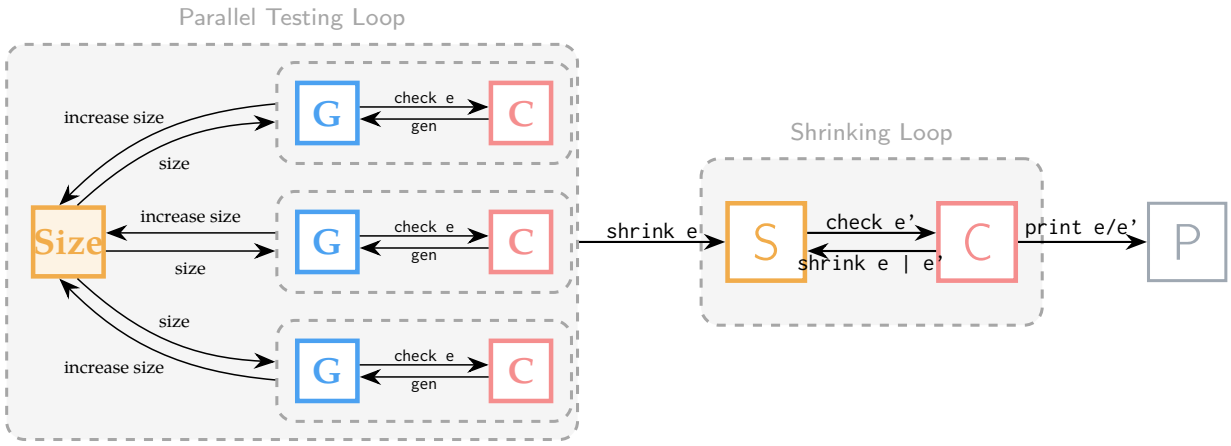


Figure 2.6: An abstract representation of the QuickerCheck property runner.

2.2.7 Stateful Property Runner

The shift from pure random generation in the Simple Generational Property Runner (§ 2.2.1) to the guided mutational generation in Fuzzing (§ 2.2.3) and Targeted (§ 2.2.4) indi-

cates a subtle social change in the usage of PBT. This change is best understood in the context of automation in PBT libraries. In the introduction, I showed the `Arbitrary` typeclass along with the definition of `prop_sort_correct` using the `forAll` combinator QuickCheck provides. In practice, QuickCheck is designed so users *do not have to* write any of those components by hand. This is possible via the `Testable` typeclass. The top-level `quickCheck` function we saw is defined as:

```
quickCheck :: Testable prop => prop -> IO ()
quickCheck p = quickCheckWith stdArgs p
```

Something is `Testable` if it can be turned into a `Property`:

```
class Testable prop where
  -- Convert the thing to a property.
  property :: prop -> Property
```

The `Testable` instance is then defined for any simple type that is expected to be the result of a property predicate:

```
-- This instance is here to make it easier to turn IO () into a Property.
instance Testable () where
  property = property . liftUnit
    where
      -- N.B. the unit gets forced only inside 'property',
      -- so that we turn exceptions into test failures
      liftUnit () = succeeded

instance Testable prop => Testable (Maybe prop) where
  property = property . liftMaybe
    where
      -- See comment for liftUnit above
      liftMaybe Nothing = property Discard
      liftMaybe (Just prop) = property prop
```

```haskell
instance Testable Bool where
  property = property . liftBool

instance Testable Result where
  property = MkProperty . return . MkProp . protectResults . return
```

These result types alone would not allow us to turn any function into a `Property`:

```haskell
instance (Arbitrary a, Show a, Testable prop) => Testable (a -> prop) where
  property f =
    propertyForAllShrinkShow arbitrary shrink (return . show) f
  propertyForAllShrinkShow gen shr shw f =
    -- gen :: Gen b, shr :: b -> [b], f :: b -> a -> prop
    -- Idea: Generate and shrink (b, a) as a pair
    propertyForAllShrinkShow
      (liftM2 (,) gen arbitrary)
      (liftShrink2 shr shrink)
      (\(x, y) -> shw x ++ [show y])
      (uncurry f)
```

Automatic currying allows this typeclass instance to recursively handle functions of any arity as long as the input type can be randomly generated (implements `Arbitrary`) and can be printed (implements `Show`). The automation goes further by allowing users to automatically derive random generators for any Haskell type [41]. The essential intuition is that every variant of a sum type leads to a random choice, and every element of a product type is independently generated and composed into the result.

However, this type-based random generator derivation is hopeless in the face of complex data structures that possess features and invariants that we cannot, or do not, engrain in the types. Binary search trees provide a simple example:

```haskell
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

The core invariant of the tree (all keys in the left sub-tree are smaller than the root, all keys in the right sub-tree are greater than the root) is not expressed in the datatype definition. So, the derived generator would have the following form:

```haskell
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized gen
    where
      gen 0 = pure Leaf
      gen n = frequency
        [ (1, pure Leaf)
        , (3, Node <$> gen (n `div` 2)
                   <*> arbitrary
                   <*> gen (n `div` 2))
        ]
```

The chances of generating valid binary trees using this generator are *extremely low*. Below is the table to view the situation quantitatively:

Table 2.2: Distribution of trees produced by the naive generator above with size budget $n = 64$ and 200,000 samples. The BST hit-rate collapses past two nodes, and the bulk of the population (trees with ≥ 8 nodes) contains no BSTs at all.

Nodes	Sampled	BSTs	% BST	% of Pop.
0	51,159	51,159	100.00	25.58
1	9,762	9,762	100.00	4.88
2	3,670	1,743	47.49	1.84
3	1,366	233	17.06	0.68
4	597	21	3.52	0.30
5	471	7	1.49	0.24
6	212	0	0.00	0.11
7	471	0	0.00	0.24
≥ 8	132,292	0	0.00	66.15

There are four solutions to this problem:

1. **Write correct-by-construction generators:** One can implement a hand-written generator that targets the valid subset of the datatype. However, this shifts the burden from

testing the program to carefully reimplementing much of the datatype's invariant structure inside the generator. As the datatype gains more interacting invariants, the generator must encode more case splits, dependencies, and repair logic, making it easy for the generator itself to become the most complicated part of the test.

2. **Derive correct-by-construction generators [42]:** This is what QuickChick in Rocq does: generators are derived via inductive Rocq specifications. Although this is a promising and powerful direction, it relies on having an inductive specification of the structure, which is a generous assumption.

3. **Mutate into interesting inputs [4]:** Coverage-guided fuzzing reduces the need for a domain-specific generator by searching from seed inputs toward executions that cover new program behavior. For highly structured inputs, however, many mutations are rejected before reaching the semantic checks that distinguish valid from invalid structures. Without a parser, repair step, or domain-specific feedback signal, short fuzzing campaigns can spend most of their budget rediscovering syntactic or invariant violations rather than exploring the valid state space.

4. **Use stateful generation.**

In stateful generation, instead of generating the structure itself, we generate *actions* or *operations* against the structure to gradually change its state. For instance, instead of creating a binary search tree from scratch, we can start with a `Leaf` and repeatedly call `insert` with random key-value pairs. A more canonical example is databases, whose complexity makes generation from scratch impractical. Instead, we define database operators such as `CREATE TABLE`,

`DROP TABLE`, `INSERT`, `UPDATE`, and `DELETE` to populate the database. Historically, stateful testing has been implemented as a separate feature. Conceptually, however, changing generators from `gen a :: Rng -> a` to `gen a :: State -> Rng -> a` is enough to express a stateful property runner. In Figure 2.7, I present the stateful runner in the same style as the previous runners in this section. The key difference is the introduction of a shared `State` that the generator consults when producing each action, and that the checker advances after executing an action against the system under test. Unlike the simple generational runner where each input is generated independently from a fresh seed, here each generation step depends on the accumulated effects of the prior actions, and the shrinking loop operates over action *sequences* rather than a single value.

This style of model-based stateful random testing has a long history in PBT, including PropEr's stateful testing interface [43]. It is also the natural shape of recent database-testing work, where tools build databases through sequences of operations and then check relational or metamorphic oracles over the resulting state [19, 34, 44–47].

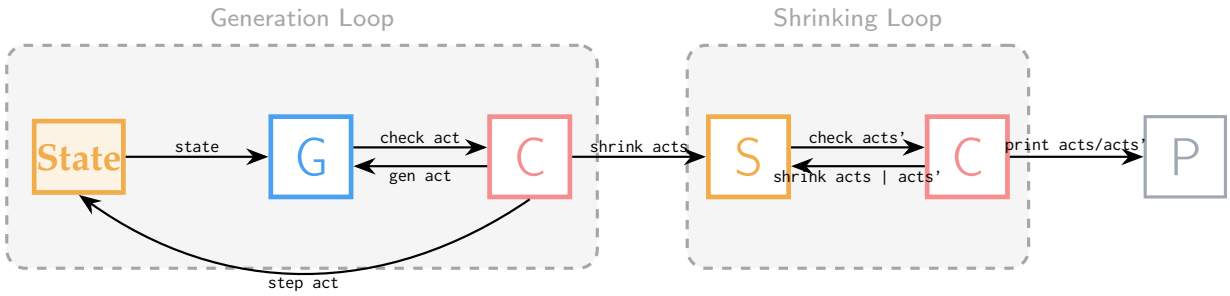


Figure 2.7: An abstract representation of the stateful property runner. The `State` box is read by the generator (`state`) and advanced by the checker after each action (`step act`); the shrinking loop minimises the sequence of actions `acts` that led to a failure rather than a single value.

2.2.8 Discussion

The property runners discussed in this section have given rise to a wide variety of frameworks across languages. Unfortunately, while their abstract representations appear similar, their implementations are anything but: a tight coupling between property runners and property DSLs means that implementation details often render it very difficult, if not entirely impossible, to switch and experiment with different approaches without major changes to the underlying code.

2.3 Property Languages

Properties are, at a fundamental level, *universally quantified predicates*. In order to test properties, we need to model them in the programming languages we use. Simply, we want to define a layer of such properties on top of some standard host language, equipped with at least booleans:

$$\begin{array}{lcl} \tau & := & \texttt{Bool} \mid \tau \to \tau \\ e & := & x \mid \lambda x.e \mid e\ e \mid T \mid F \mid \ldots \end{array}$$

In practice, the host language needs to also support some kind of randomness for generation, lists for shrinking, strings for pretty printing, etc. For presentation purposes, we begin by formally describing the core boolean structure:

$$p \quad := \quad \forall x : \tau.\ p \mid e$$

That is, properties are either universal quantifiers or an injection of a host predicate.

The core question of our ICFP paper, "Programmable Property-Based Testing", is, *how should we represent this language?* Each representation yields different trade-offs in terms of ex-

pressibility and ergonomics, and our exploration of the design space has led us to a mixed embedding that yields a significantly more flexible PBT library compared to the state of the art. In this section, I present a review of existing property-languages from the literature, followed by an introduction of *Deferred Binding Abstract Syntax* (*DBAS*), our proposed alternative.

2.3.1 Background: Deep and Shallow Embeddings

An *embedding* involves two languages: an *object language* that is being implemented, and a *host language* that the object language is being implemented in. In a deep embedding, the terms of the object language are represented as inductive data in the host language. These representations allow arbitrarily manipulating and inspecting the terms of the object language via pattern matching. By contrast, shallow embeddings directly expand the constructs of the object language in terms of constructs of the host language, which allows straightforward implementation but no user-level term manipulation [48]. Naturally, multiple hybrid alternatives have been explored, trading off between the simplicity of shallow embeddings and the flexibility of deep ones [49–51].

2.3.2 A Shallow Property Language

Defining a property language requires a construct for defining its inputs via universal quantification (∀), a construct for defining the boolean predicate validated through the property (*check*), and finally a mechanism for running the property. In PBT libraries using a shallow embedding following QuickCheck's design, the mechanism is using host-language lambdas as binders for universal quantification.

$$
\begin{array}{lcl}
\mathit{forall} & : & \forall\tau.\,(\tau \to \mathit{prop}) \to \mathit{prop} \\
\mathit{check} & : & \texttt{Bool} \to \mathit{prop} \\
\mathit{run} & : & \mathit{prop} \to \texttt{IO}\ () \\
\end{array}
$$

Concretely, the type of properties is some type *prop* in the host language (often restricted via a typeclass-like mechanism), *check* injects a host boolean into this type, and *forall* takes a host-level function and turns it into a property. Finally, frameworks also provide a *run* function to test properties constructed using this API.

The main advantage of this approach is that it is very convenient for users to write properties (using host-level binders), and easy to test them (simply invoking the framework-provided *run*). Implementing such a framework as a developer is also straightforward. A minimal implementation of this shallow API in Haskell using splittable pseudorandomness [15] is shown in Figure 2.8: the type of generators for some type `a` is simply a wrapper around a function from some source of randomness to `a`; properties are generators for `Results` (a datatype that encodes at least whether testing succeeded); `forall` takes a generator and a predicate and binds them together taking care of the underlying randomness; and the `run` loop simply generates and checks a result up to some predefined limit on the number of tests, taking care again of the underlying randomness, and reporting success or failure in the end.

In practice, implementations of such frameworks are more involved to account for better reporting, minimizing counterexamples, etc. However, a key characteristic of this approach is that the `Property` type is opaque to users: it is simply a wrapper around a function. As a result, users cannot inspect its structure or customize how properties are tested without modifying the definition of `Property` and the internals of the framework.

```haskell
newtype Gen a = Gen (StdGen -> a)

type Property = Gen Result
data Result = OK | Failed

forall :: Gen a -> (a -> Property) -> Property
forall (Gen g) f = Gen (\r -> let (r1,r2) = split r
                                  Gen h = f (g r1)
                              in h r2

run :: Property -> IO ()
run (Gen g) = do newStdGen >>= loop numTests
  where loop 0 _ = ... -- report success
        loop n r = let (r1, r2) = split r in case g r1 of
          OK     -> loop (n-1) r2
          Failed -> ... -- report counterexample
```

Figure 2.8: An implementation of QuickCheck's core functionality.

2.3.3 A Deep Property Language

If we imagine shallowness and deepness as a spectrum, the opposite end from the shallow embedding paradigm is a deep embedding in which terms of the host language are expressible in the object language, allowing them to be manipulated directly. In a compiled setting, the boundary between the host and object languages is strict, leaving staged compilation as the main implementation route. In interpreted settings, it is possible, though cumbersome, to manipulate host-language terms without explicit staging. To my knowledge, there is only one tool experimenting with the deep approach, TSTL [52]. TSTL is "the Template Scripting Testing Language" that allows for writing templates in an EBNF-like syntax that can include arbitrary Python expressions denoting a generative grammar. Below is a TSTL template for testing an AVL tree in Python:

```
@import avl

pool: <int> 2
pool: <avl> 1

<int> := <[0..10]>
<avl> := avl.AVLTree()

<avl>.insert(<int>)
property: <avl>.check_balanced()
```

The advantage of such an embedding is that any consumer of this template can theoretically write another interpretation for the same property; its flexibility is substantial. However, it also means that the users of the testing framework have to learn *yet another* language in which to write their specifications and generators, foregoing all of the convenience that host-language binders provide. Unsurprisingly, almost no frameworks have taken this route: it is hard enough to convince users to write properties and specifications without additional burdens to adoption [53].

2.3.4 A Mixed Property Language

I remember one of my undergraduate professors teaching Computer Networks class saying that the one thing computer scientists love more than anything is to find hybrids. We take two ends of a spectrum and find a middle ground that can work better than both. A pure shallow embedding results in a rigid, inflexible testing library that people cannot configure for their requirements. A pure deep embedding on the other hand, forgoes ergonomics in exchange for *perceived flexibility*, however, the libraries we create must be ergonomic enough for people to use, otherwise the potential of flexibility is never realized. *Mixed embeddings* emerge as optimal points in the spectrum where parts of the object language is reified as

deeply embedded data structures while the rest is kept as shallowly embedded combinators. Perhaps the most famous of such embeddings is *Higher-Order Abstract Syntax* (*HOAS*) [54], a popular approach to defining languages where the variable bindings in lambdas reuse host language binders instead of redefining substitution at the object level.

```
data Exp
  = Var String
  | Lam String Type Exp
  | App Exp Exp
```

```
data Exp
  = App Exp Exp
  | Lam Type (Exp -> Exp)
```

Figure 2.9: STLC encodings with and without HOAS. Non-HOAS (on left) stores binders explicitly; HOAS (on right) reuses host-language binders.

In defining our property-language, we could attempt a HOAS-style approach, where we define an inductive type of properties, with a constructor corresponding to *forall* and *check*, both indexed by the arguments of their shallow counterparts:

$$\begin{array}{l} \mathit{Inductive\ Prop} := \\ \mid \mathit{Forall} : (\tau \to \mathit{Prop}) \to \mathit{Prop} \\ \mid \mathit{Check} : \texttt{Bool} \to \mathit{Prop} \end{array}$$

Such a representation still allows us to encode universal quantifiers using host-language binders, and also makes headway into the issue at hand: given a property, we can now pattern match against it. However, there is still a problem: to access the "rest" of the property that follows a universal quantification, we need an element of the type being quantified over; since such elements are randomly generated, we cannot actually recurse down the structure of the property without restricting ourselves to something along the lines of QuickCheck's `Gen`erator monad.

The key issue is that we are hiding the definition of the "rest" of the property under a

host-language binder: the *Forall* constructor takes a function as an argument, which we cannot pattern match on to go deeper. Could we use standard host-language constructs to define the predicates at the leaves of the property (the *Check*s), while retaining the ability to pattern match on property structure?

2.3.4.1 Proposal: Deferred Binding Abstract Syntax

Our solution is what we call *deferred binding abstract syntax* (DBAS): rather than binding a variable once at the site of its universal quantification, we instead bind it at every one of its use sites.

$$\begin{array}{l} \mathit{Inductive\ Prop}\ (\mathit{env} : [\mathit{Type}]) := \\ \mid \mathit{Forall} : \forall \tau.\ \mathit{Prop}\ (\tau :: \mathit{env}) \rightarrow \mathit{Prop\ env} \\ \mid \mathit{Check} : (\llbracket \mathit{env} \rrbracket \rightarrow \texttt{Bool}) \rightarrow \mathit{Prop\ env} \end{array}$$

We still define an inductive type of properties, with one constructor for each combinator in our API. We also index our type of properties by an environment: a list of types that have already been quantified. The key change is that we move the host-level binder from the binding site (*Forall*) to its use site (the *Check*). That is, the argument to *Forall* is no longer a function, but simply a property with an extended environment; on the other hand, the argument to *Check* is no longer a simple boolean, but a function that binds *everything* in the environment. In the code above, we denote that as $\llbracket \mathit{env} \rrbracket$.

At first glance, this is a counter-intuitive trade-off: every time you want to use a variable, you have to bind *everything* quantified before that point. That only makes sense in a scenario where there are many quantifications and few variable uses—which is precisely the case for the language of properties. Compared to the previous language representations, this DBAS-based one allows us to access the structure of the "rest" of the predicate without having access

to a concrete value, which is necessary when such values are to be randomly generated.

Adding Annotations

Generalizing the property language above to include annotations for generation, shrinking, or printing of individual elements is straightforward. We simply include an optional extensible list of annotations at the *Forall* constructor:

$$
\begin{array}{l}
\mathit{Inductive\ Prop}\ (\mathit{env} : [\mathit{Type}]) := \\
\mid \mathit{Forall} : \forall \tau.\ \mathit{as} \rightarrow\ \mathit{Prop}\ (\tau :: \mathit{env}) \rightarrow \mathit{Prop\ env} \\
\mid \mathit{Check} : (\llbracket \mathit{env} \rrbracket \rightarrow \texttt{Bool}) \rightarrow \mathit{Prop\ env}
\end{array}
$$

$$
\begin{array}{l}
\mathit{as} := \emptyset \mid (k, \forall \tau.\ \llbracket \mathit{env} \rrbracket \rightarrow \tau) :: \mathit{as} \\
k\ := \mathit{gen} \mid \mathit{shr} \mid \ldots
\end{array}
$$

At a high level, we can annotate each *Forall* constructor with a (possibly empty) sequence of host-level functions that quantify over the context so far (as in *Check*) and return annotation-specific terms (e.g. a generator or a shrinker).

2.4 A Dynamically-Typed Property Language

There are multiple mechanisms for implementing a DBAS embedding. In this section, I describe an implementation of DBAS in Racket. Dynamically typed runtimes allow programmers to create data structures that are difficult or impossible to express within static type-checking boundaries. Sometimes the type system is not expressive enough to model the underlying object; other times, the objects being created are intentionally not well typed. The trade-off is that a dynamically typed setting makes the programmer responsible for maintaining well-typedness, rather than relying on static guarantees.

Specifically, the `env` in the definition of the inductive `Prop` which has the type `[Type]`, a list of types, is tricky to implement and manage in static type systems. Each layer of the property

is parameterized by a context of the variables introduced in the upper layers, and therefore has access to them. In a dynamically-typed setting, we can solve this by simply hijacking the variable bindings of the host language.

In its final form, a property looks almost identical to a property written in RackCheck [55], the existing combinator-based PBT library for Racket:

```
(define eval-opt
  (property
    (forall e #:contract expr? #:gen gen-expr)
    (equal? (eval e) (eval (optimize e)))))
```

In RackCheck's property language, `property` would evaluate the `forall`, produce the variable `e`, and then evaluate the following predicate with `e` in scope. We instead use the Racket macro system to turn each generated variable into an environment access.

```
(equal? (eval e) (eval (optimize e)))
```

is transformed into

```
(equal? (eval (get-env "e")) (eval (optimize (get-env "e"))))
```

The evaluator receives the environment and enriches it using the `augments` on universally quantified variable introductions (`forall`). The public interface is simple:

```
(struct Forall (var augments body))
(struct Implies (prop body))
(struct Check (prop))
```

Each `Forall` carries an augment dictionary. Augments map keywords like `#:contract`,

`#:gen`, and `#:shrink` to functions from the current environment to annotation-specific values. Without syntax support, this can be quite verbose:

```
(define eval-opt
  (Forall 'e (hash '#:contract (lambda (env) expr?)
                   '#:gen      (lambda (env) gen-expr))
          (Check (lambda (env)
                   (let ([e (dict-ref env 'e)])
                     (equal? (eval e) (eval (optimize e))))))))
```

The ergonomics here, compared to the flat property definition at the beginning, are drastically different. The nestedness of the constructors aside, we must explicitly bind the expression `e` to the environment reference (via `(dict-ref env 'e)`) and enclose every expression with a lambda that has access to the environment. We can fortunately rely on the extensive macro capabilities of Racket in bringing ergonomics to our language.

Concretely, `property` is implemented as a recursive `syntax-parse` macro with three cases: one for `forall`, one for `implies`, and a base case for `check`-style leaves. The `forall` clause expands into a `Forall` node and introduces a local identifier transformer for the bound variable using Racket's variable-like transformer API. Inside the remainder of the property, any use of that variable is rewritten to an environment lookup (`(dict-ref env 'x)`), rather than to a host-language binder.

To make this rewriting compositional, the macro uses a syntax parameter (`property-env`) that is rebound at each generated lambda (`(lambda (env) ...)`). This gives both predicate bodies and annotation expressions (`#:gen`, `#:contract`, `#:shrink`, ...) access to the current DBAS environment.

Writing Property Runners

The premise of our new property language is that it gives rise to flexible PBT libraries that support user-space property runners based on the conceptual architectures introduced in § 2.2. Because a property is stored as an inspectable data structure, users can in principle define many different interpretations. In practice, however, the library should expose convenient abstractions for doing so. Our abstraction layer over properties is *components*. Recall the simple generational property runner from the introduction:

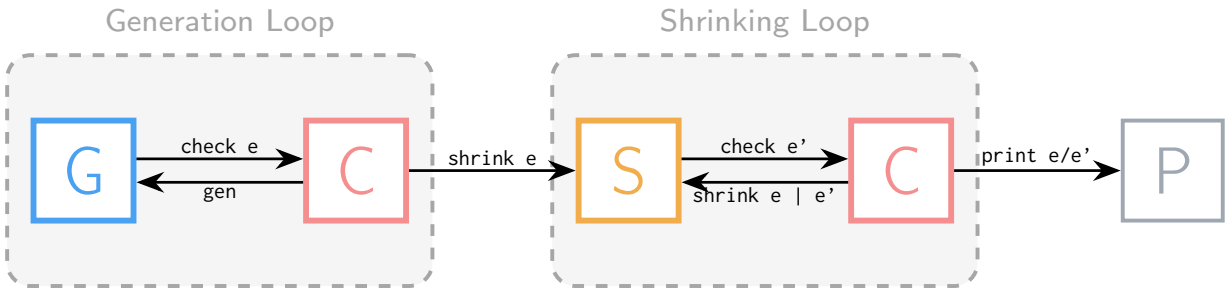


Figure 2.10: An abstract representation of QuickCheck's property runner.

The loop relies on four components, a `generator`, a `checker`, a `shrinker`, and a `printer`. If we can construct all of these components out of a given property object, we can then use them to build the property runner. The `generator` starts with an empty map of inputs, fills it as it interprets the property, and returns the environment enriched with the randomly generated inputs at the end. Each `gen` arrow in the diagram depicts a call to the generator.

```
(define (generator p size)
  (let loop ([p p]
             [env (hash)])
    (match p
      [(Forall var augments body)
       (unless (dict-has-key? augments '#:gen)
         (error 'no-generator))
       (define val
```

```
      (run-rackcheck-generator
       ((dict-ref augments '#:gen) env)
       rng
       size))
     (when (dict-has-key? augments '#:contract)
       (invariant-assertion ((dict-ref augments '#:contract) env) val))
     (loop body (dict-set env var val))]
    [(Implies _ body)
     (loop body env)]
    [(Check _) env])))
```

The `checker` takes the property and an environment of generated variables, and produces a result. The `check e` arrows in the diagram indicate calls to the checker.

```
(define (checker p env)
  (match p
    [(Forall var augments body)
     (unless (dict-has-key? env var)
       (error 'missing-variable))
     (when (dict-has-key? augments '#:contract)
       (invariant-assertion ((dict-ref augments '#:contract) env) (dict-ref env var)))
     (checker body env)]
    [(Implies prop body)
     (if (prop env) (checker body env) 'discard)]
    [(Check prop)
     (if (prop env) 'pass 'fail)]))
```

The shrinker is more involved because the `shrinker` does not produce a single shrunk value; it produces a list of candidate smaller values. For a list, a shrinker might remove an element from the beginning or end. For a tree, it might remove the left or right subtree. Instead of spelling out these possibilities one by one, a `shrink` call produces a list of possible minimizations. The shrink loop then goes through the list, `checking` each possibility, and when one candidate reproduces the failure, recursively applies the shrinking process to that candidate.

```
(define (shrink-loop prop env)
  (define (shrink-var prop env x shrinker)
    (let down ([v (dict-ref env x)])
      (let across ([shrinks (shrinker v)])
        (match shrinks
          ['() v]
          [(cons v vs)
           (if (equal? (check-property prop (dict-set env x v)) 'fail)
               (down v)
               (across vs))]))))
  (match prop
    [(Forall x augments body)
     (define contract ((dict-ref augments '#:contract (const any/c)) env))
     (if (dict-has-key? augments '#:shrink)
         (let* ([shrinker (invariant-assertion
                           (-> contract (listof contract))
                           ((dict-ref augments '#:shrink) env))]
                [v (shrink-var body env x shrinker)])
           (shrink-loop body (dict-set env x v)))
         (shrink-loop body env))]
    [(Implies _ body) (shrink-loop body env)]
    [(Check _) env]))
```

A convenient feature of dynamically typed settings is the uniformity of values: we can pass any values we are interested in throughout the program. In the Racket implementation, this means we do not need a separate `printer` component. We can now build user-space property runners with these components:

```
(define (simple-generational-property-runner tests p)
  (let loop ([n 0] [passed 0] [discards 0])
    (if (= n tests)
        (result #f passed discards #f)
        (let ([env (generator p (floor (log n 2)))])
          (case (checker p env)
            [(fail) (result #t passed discards (shrink-loop p env))]
            [(pass) (loop (add1 n) (add1 passed) discards)]
            [(discard) (loop (add1 n) passed (add1 discards))])))))
```

We can now turn to our implementation of DBAS in Rocq: a dependently typed data

structure that holds properties and provides the same expressivity in a statically checked, type-safe setting.

2.5 A Dependently Typed Property Language

As in the Racket implementation, we rely on existing infrastructure: QuickChick's generation machinery in Rocq [56]. This lets us turn existing QuickChick property-based tests into programmable form with minimal effort.

In our implementation, we explicitly encode contexts in our properties, capturing every input that *will have been generated* by that point. We use a standard inductive definition of contexts, where `Emp` is the empty context and `Ext` extends a context by one type. Given a context, we calculate the host-level tuple type corresponding to that context.

```
Inductive Ctx :=
| Emp : Ctx
| Ext : Type -> Ctx -> Ctx.
```

```
Fixpoint interp (C : Ctx) : Type :=
  match C with
  | Emp => unit
  | Ext T C' => T * interp C'
  end.
```

Now we can define a deeper property language with DBAS:

```
Inductive Prop : Ctx -> Type :=
| FORALL : forall {A : Type} {C : Ctx} (name : string)
      (generator : interp C -> G A)
      (mutator   : interp C -> A -> G A)
      (shrinker  : interp C -> A -> list A)
```

```
      (printer   : interp C -> A -> string),
      Prop (Ext A C) -> Prop C
| IMPLIES : forall C,
      (interp C -> bool) -> Prop C -> Prop C
| CHECK : forall C,
      (interp C -> bool) -> Prop C.
```

Like shallow QuickCheck-style APIs, this representation still lets users write generators, mutators, shrinkers, and printers in the host language and still supports typeclass-based defaults. The key difference is that users can now pattern match on properties and define new interpreters without modifying framework internals.

For example, the standard generate-and-check loop can be written as a structural interpreter over this property syntax:

```
Inductive RunResult {C : Ctx} (prop : Prop C) :=
| Normal : bool -> RunResult prop
| Discard : RunResult prop.
```

```
Fixpoint genAndRun (C : Ctx) (prop : Prop C) : interp C -> G (RunResult prop) :=
  match prop with
  | FORALL _ gen _ _ _ p =>
      fun env =>
        a <- gen env;;
        genAndRun _ p (a, env)
  | IMPLIES _ pre p =>
      fun env =>
        if pre env then genAndRun _ p env else ret Discard
  | CHECK _ p =>
      fun env =>
        ret (if p env then Normal true else Normal false)
  end.
```

The flexibility to define such loops at user level allows encoding generators, runners, shrinkers, fuzzers, and hybrid loops in a single framework while keeping execution and re-

porting programmable.

Usable Defaults with Dependently Typed Programming

A common downside of deeper embeddings is ergonomics. If exposed directly, users would write substantial boilerplate:

```
Definition prop_eval_bad :=
  FORALL (fun _ => gen) (fun _ => mut)
         (fun _ => shrink) (fun _ => pretty)
         (@CHECK (Ext Expr Emp) (fun '(e, _) => eval e == eval (optimize e))).
```

In practice, we expose a surface language with typeclass-driven defaults:

```
Definition prop_eval :=
  ForAll e :- Expr,
  Check (fun '(e, _) => eval e == eval (optimize e)).
```

Users can still override individual components:

```
Definition prop_eval :=
  ForAll e :- Expr gen:gen,
  Check (fun '(e, _) => eval e == eval (optimize e)).
```

The final piece for convenient `Check` definitions uses a typeclass that reconstructs contexts from tupled predicates:

```
Class Untuple (A : Type) :=
  { untuple : Ctx
  ; untuple_correct : interp untuple = A }.


Instance Untuple_empty : Untuple unit :=
  { untuple := Emp
```

```
  ; untuple_correct := eq_refl }.

Instance Untuple_pair {A B} `{Untuple B} : Untuple (A * B) :=
  { untuple := Ext A (@untuple B _)
  ; untuple_correct := _ }.



Definition Check {A} `{Untuple A} (p : A -> bool) : Prop (@untuple A _).
  refine (CHECK (@untuple A _) _).
  rewrite untuple_correct.
  exact p.
Defined.
```

Finally, QuickChick metaprogramming can derive deep properties directly from Rocq predicates for a no-effort starting point:

```
Definition eval_correct (e : Expr) := eval e == eval (optimize e).
Derive Property eval_correct.
(* ==> eval_correct_prop is defined. *)
```

2.6 Evaluation

In this section I evaluate our approach and implementation:

1. The first set of experiments demonstrates the expressiveness of DBAS, showing that it supports new property runners without requiring modifications to the library internals: we implemented all of the property runners presented in § 2.2. Here, I discuss four such runners and their implementations in detail: the standard QuickCheck-inspired property runner (§ 2.2.1), the mutation-based coverage-guided runner (§ 2.2.3), the mutation-based custom feedback-guided runner (§ 2.2.4), and a parallel property runner (§ 2.2.6) inspired by QuickerCheck [57].

2. In the second experiment, we evaluate the performance overhead of DBAS-based implementations, showing that it is comparable to their shallowly embedded counterparts. In § 2.6.2 we compare the performance of the DBAS-based property runner as implemented in Rocq and Racket to the existing property runners of QuickChick and RackCheck. We found that using the DBAS-based embedding has *no observable performance overhead*.

3. Finally, the added expressivity allows these experiments to be expressed as changes to reusable runners and runner components, rather than as changes to the property language or framework internals. We carry out three new experiments: in § 2.6.3.1 we explore how different design choices with respect to the representation and sampling of the seed pool affect testing performance, improving FuzzChick [29] in the process; in § 2.6.4, we compare the default integrated shrinking capabilities of RackCheck with a simple external shrinker we implemented for a DBAS-style Racket library; in § 2.6.5, we implement and benchmark a parallel property runner inspired by recent work on parallelizing QuickCheck [39].

2.6.1 Property Runners with DBAS

We demonstrate the expressiveness of DBAS by implementing all property runners from § 2.2. In the previous two sections we saw how to implement a simple `genAndRun` loop in both Rocq and Racket. Leveraging the ability to pattern match on properties and recurse down their structure, we can similarly develop building blocks that will help implement all property runners. In particular, for this section, we use a generator, which produces random inputs

for a property; a mutator, which given inputs to a property mutates some or all of them; a runner, which given inputs to a property executes it; a shrinker, which given a way to minimize a property's inputs and a counterexample finds a locally minimal counterexample using best-first search; and a printer, for reporting counterexamples. The underlying pattern matching ability is still useful in the presence of these abstractions. For example, we will also write domain-specific versions of these functions for some of our evaluation, such as an `instrumentedRunner` that runs a property collecting instrumentation feedback in the process.

2.6.1.1 Simple Generational Property Runner

Using the components outlined above we can implement any variation of the standard generational runner, with code that closely follows its depiction. Figure 2.11 shows the implementation of the basic generate-and-test then shrink-and-test property runner. Figure 2.12 shows the same runner in Racket, with minor syntactical adjustments.

Each component of the property runner in Figure 1.1 has a clear correspondence to the corresponding overlaid sections in Figure 2.11. The functions `gen`, `run`, `shrinker`, and `print` are the building blocks of user-level property runners, but can also be written directly by users, as shown in Sections 2.5 and 2.4. We use them here to present runners at a higher level of abstraction as enabled by DBAS, focusing on how these components interact with each other in order to show how users can implement new runners that fit their needs.

The runner itself consists of two tight loops, the first one running `gen`, `run`, `gen`, `run`... until a counterexample is found, or until a predefined limit of tests has been reached. The second loop runs `shrink`, `run`, `shrink`, `run`... until it is not able to minimize the input further, reporting the smallest input within the shrinking process.

```
Definition runLoop (fuel : nat) (cprop : Prop Emp) :=
  let fix runLoop' (fuel : nat) (cprop : Prop Emp)
      (passed : nat) (discards : nat) : G Result :=
    match fuel with
    | O => ret (mkResult discards false passed [])
    | S fuel' =>
        input <- gen cprop (log2 (passed + discards));;
        res <- run cprop input;;
        match res with
        | Normal seed false =>
            let shrunk := shrinker 10 cprop seed in
            let printed := print cprop 0 shrunk in
            ret (mkResult discards true (passed + 1) printed)
        | Normal _ true =>
            runLoop' fuel' cprop (passed + 1) discards
        | Discard _ _ =>
            runLoop' fuel' cprop passed (discards + 1)
        end
    end in
  runLoop' fuel cprop 0 0.
```

Figure 2.11: Simple Generational Property Runner in Rocq

As users now have access to the building blocks for writing this loop, they are empowered to change it. For instance, we have experimented with implementing a precondition-bypassing `run` function that can optimize property execution for correct-by-construction generators by skipping preconditions. Users can also change the shrinking algorithm, adjust sizes, move to structured printing, write results directly to a file, decide not to terminate after a single counterexample, or add specific budgets for discards and time bounds. These choices, and many more that users may invent, can be used to augment the effectiveness of testing.

```
(define (run-loop tests p)
  (let loop ([n 0] [passed 0] [discards 0])
    (if (= n tests)
        (result #f passed discards #f)
        (let ([env (generate p run-rackcheck-gen (floor (log n 2)))])
          (case (check-property p env)
            [(fail) (result #t passed discards (shrink-eager p env))]
            [(pass) (loop (add1 n) (add1 passed) discards)]
            [(discard) (loop (add1 n) passed (add1 discards))])))))
```

Figure 2.12: Simple Generational Property Runner in Racket

2.6.1.2 Mutation-Based Property Runners

Property-based testing has long relied on random generation from scratch as opposed to mutating existing inputs as fuzzing tools do. This design has been challenged in Targeted PBT [36,58] with a mutational approach that optimizes explicit user-provided feedback functions, and later by FuzzChick [29] and Zest [12], which rely on branch coverage information obtained via binary instrumentation, followed by libraries such as HypoFuzz [31] and Bolero [59].

An important design decision we must make when implementing a DBAS property encoding in a statically typed language, such as our Rocq encoding, is which associated functions are part of the property encoding. These recent advances in mutational PBT have led us to make mutation a first-class citizen of our library, and we have implemented a generic seed pool interface that mutation-based property runners can leverage. The seed pool interface abstracts away search strategy (how to select which input to mutate) and power schedule (how long to fuzz it for) concerns. I will revisit this abstraction in just a few sections (§ 2.6.3.1). We implemented two property runners using these components: a coverage-guided fuzzing

runner and a custom feedback-guided (targeted) runner. The targeted runner is detailed in the next section.

```
Definition fuzzLoop (fuel : nat) (cprop : Prop Emp) {Pool}
    `{pool : SeedPool Pool} (seeds : Pool) : G Result :=
  let fix fuzzLoop' (fuel passed discards : nat) (seeds : Pool) : G Result :=
    match fuel with
    | O => ret (mkResult discards false passed [])
    | S fuel' =>
        let directive := sample seeds in
        input <- match directive with
                 | Generate => gen cprop (log2 (passed + discards))
                 | Mutate source => mutate cprop source
                 end;;
        res <- instrumentedRun cprop withInstrumentation;;
        let '(res, feedback) := res in
        match res with
        | Normal seed false =>
            let shrunk := shrinkLoop 10 cprop seed in
            let printed := print cprop 0 shrunk in
            ret (mkResult discards true (passed + 1) printed)
        | Normal seed true =>
            match useful seeds feedback with
            | true =>
                let seeds' := invest (seed, feedback) seeds in
                fuzzLoop' fuel' (passed + 1) discards seeds'
            | false =>
                let seeds' := match directive with
                              | Generate => seeds
                              | Mutate _ => revise seeds
                              end in
                fuzzLoop' fuel' (passed + 1) discards seeds'
            end
        | Discard _ _ =>
            match directive with
            | Generate => fuzzLoop' fuel' passed (discards + 1) seeds
            | Mutate _ =>
                match useful seeds feedback with
                | true => fuzzLoop' fuel' passed (discards + 1) seeds
                | false =>
```

```
                fuzzLoop' fuel' passed (discards + 1) (revise seeds)
            end
        end
    end
  end in
fuzzLoop' fuel 0 0 seeds.
```

The coverage-guided fuzzing property runner introduces some complexity beyond the simple QuickCheck-style generational runner. This complexity is mainly related to the orchestration logic that manages the seed pool, which is parametric over the `SeedPool` interface. At each iteration of the loop, the seed pool produces a directive, either to generate an input from scratch, or mutate a previous input. The generated input is then passed into `instrumentedRun` function that is also parameterized over a custom instrumentation function.

In classic fuzz testing, this instrumentation function is branch or path coverage, yet our Rocq library can accommodate any function that observes information about the state of the executed program, as in [8]. This is reflected in the fuzzing loop in Figure 2.6.1.2 where feedback is received from the execution of the `instrumentedRun`. Such feedback can range from traditional branch or path coverage (as in coverage-guided fuzzing) to timing or memory usage (as in performance fuzzing [33]).

Thus, we view the fuzzing property runner presented in Figure 2.6.1.2 as (1) a more generalized version of the classic coverage-guided fuzzing, and (2) a property-based testing version of FuzzFactory [8], which allowed for arbitrary instrumentation functions to guide the search similar to the fuzzing property runner I present here.

2.6.1.3 Custom Feedback-Guided (Targeted) Property Runner

In recent years, PBT tools and mutation-based fuzzers have begun to find common ground. On one hand, fuzzing tools have been moving towards more structured input generation and more complex properties than simply "the program does not crash" [60–62]. On the other hand, we have seen a rise in the popularity of property-based testing tools that are able to guide the generation of inputs using feedback [12, 29, 36]. Following this trend, we have used DBAS to implement mutation-based generation, developing two property runners leveraging mutation and feedback.

A mutation-based targeted property runner has two important differences from the simple generational property runner described in Figure 2.11: the feedback and the targeting. It uses a genetic algorithm based on a user provided custom feedback function for guiding the search towards interesting inputs, and it employs a user-provided or type-derived mutator function to mutate the input it currently focuses on. The main benefit of such a runner is that it means developers can potentially avoid writing complex generators for complex types and preconditions.

The runner is further parameterized by a seed pool (to keep track of interesting inputs) and a utility function (to accommodate different types of feedback). As shown in the pioneering work of [8], customizable feedback under user control can lead to very effective testing, and DBAS allows even more relevant choices to be made without needing to modify the internals of a framework. A pictorial depiction of the targeted property runner is illustrated in Figure 2.4.

In Figure 2.6.1.3, I provide an annotated Rocq implementation of the custom-feedback

guided (targeted) property runner in Rocq. Similar to the simple generational runner or coverage-guided (fuzzing) runner in the previous section, this runner reuses the building blocks we have provided for writing novel property runners.

```
Definition targetLoop (fuel : nat) (cprop : Prop Emp)
(feedback_function : Seed— > Z) {Pool : Type}
{pool : SeedPoolPool} (seeds : Pool)
{utility : UtilityPoolZ} : G Result :=
  let fix targetLoop' (fuel passed discards : nat) (seeds : Pool) : G Result :=
    match fuel with
    | O => ret (mkResult discards false passed [])
    | S fuel' =>
        let directive := sample seeds in
        input <- match directive with
                 | Generate => gen cprop (log2 (passed + discards))
                 | Mutate source => mutate cprop source
                 end;;
        res <- run cprop input;;
        match res with
        | Normal seed false =>
            let shrunk := shrinkLoop 10 cprop seed in
            let printed := print cprop 0 shrunk in
            ret (mkResult discards true (passed + 1) printed)
        | Normal seed true =>
            let feedback := feedback_function seed in
            match useful seeds feedback with
            | true =>
                let seeds' := invest (seed, feedback) seeds in
                targetLoop' fuel' (passed + 1) discards seeds'
            | false =>
                let seeds' := match directive with
                              | Generate => seeds
                              | Mutate _ => revise seeds
                              end in
                targetLoop' fuel' (passed + 1) discards seeds'
            end
        | Discard _ _ =>
            targetLoop' fuel' passed (discards + 1) seeds
        end
    end in
```

```
targetLoop' fuel 0 0 seeds.
```

It is important to note that this specific runner is not a fixed part of the library, but merely a default behavior that is convenient and can be readily used. Alternative implementations may change the feedback behavior to accommodate feedback in the discard cases, or change the printing or shrinking behaviors. Even the seed pool and utility typeclasses are provided as sensible defaults and building blocks, rather than static design choices as is the case in shallow embedding based PBT frameworks.

2.6.1.4 Parallel Property Runner

Figure 2.13 shows a slightly abridged version of our Racket parallel testing runner. Parallelism is implemented through Racket futures. We use a lock-free shared counter and a flag that is set if a counterexample is found, which is necessary because Racket futures are not able to be halted arbitrarily. Each worker thread grabs a test number from the counter, and loops until it either finds a counterexample or the counter exceeds the test number, with the main thread waiting for results from the workers and combining them when they finish.

Like all the other loops I presented, this loop is not a fixed part of the library, and more efficient or sophisticated parallel runners can be implemented without changing the underlying property representation.

2.6.2 Comparison of Deep and Shallow Embeddings

In this section, I present our comparisons of the performance of a DBAS-powered implementation to its shallow counterpart. We turn to the ETNA [18,63] framework for evaluating

```
(define (parallel-run-loop tests prop [num-workers (processor-count)])
  (define counter (box 0))
  (define found-counterexample? (box #f))
  (define (worker-thunk)
    (define rng (make-pseudo-random-generator))
    (let worker-loop ([passed 0] [discards 0])
      (define n (box-faa! counter 1))
      (cond
        [(>= n tests) (results #f passed discards #f)]
        [(unbox found-counterexample?) (results #f passed discards #f)]
        [else
         (let-values ([(res env) (gen-and-run prop run-rackcheck-generator rng n)])
           (case res
             [(fail)
              (set-box! found-counterexample? #t)
              (results #t passed discards env)]
             [(pass) (worker-loop (add1 passed) discards)]
             [(discard) (worker-loop passed (add1 discards))]))]))))
  (define workers
    (for/list ([_ (in-range num-workers)])
      (future worker-thunk)))
  (for/fold ([res (results #f 0 0 #f)]) ([worker workers])
    (define worker-res (touch worker))
    (results (or (results-foundbug? res) (results-foundbug? worker-res))
             (+ (results-passed res) (results-passed worker-res))
             (+ (results-discards res) (results-discards worker-res))
             (or (results-counterexample res)
                 (results-counterexample worker-res)))))
```

Figure 2.13: Parallel Property Runner in Racket

property-based testing performance, which Chapter 3 presents in detail. ETNA comes with a series of Rocq workloads—programs along with injected bugs—in the form of Binary Search Trees, Red-Black Trees, and the Simply Typed Lambda Calculus. For the purposes of this experiment we extended the ETNA tool to support Racket and ported these workloads.

Throughout the sequence of case studies that follow, our performance results will use ETNA bucket charts: each bucket represents the tasks (mutant-property pairs) solved within

Figure 2.14: ETNA bucket-chart shading legend.

a certain time limit in the average of 10 trial runs. The leftmost bucket with the darkest color denotes the tasks solved within 0.1 seconds, where the remaining buckets progressively denote the tasks solved within 1, 10, 60 seconds, and the last bucket denotes the tasks that were not solved within 60 seconds for at least one of the 10 trial runs. The legend follows the same ETNA bucket-chart conventions used throughout this chapter.

2.6.2.1 Comparison of Deep and Shallow Embeddings in Rocq

The first case study focuses on the performance implications of using deferred binding abstract syntax instead of the standard QuickCheck-style runner.

We benchmark our implementations of this loop for both Rocq and Racket against the existing loops of the QuickChick [21] and RackCheck [55] libraries in 3 ETNA workloads: binary search trees (BST), red-black trees (RBT), and the simply-typed lambda calculus (STLC). Our results show that libraries implemented via DBAS, using the runner shown earlier in this section, are on par with both QuickChick and RackCheck.

The Rocq bucket-chart results show that using DBAS does not incur a performance penalty compared to QuickChick in the BST, RBT, and STLC workloads. In a total of 12 strategy/workload pairs, DBAS-style Rocq library outperforms QuickChick in 9 of them, while QuickChick has a better performance in 3 of them. Yet, there are no notable differences in the results of the two libraries in terms of mean time to solve the tasks.

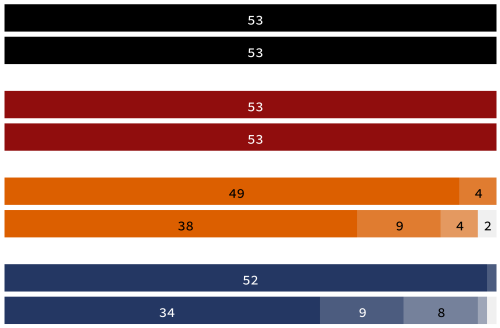


Figure 2.15: Binary Search Trees

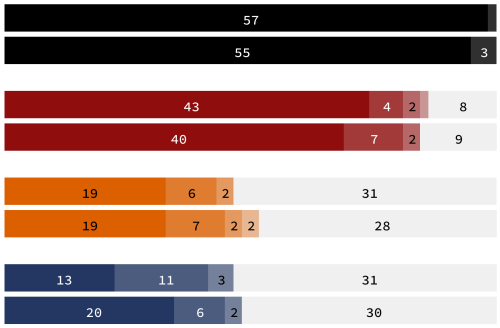


Figure 2.16: Red-Black Trees

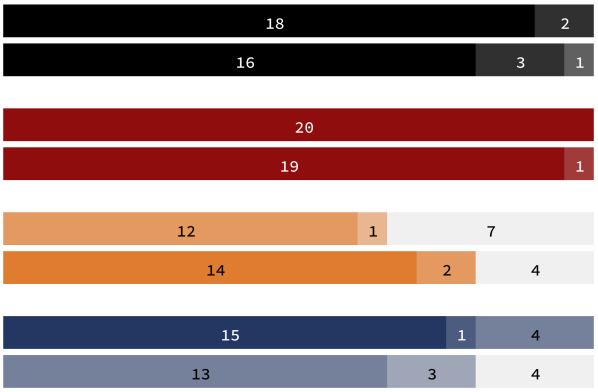


Figure 2.17: Simply-Typed Lambda Calculus

Figure 2.18: Comparison of shallow and deep embeddings in Rocq. Within each strategy color, the top row is the DBAS/deep version and the bottom row is the shallow QuickChick version.
■ = Bespoke generator, ■ = Specification-based generator,
■ = Type-based fuzzer, ■ = Type-based generator.

2.6.2.2 Comparison of Deep and Shallow Embeddings in Racket

In the Racket experiments, we focused on comparing the DBAS-style Racket library with the shallow embedding implemented by RackCheck. We conducted our experiments on the same BST, RBT, and STLC workloads by porting the existing Haskell workloads to Racket. In the process of porting these workloads, we also discovered and reported a bug in the RackCheck core, which the authors have fixed in the latest version of the library.

Our Racket results for deep versus shallow embeddings show that, once again, we find

no notable differences in performance between the two versions. However, during the course of these experiments we found that the default configuration of RackCheck in terms of size of generated terms led to considerable performance degradation in the RBT case study. Since size is configurable in most PBT APIs, we changed the RackCheck size function to be logarithmic with respect to number of tests, as our property runner does. Still, this further reinforces our point on programmability: if sizes and similar aspects of generation are configurable (and severely impact testing performance), why should the runners themselves not be configurable?

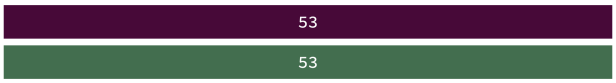


Figure 2.19: BST

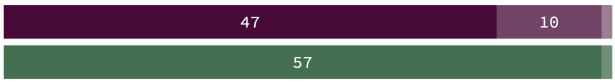


Figure 2.20: RBT

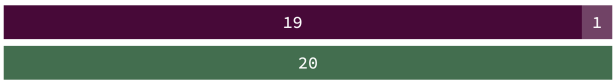


Figure 2.21: STLC

Figure 2.22: Comparison of shallow and deep embeddings in Racket. = DBAS-style Racket, = RackCheck.

2.6.3 DBAS-powered Experiments

In this section, I present three case studies enabled by the expressiveness of DBAS.

2.6.3.1 An Exploration of Seed Pool Design Choices

First, we turn to feedback-guided property runners and their programmability. Fuzzing research often explores different strategies and power schedules [4,5,64,65], with researchers proposing increasingly refined designs. On the other hand, property-based testing frame-

works that support feedback-guided generation of inputs such as FuzzChick or Hypothesis emphasize the ability to test arbitrary user-defined specifications, but do not provide options for configuring such crucial parameters of their feedback-guided property runners. The feedback-guided runner shown earlier in this section abstracts these concerns into a small API, which we implemented in Rocq.

The configurability enabled by DBAS allows users to rely on a set of community-accepted defaults chosen by framework developers, while also exploring whether choices from the literature, or novel choices of their own, fit their testing needs better. In this case study, we explored six different data structure representations to hold interesting seeds, as well as four different power schedules, leading to a total of 21 different configurations. We picked these configurations to explore different parts of the design space, such as the queuing strategy, the size/cardinality of the pool, whether the pool is monotonic, and how many times a given seed is reused. We conducted our experiments on the IFC workload in ETNA, using a type-based generator alongside a type-based mutator. While this experiment reveals a clear winner for this case study, the primary goal is not the exploration itself, but rather to demonstrate that such an exploration can be expressed by varying reusable runner components without modifying the property language or framework internals.

More concretely, we explored the following data structure representations, three that hold collections of seeds (as in FuzzChick), and three that only hold a single seed (as in Targeted PBT):

1. *FIFO Queue Seed Pool*: A pool that holds a queue of seeds, and reduces the energy of the current seed after usage. The pool only generates a new seed from scratch when the

queue is empty, and mutates the seed otherwise. When the energy of the current seed drops to 0, it is removed from the queue. The next seed is chosen from the front of the queue. This was the default behavior of FuzzChick.

2. *FILO Queue Seed Pool*: The same as the *FIFO Queue Seed Pool*, but the next seed is chosen from the back of the queue.

3. *Heap Seed Pool*: Similar to the FIFO and FILO Queues, but the seeds are stored in a heap, creating a priority queue.

4. *Static Singleton Pool*: A pool that holds a single seed, and does not reduce its energy after usage. The pool generates a new seed from scratch at the first iteration, and mutates its seed for the subsequent iterations. The seed is only updated when a new seed with a better feedback is generated via mutation. This essentially devolves the search to hill climbing, as in the original Targeted PBT work [36].

5. *Dynamic Monotonic Singleton Pool*: A pool that holds a single seed, and reduces its energy after usage. The pool generates a new seed from scratch at the first iteration and when the energy of the current seed is 0, mutates the seed otherwise. The seed is only updated when a new seed with a better feedback is generated.

6. *Dynamic Resetting Singleton Pool*: A pool that holds a single seed, and reduces its energy after usage. As opposed to the *Dynamic Monotonic Singleton Pool*, once the current seed's energy is 0, the seed is effectively discarded and a new seed is generated from scratch.

All of the queues except the *Static Singleton Pool* have been tested with 4 different energy schedules, where the energy of a new seed was respectively up to 1, 10, 100, 1000, depending

on its interestingness. We report the experiments over 5 trials for each configuration within 65 tasks in the IFC workload in ETNA. For brevity, the graphs omit 34 tasks none of the configurations have solved, and only show the remaining 31 tasks. Each bucket represents the tasks solved within a certain time limit for at least one of the 5 trials, where the leftmost bucket with the darkest color denotes the tasks solved within 0.1 seconds, where the remaining buckets progressively denote the tasks solved within 1, 10, 60 seconds, and the last bucket denotes the tasks that were not solved within 60 seconds.

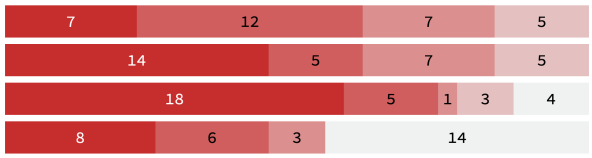


Figure 2.23: Heap Seed Pool

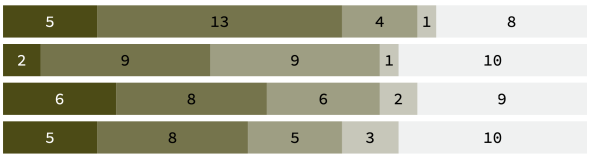


Figure 2.24: FILO Queue Seed Pool

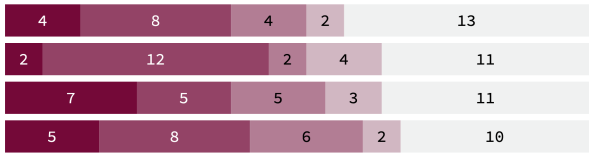


Figure 2.25: FIFO Queue Seed Pool

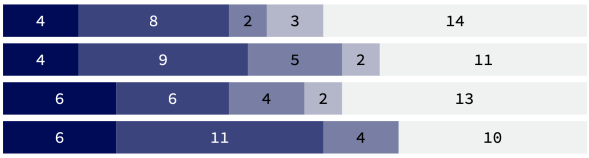


Figure 2.26: Dynamic Monotonic Singleton Seed Pool

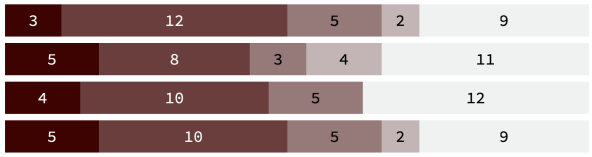


Figure 2.27: Dynamic Resetting Singleton Seed Pool

Figure 2.28: Static Singleton Seed Pool

Figure 2.29: Comparison of seedpool strategies in Rocq. Within each subfigure, rows show maximum seed energies 1, 10, 100, and 1000 from top to bottom; the static singleton pool has only one configuration. Bucket shading follows Figure 2.14.

These results mark a task as "solved" for the purposes of a bucket chart if at least 1 out of 5 fuzzing campaigns finds a bug. If we switched to requiring *all* fuzzing campaigns to find

the bug, the results paint an even more compelling argument: *only* Heap-based pools consistently find counterexamples. I omit the graphs because all other options fail to consistently solve a task across runs, indicating a very high variance that heap seed pool does not exhibit. As a result, we plan to upstream the Heap-based seed pool to QuickChick as the effective configuration as a default going forward.

2.6.4 Comparison of Integrated Shrinking with External Shrinking

Test case reduction, counterexample minimization, or shrinking is a fundamental aspect of random testing [10]; however, it receives much less attention than random generation. A major split in PBT frameworks has been their approach to shrinking, with the original QuickCheck using delta-debugging style structural shrinking where the generated counterexample structure is shrunk by removing its substructures to produce smaller versions with an accompanying reproducer that tests if the smaller versions keep triggering the bug; and Hypothesis [16] in Python, falsify [28] in Haskell, and RackCheck [55] in Racket taking an alternative route. Instead of shrinking the generated structure, these frameworks shrink the source of randomness that led to generation. This method, called integrated or internal shrinking, removes the requirement of a separate shrinking function and preserves the generation invariants in the generator while making no guarantees on the similarity of the shapes obtained in the shrinking phase.

The central point of the study is that such splitting points should not be fixed at the level of the library, but should instead be decisions made in relevant contexts by the users of the library. In an ideal scenario, a user should be able to switch between using an integrated shrinker versus an external one, and vice versa; and perhaps even use or build a new approach

that might not be available when the library was written in the first place. Instead of forcing the users to choose or change libraries based on their context and approach to shrinking, DBAS allows for building your own property runner with its custom approach to shrinking, and that is precisely what we have done in this experiment.

We have compared the effectiveness of the integrated shrinker of RackCheck against a simple external shrinker we have implemented in our DBAS PBT library in Racket. We have conducted our experiment on the `System F` workload in ETNA, and we used the same generator in both RackCheck and DBAS-style Racket library, equipping our library with a simple type-based external shrinker we implemented in place of the integrated shrinker.

The results show that the external shrinker successfully shrinks System F terms to a minimal counterexample that is an average of 2.66 times smaller than the original input with a standard deviation of 1.23, while the integrated shrinker of RackCheck only shrinks the inputs to an average rate of 1.04 times smaller than the original input with a standard deviation of 0.37. RackCheck only shrunk the inputs to smaller inputs in 66 out of the 360 trials, kept the size the same in 77, grew the inputs in 68, and failed to shrink in 149 trials. These results may be surprising at first glance: does the internal shrinker really only shrink the inputs in 20% of the trials? In this case, it does. Its notion of size is based on the structure of the generator rather than on the input itself, so "smaller randomness" may not actually lead to smaller input values.

The point of this experiment is not to dismiss internal shrinking—for many testing situations, it is perfectly sufficient and much more user-friendly than a bespoke shrinker. However, in pathological cases, users need an escape hatch, and DBAS provides the necessary configurability.

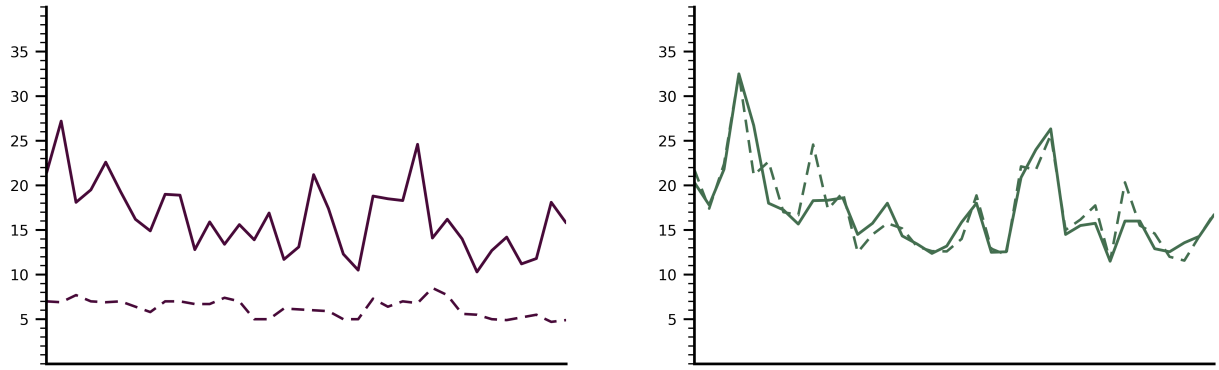


Figure 2.30: Comparison of shrinking in DBAS vs RackCheck.
Original sizes (continuous) and shrunk sizes (dashed) across tasks (x axis) for the external shrinker (left) and internal shrinker (right).

2.6.5 Parallelizing Property-Based Testing

An important recent work on novel property runners is QuickerCheck [39], where authors implement and evaluate a parallel run-time for QuickCheck [1], achieving massive performance gains on a variety of workloads as opposed to prior work leveraging parallelism by producing a swarm of testing configurations [40]. The underlying idea is simple: the authors provide an alternative, parallel property runner `quickCheckPar`, in which multiple worker threads share a common variable that controls the size of the generated inputs. Using this strategy, the authors achieve an almost linear speed-up with respect to the number of physical cores used in testing. Unfortunately, due to the rigid structure of shallow embedding based PBT libraries, creating such a parallel runner requires intensive engineering effort. The commit history of the project shows that the implementation of this new runner is an effort spanning 2 years and more than 50 commits.

In contrast, the DBAS-style Racket library lets us implement a naive version of the QuickerCheck algorithm as a user-space runner, without changing the property representation or

the core framework. Our implementation uses worker threads with 2 shared atomic variables, one keeping the current number of tests across threads, and one indicating if the process of testing is finished or not.

We compared this 4-thread parallel runner against the single-threaded runner across 131 tasks (53 BST, 58 RBT, 20 STLC). The result is mixed, which is expected for these workloads: most ETNA tasks are solved by bespoke Racket generators within 0.1 seconds (§ 2.6.2.2), leaving little room for parallelism to overcome thread-management overhead. Among the 49 tasks where the single-threaded and parallel times differ by more than one millisecond, the parallel runner is slower on average (the average time ratio is 1.2, i.e., 20% more time). On the 10 tasks where the difference exceeds 0.1 seconds, the parallel runner is faster by roughly 3x on average (the average time ratio is 0.33). The only task with a difference above one second takes 3.56 seconds with the single-threaded runner and 1.16 seconds with the parallel runner. These numbers should not be read as evidence that this naive Racket implementation matches QuickerCheck's optimized runtime; rather, they show that DBAS lets a user implement and evaluate a new runner design directly, exposing the expected trade-off between parallel overhead on tiny tasks and speedup on longer-running ones.

2.7 Related Work

Throughout the chapter, I have thoroughly discussed various property-based testing frameworks, their property languages, and the property runners that they come bundled with. Here, I briefly summarize related work in PBT and in deeply embedded domain-specific languages.

Property-Based Testing Frameworks

To our knowledge, no widely-used frameworks expose a reified property representation that supports user-defined runners without modifying library internals. There are multiple frameworks that make distinctly different choices in what capabilities they provide to users as presented in Table 2.1.

Free Generators

The design of our deeply embedded property language builds on a rich literature of embedded DSLs [66]. In particular, our approach parallels work on *free generators* [14], which present a deeply embedded DSL for writing random generators. Deeply embedded properties and generators are largely orthogonal—free generators may be able to simplify the implementation of some of the different runners described in § 2.6, but they do not allow the developer to re-program the loop itself. This follows a more general trend of using free structures to increase expressivity or usability in the programming languages community. For example, itrees [67] introduced a general-purpose Rocq structure which is essentially a coinductive variant of free monads, which allows them to represent and reason about interactive recursive programs. In follow-up work, [68] explored how other free structures (such as applicative functors) can be used as an alternative to free-monad-based embeddings.

Mixed Embeddings

There is also a long line of related work in attempting to bridge the benefits of shallow embeddings (ease of use, as in the current property language) with those of deep ones (extensibility). For example, [69] showed how to use typeclasses in Haskell to allow for shal-

low embeddings that can be interpreted in different ways, hinting at a way of incorporating a deeper property language in a type system such as Haskell's. More recently, [70] showed how to convert between embedding representations by *unembedding* alleviating some of the problems with Higher-Order Abstract Syntax representations. Finally, [49] introduced a hybrid embedding where typing derivations are represented as a deep embedding indexed by shallow terms in the host language, offering pattern matching capabilities.

2.8 Conclusion and Future Work

In this chapter, I have presented Programmable Property-Based Testing, a PBT paradigm that changes the underlying representation of properties using deferred binding abstract syntax (DBAS), a mixed embedding we introduce. Programmable PBT is a new approach to encoding properties for PBT that enables more flexible and programmable testing. The key advance made by DBAS is to reify properties as a free data structure; this allows them to be written in a clear and readable way, separate from the property runner that tests them. These more deeply embedded properties can then be inspected and interpreted by user-defined property runners. With the help of DBAS, developers in Rocq, Racket, and hopefully soon other programming languages, can tailor, experiment, and adapt their testing setup to their domain.

In the future, we intend to make DBAS convenient to program in more mainstream languages than Rocq and Racket. On the static typing side, we chose to implement our approach in Rocq as its dependent type system and powerful notation mechanism allowed us to conveniently express the heterogeneous contexts involved. The same approach should also be im-

plementable in Haskell using standard GADT-based heterogeneous lists [71]; however, without access to notational conveniences as in Rocq, it remains future work to see how to regain a similar degree of ergonomics in that setting. On the dynamic typing side, the techniques described should be readily transferrable to other languages. For example, given Python's powerful reflection capabilities and its convenient decorator features, a deferred-binding style property language should be implementable on top of, e.g., Hypothesis.

Chapter

3 Quantitative Evaluation of Property-Based Testing

The second chapter of this thesis explores a fragmentation caused by the design of property-based testing libraries, leading to diverging features, separated ecosystems. In this chapter, I focus on yet another fragmentation, this time caused by something much more banal, a lack of general purpose evaluation benchmarks for PBT. The literature is full of interesting novel techniques for enhancing random generation and counterexample minimization, but these techniques are rarely evaluated within a coherent context. Many times, the studies have relied on bespoke benchmarks: John Hughes's tutorial on how to write properties [20] focuses on binary search trees; Luck [72] is evaluated on red-black trees, simply-typed lambda calculus and a stack-based information flow control machine; Smallcheck and lazy smallcheck [22] are evaluated on red-black trees, huffman encoding, the Turner program and a mate-in-N chess problem; Combinatorial PBT [37] is evaluated on System F; and coverage-guided PBT work builds around the information-flow-control workload developed for testing noninterference [29, 73, 74]. If one were to build a web of studies and the benchmarks they used, the result is somewhat reminiscent of the infamous public transport coverage of Ankara, a small number of hub workloads that many studies reuse, many one-off benchmarks that haven't been validated or reused in other studies, and a complete lack of standardization.

This lack of common comparative analysis surely contributes to a lack of understanding in the effects of these new techniques and changes; but the problem is persistent beyond the literature and into the libraries themselves. All the property-based testing libraries I have read

in the course of developing this thesis are full of bespoke decisions accumulated in the last 25 years. The choice of which sizing function to use, the shrinking algorithm, the default settings on budgets and parameters... Many of these decisions have never been subject to quantitative judgement beyond the initial discussions, never been evaluated at scale, or understood in the broader context of PBT usage.

In the beginning of my Ph.D. we set out to remediate this fragmentation and help build a common quantitative understanding for the field. With Jessica Shi and Harry Goldstein implementing the common evaluation framework in Python as well as the Haskell workloads, and I working on the implementation of the Rocq pipeline and workloads, we have published "Etna: An Evaluation Platform for Property-Based Testing (Experience Report)" [75] in ICFP 2023.

After the initial publication, I have taken the lead role in future development of the ETNA platform and horizontally grown it for OCaml with Nikhil Kamath, Racket with Ceren Mert and Justine Frank, for Haskell with George Miao, and for Rust by myself. The initial implementation as a library has been replaced with a standalone tool that could be easily run cross-platform on many machines, eventually to the point we can start to run our experiments as GitHub Actions. Many of these changes have found their ways into publications, our JFP extension [63] for the ICFP 2023 paper has incorporated the work by Ceren and Nikhil as well as Tin Nam Liu from UPenn working together with Jessica.

The rest of this chapter includes excerpts edited from these papers, mainly the 2026 JFP extension of ETNA, with a major section on evaluation of shrinking from our experience report that appeared in Haskell 2026 with George and Leo.

3.1 ETNA

ETNA is, as you may have thought of, named after the famous Sicilian volcano. The name follows a convention established by prior work in the fuzz testing community, Lava [76] and Magma [77], both of which have been instrumental to our design, particularly with the ground truth approach of Magma.

During the design and development of ETNA, we focused on a few key design principles, centered around usefulness, extensibility, and maintainability.

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

How should an evaluation platform measure the effectiveness of a generator? The software testing literature offers two common answers: *code coverage* and *mutation testing*. Code coverage is attractive because it is easy to instrument and compare, but it is at best an indirect signal: higher coverage does not always translate to better bug finding [78,79]. For ETNA, we instead chose mutation testing [80], which measures the effectiveness of a testing technique by artificially injecting mutations into the system under test and checking whether testing is able to detect them.

Mutations used in prior evaluation work fall on a spectrum from manually sourced to automatically synthesized [74,77,79,81]. ETNA opts for manual sourcing. This choice makes the benchmark harder to populate, since every task requires a meaningful mutant and a property that exposes it, but it also lets us maintain a clearer ground truth: each mutant should violate some aspect of the property specification. The workloads, properties, and injected bugs used in this chapter are detailed in § 3.3.

3.1.2 Use minimal, but precise interfaces.

A key challenge during ETNA's development was that the platform needed to gather comparable data about the testing effectiveness of a wide variety of existing PBT frameworks, each of which reports such data in an ad-hoc and non-standardized manner. Most frameworks report the number of inputs generated before a counterexample is found. Far fewer expose timing statistics, and none of the frameworks we studied initially offered a detailed breakdown of generation, testing, and minimization time. Similarly, most frameworks report the number of inputs that fail to satisfy a precondition as discards, while others, such as Rust's QuickCheck and Racket's RackCheck, do not separate discards from passing tests.

These differences matter because ETNA is not evaluating a single library in isolation; it is trying to compare workloads, strategies, frameworks, and languages. To make those comparisons meaningful, we settled on a small set of metrics that are easy to measure across frameworks and on a precise output format that implementations can validate themselves: JSON conforming to a schema exposed by the ETNA protocol.[1] The resulting interface is intentionally modest. It does not try to encode every feature of every PBT library. Instead, it records the information that ETNA needs to run experiments reproducibly and compare their outcomes.

3.1.3 Every ETNA capability should be available to use manually.

ETNA acts as an orchestration mechanism: it invokes testing tools, toggles mutations, parses results, and performs analysis over the collected data. Such orchestration is fragile

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

even when every component is individually well engineered, because ETNA sits between loosely coupled systems that were developed independently. In practice, any opaqueness in this process can make failures difficult to diagnose and, in the worst case, unrecoverable.

The final guiding principle of ETNA is therefore transparency. Each step of an experiment should correspond to an operation that a user can understand and, when needed, reproduce manually. This principle shaped the transition from the initial library implementation to the standalone command-line tool described in the next section. Rather than hiding build steps, mutation activation, test execution, and analysis behind a single opaque pipeline, ETNA exposes them as ordinary commands. That design makes the platform easier to debug, easier to extend to new languages, and easier to trust when an experiment produces surprising results.

3.2 ETNA Tools

Adhering to the principles outlined in § 3.1, I have built a set of tools that allow for easier experimentation. The first is the `etna-cli` [82], a tool that allows for managing ETNA *experiments*, standalone version-controlled units that hold *tests* that mark the set of *tasks* and the corresponding parameters such as timeout, number of trials, or any other runtime injected parameter to the running test. Experiments are created by *etna experiment new*, reproduction and collaboration is as seamless as possible, a user can upload the automatically committed repository to a version control tool such as GitHub, the collaborator can clone the repository; run *etna experiment register* to start tracking the experiment.

Another importance concept is a *workload*, a workload is a program with a set of injected

bugs; and properties that define correctness for some behaviors of the workload. Bugs are paired with the subset of properties they invalidate to constitute *tasks*, targets for the property-based testing tools to evaluate on. Each ETNA workload is hosted on its own GitHub repository, and the `etna-cli` hosts an index [83] to all currently validated workloads.

Workloads, in addition to defining tasks for the PBT tools, define a set of *steps* for configuring, building, and running the workload. The `schemas` [84] section in the `etna-cli` repository define the shape of this schema as well as the expected shape of the reporting mechanism for the results and the persistent messaging format for *capabilities*. Currently, ETNA workloads have four possible capabilities: *solve* for running a full testing campaign for a given *testing strategy*; *sample* for sampling a random generator; *test* for testing a given set of inputs against the strategy (sample-test pairing is used in the cross-language experimentation in § 3.4.2.3); and finally *shrink* for minimizing a given input with a shrinking algorithm.

The testing strategies can be across *frameworks*, such as implementing the same generator in Hedgehog [23] and QuickCheck [85] as we have done in § 3.4.3, or within frameworks such as comparison of derived and bespoke generators in Rocq's QuickChick [21] as we have done in § 3.4.2.1. ETNA workloads typically have an entry point that acts as a simple CLI for setting the current property and strategy; adding a new strategy or property is as simple as adding the required functionality to the workload and updating the entry point, although the design is flexible as the procedure for running a workload is ultimately defined in the *steps* file.

Once the user designates a task to solve, they define the strategy, the timeout, the number of trials and any runtime parameters, all of which constitute a *test*. The users can bundle tests together into a test file, and run them with additional settings we have found useful in our own experiments. `--short-circuit` tells the system to only run a given test until the first

failure to solve, meaning that either the predefined maximum budget within the strategy was exhausted, or ETNA terminated the test with an external timeout. `--parallel` allows running multiple tests in parallel, which is especially useful as many PBT tools are single threaded so multicore execution enables faster experimentation with minimal interference.

The test is run via the command `etna experiment run --tests <test-name>`, after which ETNA reads the workload running procedure from the provided steps file; runs the configuration, *activates the mutation*, builds the project, and runs the test for the given number of trials. One point I am yet to talk about is how the mutations are defined and how are they managed, which I will now do before diving into the workloads in § 3.3.

Workloads encode mutations directly in the source language using a comment syntax recognized by ETNA's mutation management tool [86]. For example, the following Haskell implementation of binary-search-tree insertion contains both the correct implementation and a mutant named `insert_duplicate_entries`.

```
insert k v Leaf = Node Leaf k v Leaf
insert k v (Node l k' v' r)
  {-! -}
  | k < k' = Node (insert k v l) k' v' r
  | k > k' = Node l k' v' (insert k v r)
  | otherwise = Node l k' v r
  {-!! insert_duplicate_entries -}
  {-!
  | k < k' = Node (insert k v l) k' v' r
  | otherwise = Node l k' v' (insert k v r)
  -}
  {- !-}
```

The uncommented base implementation preserves the search-tree invariant by replacing the value when the inserted key is already present. When ETNA activates `insert_duplicate_entries`,

the marked mutant body replaces the base code with an implementation that treats equality like the right-subtree case, thereby allowing duplicate keys. In general, a variation begins with a language-specific marker comment, contains the base implementation, then lists one or more named mutants, each with a header and a marked replacement body. Since the markers are comments, the workload continues to compile normally when no mutant is active.

I have also spent time carrying this mechanism further by enabling multiple forms of mutations, and conversions between them. Today ETNA can use diff patches, C-Preprocessor macros, match-and-replace mutations, and finally intra-AST mutations that can reduce the experimentation time by removing the compilation requirement, which is currently only implemented for Rust.

3.3 Workloads

The initial ETNA workload suite was drawn from application domains that are both practically relevant to functional programming and common in the PBT literature. Not every experiment in this chapter uses every workload: the Haskell and Rocq experiments focus primarily on binary search trees, red-black trees, lambda calculi, and information-flow control, while the later OCaml and cross-language experiments reuse the portable subset of these workloads. The complete workload definitions, including their properties and associated mutants, are hosted in the ETNA workload repositories indexed by `etna-cli` [83].

Data Structures

The first workload, binary search trees (BST), is a small but useful baseline because BST generation appears repeatedly in property-based testing papers. In particular, John Hughes's

How to Specify It! [20] uses BSTs as an extended example for writing QuickCheck properties. ETNA's BST workload ports the mutations and properties from that example, including operations such as insertion, deletion, lookup, union, and validity checking.

The second workload, red-black trees (RBT), keeps the same broad data-structure domain but adds a substantially richer invariant. Red-black trees combine ordering constraints with balancing and coloring constraints, making insertion and deletion easy places to introduce subtle implementation mistakes. RBTs have also been used in prior PBT work [22, 72, 87, 88]. ETNA's RBT workload combines BST-like mutants with additional mutants targeting balancing and coloring errors.

Lambda Calculi and Type Systems

The third workload is a De Bruijn-indexed implementation of the simply typed lambda calculus (STLC) with booleans. Generating well-typed lambda terms is a long-running benchmark problem for PBT [89, 90]; the STLC mutations used in ETNA are drawn from the relevant fragment of a System F case study and mainly exercise mistakes in substitution, shifting, and lifting [37].

The fourth workload extends this line to System $F_{<:}$, following the fuller case study of [37]. Adding polymorphism and subtyping makes the workload large enough to expose more complex type-manipulation errors, including type substitution, shifting, and lifting. Bespoke generators for System F terms have also been studied in recent PBT work [37, 91], and the same ideas can be extended to the subtyping variant.

Parsing

The fifth workload is a parser and pretty-printer for Lu, a language based on Lua whose implementation was drawn from a Haskell course at the University of Pennsylvania. Its main property is a round trip: printing a valid Lu expression and then parsing the printed text should recover the original expression. The released workload includes a bespoke generator for valid Lu expressions.

Security

The sixth workload is information-flow control (IFC), based on the case study from [73,74]. The workload tests whether low-level monitors for abstract machines enforce noninterference: differences in secret data should not become publicly visible through execution. Its mutants systematically weaken security checks or taint-propagation rules, yielding tasks that are especially useful for evaluating feedback-guided generation strategies.

3.4 Experiments

3.4.1 Analysis and Presentation

ETNA is configurable in terms of timeout, number of trials, and different ways of the results of the experiments. However, most of the work I present in this thesis uses a configuration we argue is realistic in terms of running a PBT workload, and a common presentation technique we found useful.

All the experiments, unless explicitly noted, use 10 trials with a 60-second external timeout. A strategy is said to have *solved* a task if it succeeds in solving it in all 10 trials under the

timeout. We use *task-bucket charts*, a type of stacked-bar chart that adds up to the total number of tasks, marking progressively larger mean time buckets as presented in Figure 3.1 and used in the previous experiment presentations in Chapter 2 such as Figure 2.18 and Figure 2.22.

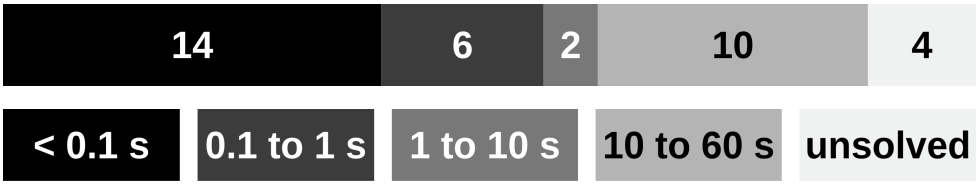


Figure 3.1: ETNA task-bucket chart presentation.
The upper chart shows a hypothetical strategy/workload result split across timing buckets; the lower key explains the bucket shading convention.

We mark a task *unsolved* if any of the 10 trials fail to solve it within the timeout, this is configurable in the visualization tool (`etna visualize`) command under the flag `any/all`, where `any` means a task is solved if at least one trial has found the bug, and `all` means a task is solved if all trials have solved it.

The thesis typically uses a number of bucket charts for different strategies on the same workload for easy visual comparison. Each row stacks up to 5 buckets in generation, and 6 buckets in shrinking experiments (which adds another bucket of 60-360s). The darkest bucket denotes tasks solved within 0.1 seconds, and the remaining buckets denote tasks solved within 1, 10, 60, and 360 seconds. The lightest bucket denotes tasks that were not solved within the timeout. For example, Figure 3.1 shows a strategy/workload result where 14 tasks are solved almost immediately and 4 tasks remain unsolved.

Some comparisons need more detail than a task-bucket chart alone can show. In those cases, I either use alternate bucket definitions, such as the `any` configuration, or supplement the bucket chart with statistical comparisons. To keep the figures self-contained, bucket-chart captions state the aggregation rule they use.

3.4.2 Evaluating Random Generation

In this thesis, I have focused on the parts of the ETNA papers [63, 75] that I have personally worked and focused on. Hence, the experiments in this section omit the Haskell experiments comparing QuickCheck, SmallCheck and LeanCheck as well as the analysis on the effect of size and enumeration order. The Rocq experiments in § 3.4.2.1 evaluate different generation strategies in QuickChick; the OCaml experiments in § 3.4.2.2 evaluate generators across QCheck, Crowbar and Base_quickcheck; and cross-language experiments in § 3.4.2.3 use generators from 5 different languages to sample inputs, and use a Rust port of the BST/RBT/STLC workloads for evaluating generation time across language ecosystems.

3.4.2.1 Evaluation of Random Generators in Rocq

We now turn our attention to the single-framework but multi-strategy landscape in Rocq. PBT in Rocq revolves around QuickChick [56], which, in addition to the type-based and bespoke strategies, provides two additional options: a *specification-driven* strategy that derives correct-by-construction generators from preconditions in the form of inductive relations [42] 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 [29].

Both papers exemplify the lack of performance comparisons across approaches discussed in the introduction. First, [42] 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 hand, FuzzChick [29]

is evaluated in the more realistic IFC workload of [74] 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.

Comparison of Fuzzers, Derived Generators, and Handwritten Generators

Our evaluation aimed 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 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 [92]); for IFC, such specifications do not exist, so the specification-driven generators of [42] do not apply.

Results

In Figure 3.2, we visualized the results of the experiments with a task bucket chart. Results for the simple BST workload (Figure 3.2a) 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)

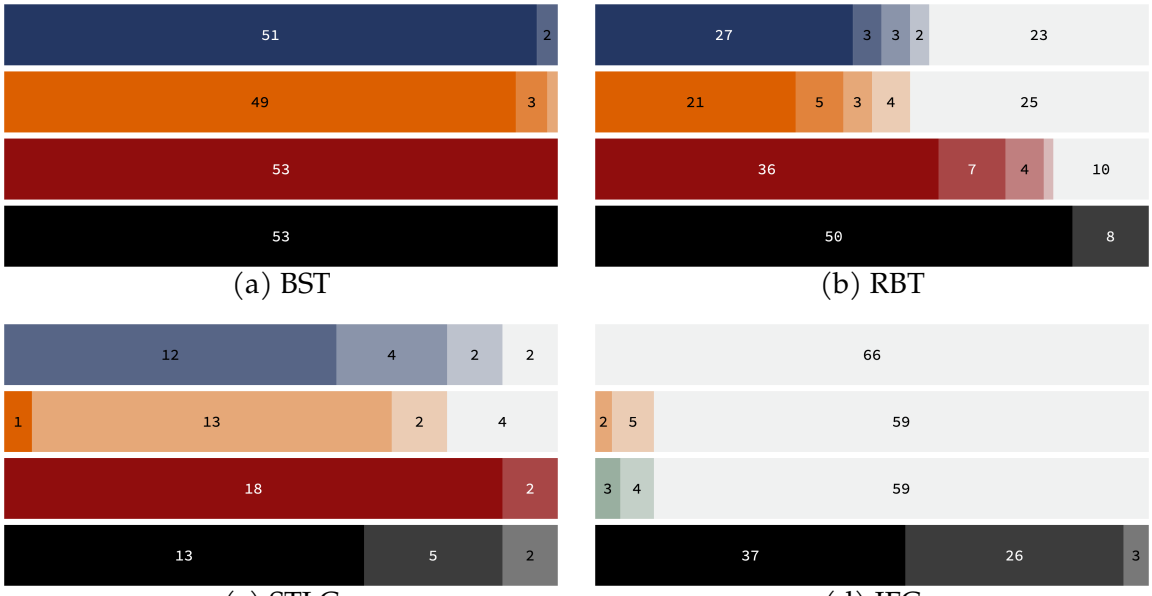


Figure 3.2: Effectiveness of Rocq generation strategies on four workloads. Buckets use the all-trials rule: a task is solved only if all 10 trials finish within 60 seconds.
■ = Type-based generator, ■ = Type-based fuzzer,
■ = Specification-based generator ((a) - (c) only), ■ = Variational fuzzer ((d) only),
■ = Bespoke generator.

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.

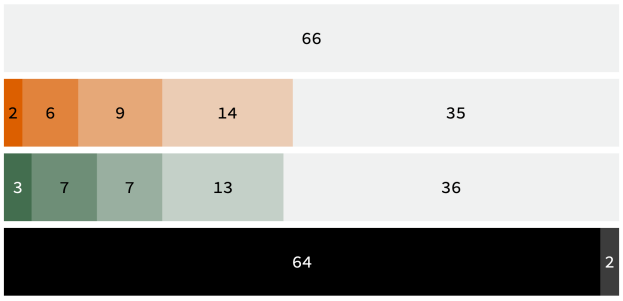


Figure 3.3: IFC Tasks solved within the timeout in one or more trials.
Empty = Type-based generator.
= Type-based fuzzer,
= Variational fuzzer,
= Bespoke generator.

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, *variational fuzzer*, borrowed from the original paper that introduced FuzzChick [29] 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 3.3 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 3.3), but less than 10 tasks are fully solved (Figure 3.2d).

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.

Validation and Improvement of Fuzzers

The conclusion of [29] 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 ETNA?*

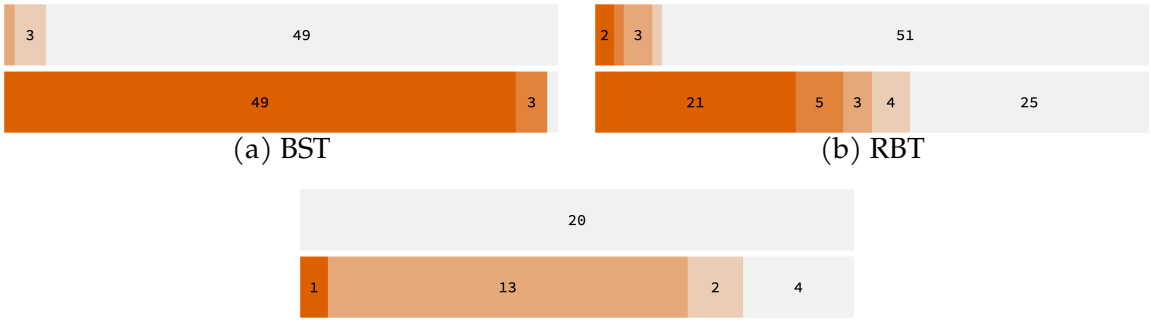
We focused on two different aspects of fuzzing: *size* and *feedback*. FuzzChick's generation strategy started small but quickly increased to quite large sizes, relying on the idea of "combinatorial advantage" discussed in the ETNA size experiment [75]—i.e., that larger inputs contain exponentially many 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 [12]) 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 3.4 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.[2]



(a) BST (b) RBT

(c) STLC

Figure 3.4: Comparison of the original FuzzChick (top) with the tuned one (bottom). Buckets use the all-trials rule: a task is solved only if all 10 trials finish within 60 seconds.

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:

[2]Note that we continued on this line of work via Programmable PBT, presenting further algorithmic improvements that we could not attempt in this study in § 2.6.3.1.

```
Fixpoint loop fuel ... :=
  match fuel with
  | O => (* base *)
  | S fuel' => (* rec *)
  end.
```

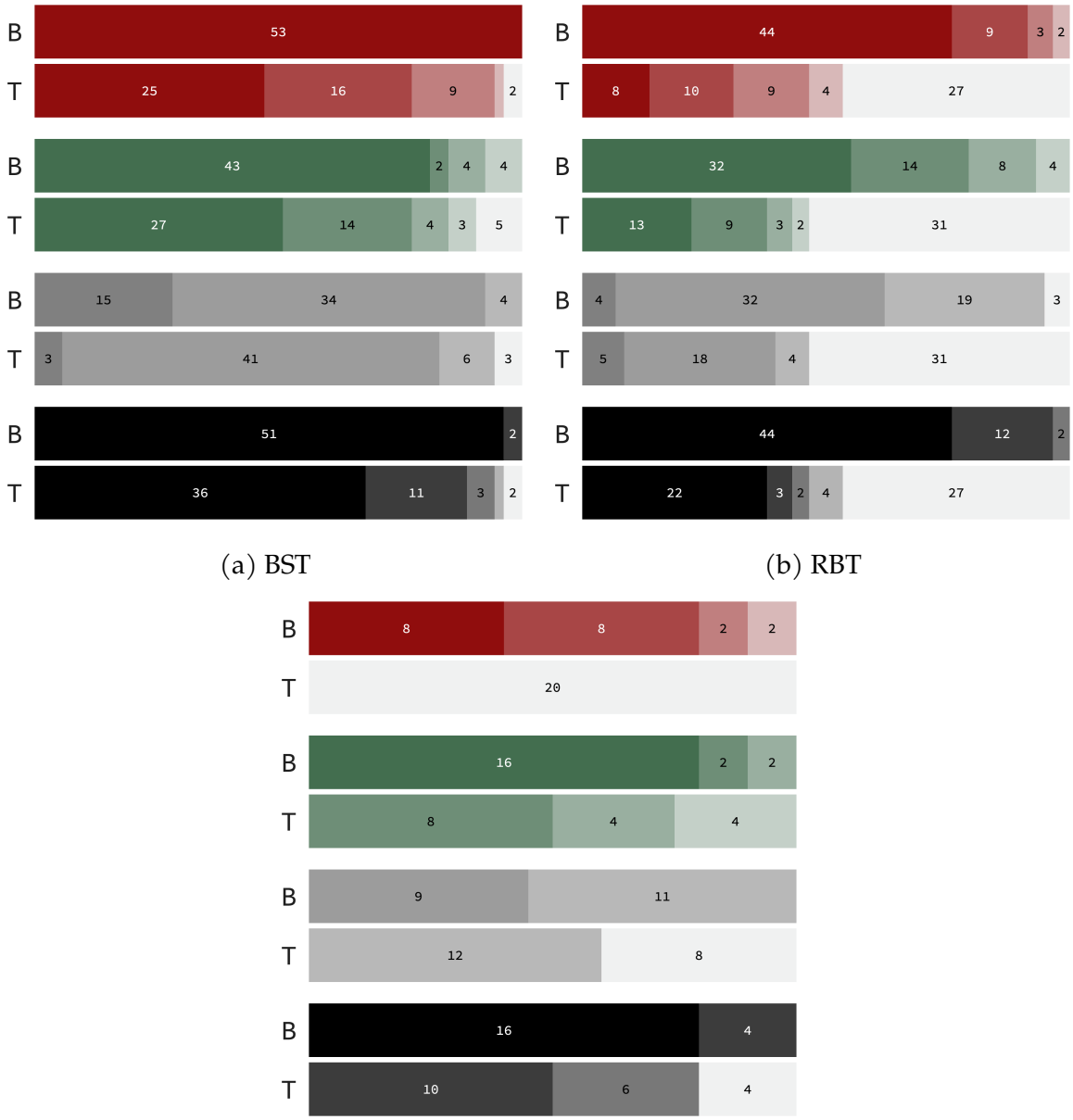
```
let rec loop fuel ... = (fun fO fS n ->
  if n = 0 then fO () else fS (n-1))
    (fun () -> (* base *))
    (fun fuel -> (* rec *))
    fuel
```

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!

3.4.2.2 OCaml

Whereas QuickChick is the de facto 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.



(a) BST

(b) RBT

(c) STLC

Figure 3.5: Effectiveness of OCaml generation strategies on three workloads. The first and second buckets for each framework represent the bespoke and type-based generators respectively. Buckets use the all-trials rule: a task is solved only if all 10 trials finish within 60 seconds.
■ = QCheck, ■ = Crowbar (Random), ■ = Crowbar (AFL), ■ = Base_quickcheck.

Controlling for Size

As the ETNA JFP size experiment 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 the ETNA JFP size experiment, 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[3]. As a result, we re-implemented a list generator using the rest of Crowbar's API to construct longer lists.

Figure 3.5 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[4] 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 (§ 3.4.2.1), if one takes the effort to handcraft bespoke generators that satisfy a property's precondition by construction, coverage-guided fuzzing only adds

[3]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.

[4]`https://github.com/alpaylan/etna-cli/blob/docs-bst/workloads/OCaml/BST/lib/generators/gen_bespoke_crowbar.ml`

overhead for minimal gain.

3.4.2.3 Cross-Language Comparison of PBT Frameworks

ETNA allows for running complex experiments that can provide powerful insights to property-based testing practitioners or framework developers. However, the scope of these experiments were initially limited to the level of a single programming language. Given two testing frameworks or generation strategies within the same language, we could measure and compare their bug-finding 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 [91]; for another, a bespoke generator for file-system interactions written in Erlang were used to test Dropbox's Python-based file synchronization service [93]. 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 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. Depictions of the intra-language and cross-language workflows of ETNA are presented in Figure 3.6.

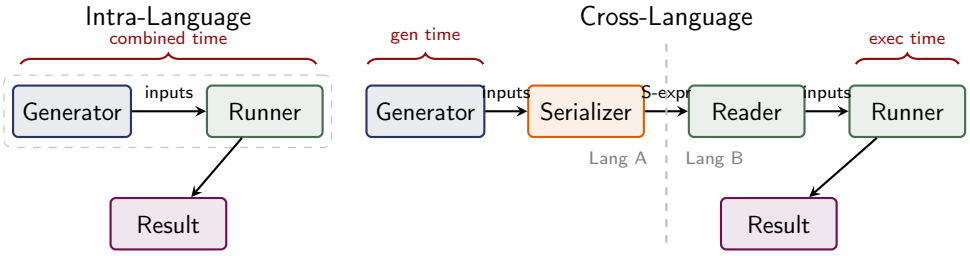


Figure 3.6: Intra-language vs cross-language workflows. Left: generator and runner in same language with combined timing. Right: generator serializes inputs; separate reader/runner enables decoupled timing metrics across language boundaries.

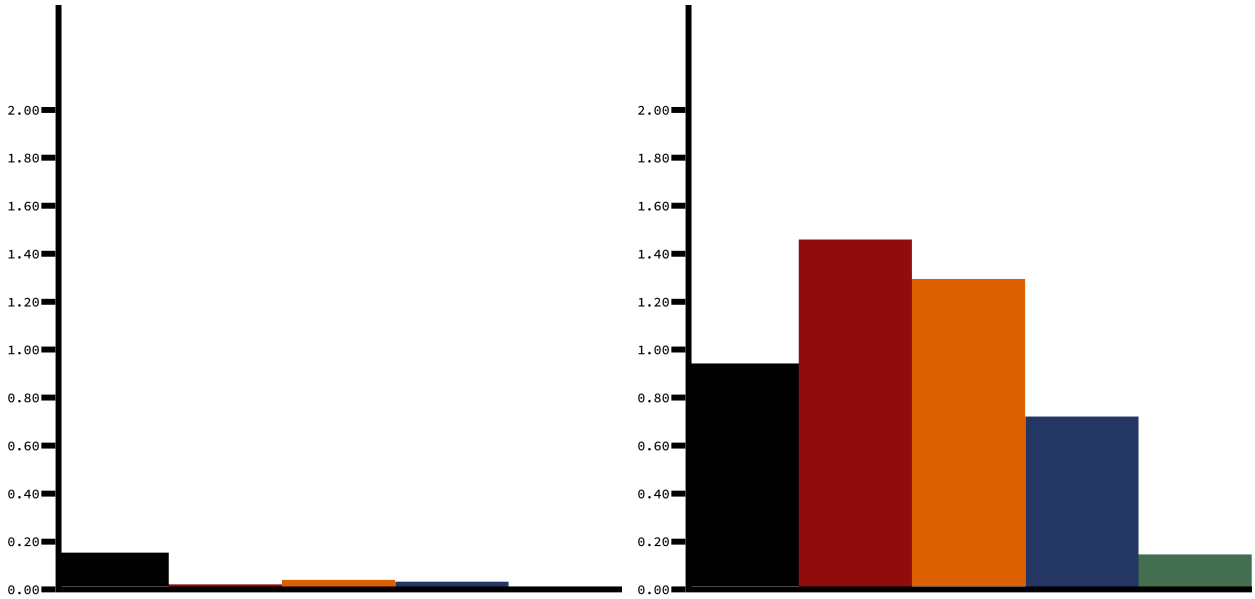
To demonstrate ETNA's new capabilities, we pit bespoke generators written in Haskell,

Rocq, OCaml, Racket, and Rust against the three workloads. The results are shown in Figure 3.7. 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 simultaneously 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).

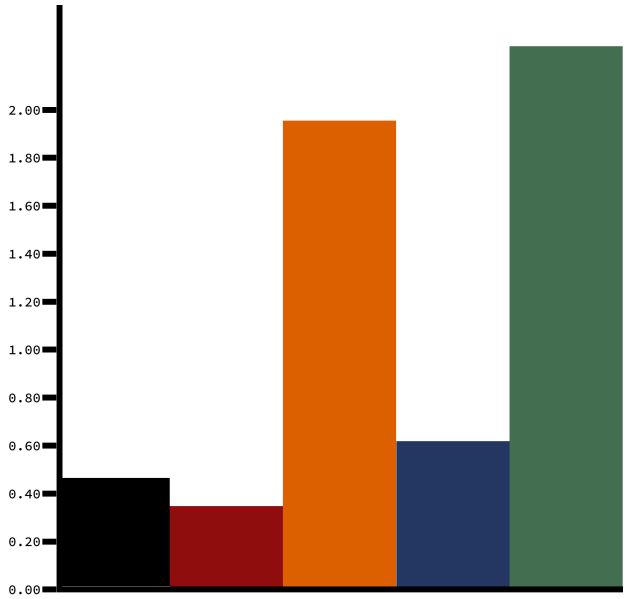
Another takeaway from Figure 3.7 is the difficulty of the workloads themselves: BST is comprised of 52 tasks, and the slowest generator (Haskell's QuickCheck) still finds all 52 injected bugs in 130ms; on the other hand, STLC is comprised of only 20 tasks, but the fastest generator (Rocq's QuickChick) 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 experimentation.



(a) BST

(b) RBT



(c) STLC

Figure 3.7: Generation time to failure (in seconds) for bespoke generators written in different languages.
■ = Haskell – QuickCheck, ■ = Rocq – QuickChick, ■ = OCaml – QCheck,
■ = Racket – RackCheck, ■ = Rust – QuickCheck.

3.4.3 Evaluating Shrinking

The generation experiments above show why ETNA is useful as shared evaluation infrastructure: it lets us compare strategies, frameworks, and implementation languages against the same workloads and bug corpus. But bug finding is only the first half of a property-based testing run. Once a strategy finds a failure, the framework still has to turn the original random input into a counterexample that a programmer can understand. That second phase has its own design choices and costs, so an evaluation methodology that stops at time-to-failure gives an incomplete picture of a PBT framework. This section extends the ETNA perspective from generation to shrinking by comparing QuickCheck's structural shrinking with the integrated shrinking approaches used by Hedgehog [23] and Falsify [25].

We can start by remembering a variant of the binary-search tree insertion property from the last two chapters:

```
prop_insert :: Tree -> Key -> Value -> Maybe Bool
prop_insert t k v = isBST t ==> isBST (insert t k v)
```

To test this property, users specify how to generate test inputs, run the property-based testing loop, the random generation exhausts the search space to detect behaviors that is otherwise unexplored. However, the very same randomness that tames the exponentially large search space also means that most of the time the first counterexample found is unusable for debugging: it simply contains too much noise. For example, leveraging a random tree generator to test the `prop_insert` property above, often yields counterexamples like the following:

```
((T (T (T (E) -684 128 (E)) -563 533 (T (E) -552 252 (T (T (E) -479 27 (E)) -412 -910 (T
(T (E) -395 79 (T (E) -349 -332 (E))) -330 779 (E))))) -253 -554 (T (T (T (T (T (T (E)
-252 -550 (T (T (T (E) -231 -330 (E)) -222 -19 (E)) -175 279 (T (E) -113 136 (T (E) -111
```

```
170 (E))))) -48 30 (E)) -25 238 (E)) 76 632 (T (T (T (E) 85 -717 (T (E) 90 -781 (E))) 161
167 (E)) 192 -295 (T (T (T (E) 200 -19 (E)) 256 -286 (E)) 285 -808 (T (E) 306 156 (E)))))
401 260 (T (E) 402 95 (E))) 460 661 (T (T (E) 545 -389 (E)) 743 60 (E)))), -25, 5)
```

Such an input is difficult to use for debugging: of the 31 nodes in the tree, only one is needed to trigger the fault. If only we could devise a method to simplify this input to the core of the bug, we could present the user with a simpler counterexample such as `(T (E) -25 0 (E), -25, 1)`, from which the bug is immediately apparent.

This problem is solved by counterexample minimization, also known as *shrinking*. QuickCheck uses a second straightforward shrink-and-test loop: users provide a function that maps a counterexample to smaller variations of it, and the framework repeatedly tries those variations until one of them is also a counterexample. The process repeats until a (local) minimum is found. This particular approach is called *external* shrinking; the shrinker is written independently from the generator and focuses on minimizing the generated structure. This independence poses a problem that valid counterexamples (produced by hand-crafted generators that produce inputs valid-by-construction) are minimized into invalid candidates, spending the testing budget in sifting through the invalid cases.

It would be nice if we could have correct-by-construction shrinkers too, and that is precisely the problem solved by *integrated*, or *internal*, shrinking. Python's Hypothesis [11] exemplifies integrated shrinking as an alternative path, where instead of independent shrinkers, the generators themselves are used in shrinking. That is achieved by shrinking the randomness coming into the generators and then replaying them, rather than shrinking the structures produced, usually by representing this randomness as a *randomness buffer* or a *choice sequence* to maintain some structural correspondence with those outputs. Variations of this

approach are found in other Haskell property-based testing frameworks, most prominently Hedgehog [23] and Falsify [25].

How do these approaches compare to each other? Integrated shrinking is quite appealing in theory: it relieves the users of the responsibility to implement shrinkers, automatically giving rise to correct-by-construction shrinkers instead of requiring users to program them themselves. However, as noted in the study of the Hypothesis reducer [16], external shrinkers can be more *effective* in finding smaller counterexamples than internal shrinkers. For example, code generators for testing compilers might produce test inputs with a particular boilerplate structure which will always be present with internal shrinking, regardless of whether it's necessary. The performance implications of the shrinking choice is also a key concern: given a fixed testing budget, overhead from using an internal-shrinking-based approach can mean that the testing campaign will be able to cover a smaller part of the search space. Ideally, the users should have an informed view of the trade-offs present in selecting between different approaches.

The PBT literature, however, overwhelmingly focuses on evaluating bug-finding performance with little quantitative evaluation of the shrinking process. The rest of this section presents our work on extending ETNA to support evaluation for shrinking. This extension entails defining quantitative metrics for shrinking effectiveness and cost, including tree edit distance (TED) to a ground-truth minimum and time per unit of shrinking progress; extending the current ETNA workloads in Haskell with the reporting of shrinking related metrics as well as implementing Hedgehog and Falsify adapters.

3.4.3.1 Background: Understanding Shrinking

Shrinking is a conceptually simple problem. The shrinking algorithm generates smaller candidates, checks whether they still reproduce the failure, and repeats until no smaller failing input is found. The search typically reaches a local minimum rather than a guaranteed global minimum, so the structure of the candidate space matters.

Researchers have proposed two broad implementations of this pattern. The first is called "external", "type-based", or "structural" shrinking, and it uses explicit functions that operate directly on the structure of the counterexamples. The second is called "internal" or "integrated" shrinking, where shrinking behavior is integrated into generators. This section briefly presents the respective techniques used in the Haskell PBT frameworks we evaluate.

Structural Shrinking—QuickCheck

In structural shrinking, the user provides a *shrinker*, a pure function `a -> [a]` that maps a value to a list of *candidate smaller values*. When a test fails, the framework searches those candidates for a new failing input, repeating until no candidate reduces further. The shrinker operates directly on the structure of the value and is entirely independent of the generator. Take QuickCheck's implementation as an example, the user-facing API is the `Arbitrary` typeclass:

```
class Arbitrary a where
    arbitrary :: Gen a
    shrink :: a -> [a]
```

The `arbitrary` and `shrink` methods are decoupled. The shrinker has no access to the `Gen` monad and no knowledge of how the value was produced. This means any invariants encoded in the

generator must be re-enforced independently in `shrink`—if the shrinker produces a candidate that violates a precondition, the property may fail for the wrong reason.

QuickCheck implements this typeclass for basic Haskell types, such as integers (which shrink to some smaller number) or lists (which can shrink to a sublist or to a list where one of its elements has recursively been shrunk). Still, users must provide this implementation for any user-defined datatypes, or any types whose shrinking behavior they want to override.

The Search Loop

Internally, QuickCheck represents the shrink space as a lazy rose tree [1]:

```haskell
data Rose a = MkRose a [Rose a]
```

Each `MkRose v cs` node holds a test result `v` paired with the shrink candidates of `v`, each rooted in their own subtree. The tree is built lazily, with only the subtree actually visited ever being forced.

Once a failure is found, QuickCheck searches the tree with a greedy left-to-right depth-first traversal. At each step, if the first candidate `t` still fails the property, descend into its children `ts'`; otherwise advance to the next sibling `ts`. The algorithm never backtracks, so the result is a *local* minimum. This makes the ordering of the shrink list significant, since the greedy search locks in the first improvement it finds, front-loading the list with globally small values gives the algorithm its best chance of reaching the global minimum.

SampleTrees—Falsify

In integrated shrinking, generators carry their shrinking behavior implicitly. Instead of a separate shrinker function, the generator runs on an internal randomness buffer, and shrink-

ing works by shrinking the buffer with smaller values and re-running the same generator. Falsify implements this approach with *SampleTrees*, a lazy tree structure representing both the original random samples and all possible shrunk variants.

The user-facing API is simply a generator monad:

```
newtype Gen a = Gen { runGen :: SampleTree -> (a, [SampleTree]) }
```

The `Gen` monad consumes a `SampleTree`, a lazy binary tree of `Word64` samples, and returns both the generated value and a list of *shrunk sample trees*. Each shrunk tree represents a candidate shrink of the input: it replaces one or more samples with smaller values. When a property fails, Falsify re-runs the generator on each shrunk tree until no candidate produces a smaller failing input.

Shrink Tree—Hedgehog

Hedgehog also uses integrated shrinking, but exposes it through a generator that directly produces a shrink tree. Internally, a generator is a function of the current size and random seed:

```
type Gen = GenT Identity

newtype GenT m a = GenT {
    unGenT :: Size -> Seed -> TreeT (MaybeT m) a
}
```

Running a `Gen` produces a lazy rose tree, where the root is the generated test case and children are the shrink candidates. Discarded inputs are represented by the `MaybeT` layer. As with Falsify, the user writes only a generator; the shrink behavior is attached to the choices made

by that generator instead of a separate function.

3.4.3.2 Evaluation

In our evaluation, we asked two questions, (1) how *effective* is the shrinker via measuring the distance to the truly minimal input, and (2) how *performant* is the shrinker via measuring the time it takes to shrink. Concretely, we measured the following metrics for answering our questions:

- **Minimal counterexample discovery:** Properties can tested with many different techniques for supplying inputs, from manual user-written inputs to our usual random input generation to symbolic execution to exhaustive enumeration [22, 38, 94]. We leverage an enumerative property-based testing strategy, LeanCheck [94], to search for the smallest discovered input that constitutes a bug. We use these LeanCheck results as ground-truth minima for the tasks where exhaustive search finds the bug.

- **Effectiveness of shrinking strategy:** Regardless of the original input, an effective shrinker should move the reported counterexample closer to a minimal failing input. We use the tree edit distance [95] computed via zss package in Python [96] between the ground truth minimal counterexample and the shrinker output to measure effectiveness.

- **Performance of shrinking strategy:** We measure shrinking time and time per unit of tree-edit-distance reduction to understand how much time the shrinker spends per unit of progress instead of just looking at the absolute numbers. We measure different shrinker parameter choices in the libraries to understand their effects.

- **Cost of shrinking strategy:** There is an inherent cost to the shrinking strategies them-

selves that affects the performance of the bug-finding stage of the testing process. We measure the bug-finding capabilities of different frameworks as well as the overhead of enabling the shrinking.

- **Stability of shrinking strategies:** It is possible to write different generators for the same input space. We have briefly mentioned how it is possible to write correct-by-construction generators that target the valid subset of the input space instead of generators that samples the entire domain, but we can also have different styles of correct-by-construction generators too, such as API-based generation, which instead of generating the structure itself, generates operations that mutate it. We measure the difference in shrinking performance and effectiveness in the context of different generation strategies in our evaluation.

Experiments

In our evaluation, we measured bug-finding time (time spent before a failing execution) and shrinking time (time spent after a failing execution) as performance metrics. The libraries provide shrinking budget controllers, albeit not very well documented, that we use to turn off shrinking as well as tune it with different parameters to measure its effect.

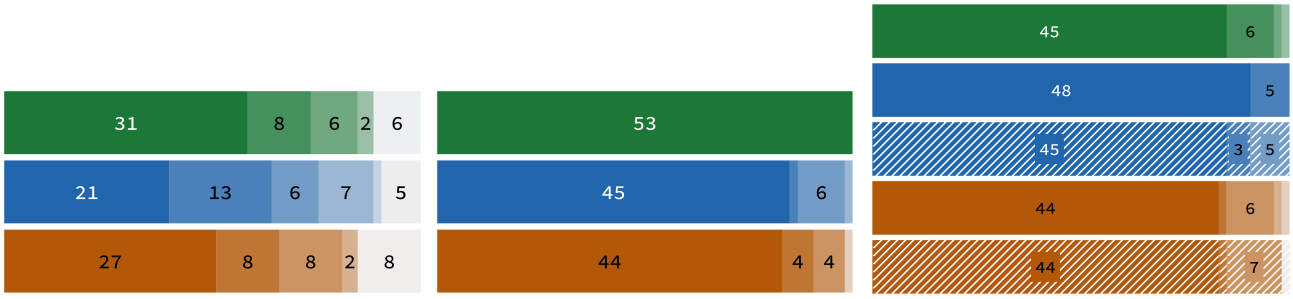
For BST and RBT, we used three different generators that represent the most common generators in the literature: a naive, type-based generator that uses rejection sampling for discarding invalid trees; a correct-by-construction generator that directly generates valid trees; and a second correct-by-construction generator that generates key-value pairs that are inserted to the trees; we will refer to the latter as an API-based generator, as it leverages the API of the structure under test to generate instances that satisfy its invariants. In all three

frameworks–QuickCheck, Hedgehog and Falsify–we tried to create the same generator design as closely as each API allowed. Specifically for the correct-by-construction BST generators, we created additional idiomatic variants for Hedgehog (that uses `HH.recursive`) and Falsify (that mimics `Falsify.Generator.bst`) and evaluated the effects of the changes. For STLC and $F_{<:}$, we ported the existing naive and correct-by-construction QuickCheck generators. A notable usability advantage of integrated shrinking is that it relieves the user of the requirement to write shrinkers [16,25]. In order to keep the comparison fair, we used the `genericShrink` from `generic-random` [41] library that automatically derives shrinkers for Haskell types.

Comparison of Bug-Finding Performances

The BST results in Figure 3.8 match the intuitive expectation that correct-by-construction generators largely eliminate the wasted effort of naive type-based generation, which spends substantial effort producing invalid trees. In the bucket charts, almost all API-based and correct-by-construction generators solve every BST task, while the type-based generators fail several tasks (`not found` bucket); API-based QuickCheck generator has the strongest bug-finding profile, placing all tasks in the fastest bucket.

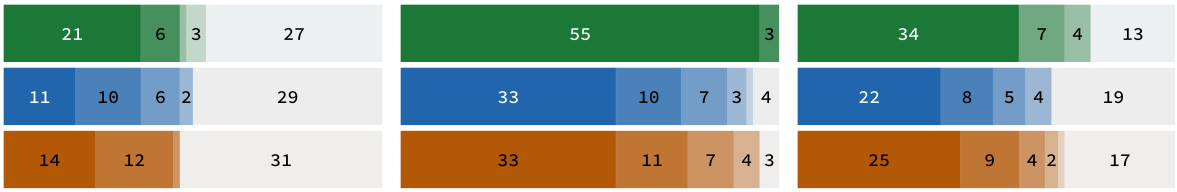
Across libraries, QuickCheck is significantly faster for bug-finding in both the type-based and API-based settings. We model the experiments as a paired design with repeated measures following Demšar [97]: each task is a data set, trial-level measurements are collapsed to per-task medians, and we run a Friedman test followed by Holm-corrected Wilcoxon signed-rank post-hoc tests. For BST bug-finding time, the type-based and API-based comparisons are statistically significant ($p < 0.001$): QuickCheck has lower bug-finding times than both Hedgehog and Falsify, and Falsify also has lower bug-finding times than Hedgehog in



(a) Type-based generation (b) API (c) Correct-by-construction

Figure 3.8: Bug-finding bucket charts on BST
Buckets use per-task median time-to-failure; `not found` means at least one trial failed to find the bug within the timeout.
= QuickCheck, = Hedgehog, = Idiomatic Hedgehog,
= Falsify, = Idiomatic Falsify

the type-based setting. In contrast, the correct-by-construction generators are statistically indistinguishable for bug-finding time among successfully solved tasks.[5]



(a) Type-based generation (b) API (c) Correct-by-construction

Figure 3.9: Bug-finding bucket charts on RBT
Buckets use per-task median time-to-failure; `not found` means at least one trial failed to find the bug within the timeout.
= QuickCheck, = Hedgehog, = Falsify

The RBT results in Figure 3.9 show a sharper separation between generator families than BST. Naive type-based generation struggles to produce valid inputs: all three libraries fail to solve half of the RBT tasks. API-based generation substantially improves this picture, with QuickCheck solving all tasks and placing them within one second, while Hedgehog and

[5]To avoid assigning arbitrary penalties, the paired statistical comparisons exclude `not found` cases and are therefore computed only on tasks with successful failed trials for all compared strategies. This can differ from the visual impression of the bucket charts, where `not found` is shown explicitly.

Falsify miss only a few tasks. Correct-by-construction generation is in the middle: it improves over type-based generation, but still leaves more tasks unfound than API-based, especially for Hedgehog and Falsify.

The paired statistical tests agree with the bucket-chart ordering for bug-finding time. All three generator-family comparisons are significant ($p < 0.001$). QuickCheck is significantly faster than both Hedgehog and Falsify in every generator family. Falsify is also significantly faster than Hedgehog for type-based and correct-by-construction generation, while the API-based comparison between Hedgehog and Falsify is not significant after Holm correction.

(a) Type-based generation

(b) Correct-by-construction

Figure 3.10: Bug-finding bucket charts on STLC
Buckets use per-task median time-to-failure; `not found` means at least one trial failed to find the bug within the timeout.
■ = QuickCheck, ■ = Hedgehog, ■ = Falsify

The STLC results in Figure 3.10 follow the same pattern as BST and RBT: the bucket charts show slower and less complete bug-finding for type-based generation than for correct-by-construction generation. For type-based generation, the Friedman test does not find a significant difference in bug-finding time, matching the similar bucket profiles for QuickCheck, Hedgehog, and Falsify. For correct-by-construction generation, the difference is significant ($p < 0.001$): QuickCheck is significantly faster than both Hedgehog and Falsify, while Hedgehog and Falsify are statistically indistinguishable from each other after Holm correction.

(a) Type-based generation

(b) Correct-by-construction

Figure 3.11: Bug-finding bucket charts on $F_{<:}$
Buckets use per-task median time-to-failure; `not found` means at least one trial failed to find the bug within the timeout.
■ = QuickCheck, ■ = Hedgehog, ■ = Falsify.

The $F_{<:}$ results in Figure 3.11 mirror STLC but with a larger gap among the naive type-based generators. Among the successful runs, QuickCheck has the strongest profile, followed by Falsify, while Hedgehog places substantially more tasks in slower buckets. Correct-by-construction generation removes the coverage problem entirely: all generators solve every task within one second, with QuickCheck placing all tasks in the fastest bucket.

The statistical tests agree with this ordering for bug-finding time. In the type-based comparison, the Friedman test rejects the null hypothesis of equal medians ($p < 0.001$), QuickCheck is faster than both Hedgehog and Falsify and Falsify faster than Hedgehog. In the correct-by-construction comparison, QuickCheck is faster than both Hedgehog and Falsify, and Falsify is faster than Hedgehog; the results of the statistical tests can be found in Appendix A.1.

Comparison of Shrinking Effectiveness

Figure 3.12 presents Cumulative Count Plot (CCP) charts denoting shrinking effectiveness for different libraries where X axis is the edit distance between the reported shrunk counterexample against the ground truth minimums computed via exhaustive search using

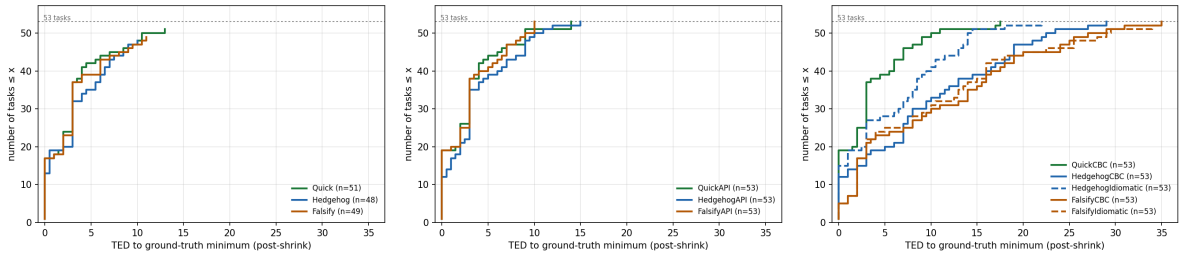
LeanCheck and the Y axis is the number of tasks for the corresponding workload (top-left is the best, top-right indicates a long tail).

The type-based campaigns show relatively little shrink movement across all four workloads. All four workloads have preconditions–invariant preservation in BST/RBT and well-typedness in STLC/$F_{<:}$–that become sparser as input size grows. Consequently, many large generated inputs are discarded before shrinking begins, and the successful first counterexamples are already biased toward smaller valid inputs. The measured shrink distance supports this interpretation: for type-based generators, shrinking reduces the tree edit distance to the ground-truth minimum by a median of only 3–9 edits per workload (BST 4, RBT 3, STLC 7, $F_{<:}$ 9), substantially below the API-based (41–42) and correct-by-construction (18–98) families.

The remaining campaigns are more workload-dependent. For correct-by-construction generation on BST and RBT, QuickCheck's structural shrinker reports counterexamples closer to the LeanCheck minimum than the integrated shrinkers, while the RBT API-based comparison is statistically indistinguishable. On STLC, Falsify reports smaller counterexamples than both QuickCheck and Hedgehog. On $F_{<:}$, QuickCheck reports the closest counterexamples, followed by Falsify and then Hedgehog. These results do not support a blanket claim that either structural or integrated shrinking is always more effective; the generator family and workload both matter. The detailed statistical comparisons are available in Appendix A.1 (Tables A.1–A.4), including the full per-family, per-metric Friedman and post-hoc Wilcoxon tables.[6]

Following up with another performance measurement, this time on shrinking itself. I

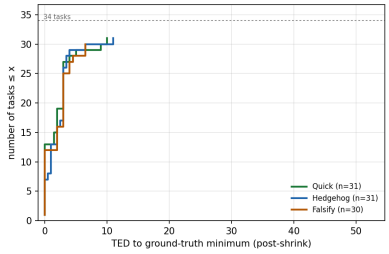
[6]For RBT, LeanCheck found ground-truth minima for only 34 tasks in reasonable time. We exclude the remaining 24 tasks from distance-to-ground-truth comparisons and leave ground-truth-free shrinking evaluation as future work.



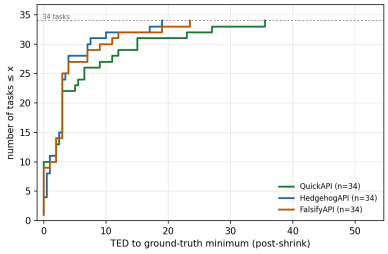
(a) BST, type-based

(b) BST, API

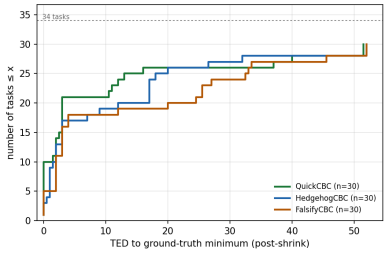
(c) BST, CBC



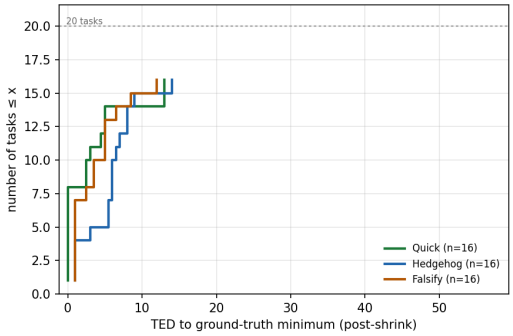
(d) RBT, type-based



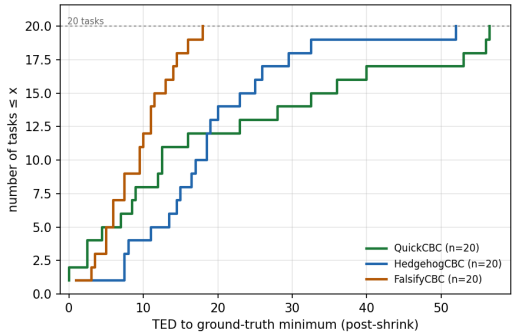
(e) RBT, API



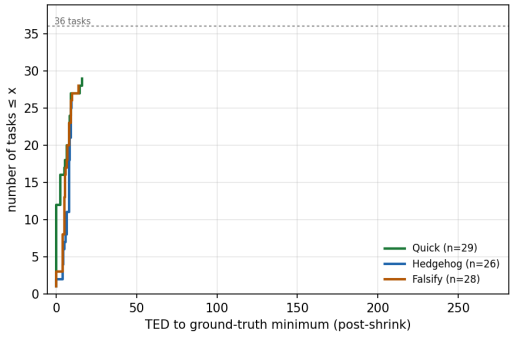
(f) RBT, CBC



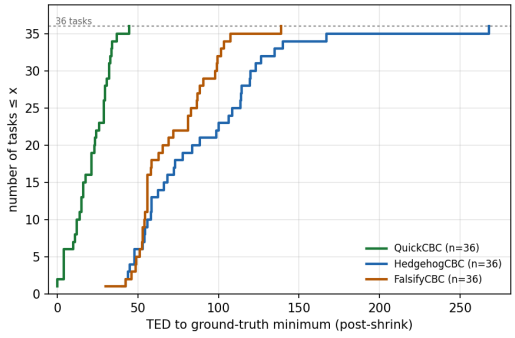
(g) STLC, type-based



(h) STLC, CBC



(i) $F_{<:}$, type-based



(j) $F_{<:}$, CBC

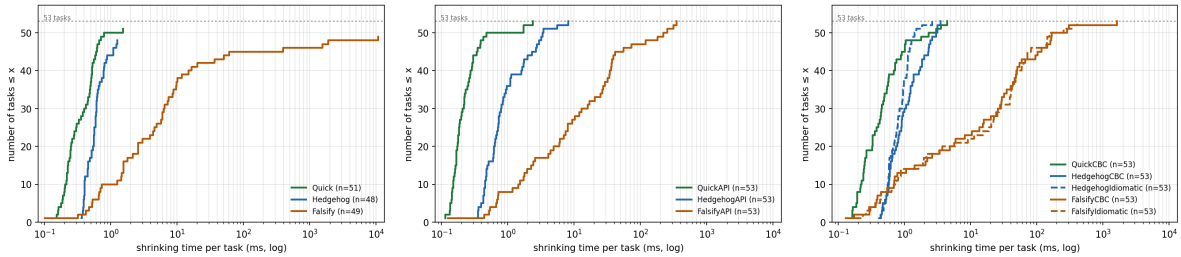
Figure 3.12: CCP of the post-shrink tree edit distance from each strategy's minimal counterexample to the ground-truth minimum found by LeanCheck.

present both absolute shrinking time in Figure 3.13 as well as time per edit distance between the original counterexample and the reported shrinking result in Figure 3.14 to normalize over different counterexamples reported by each library.

For type-based generators, QuickCheck and Hedgehog have similar shrinking times across the four workloads, while Falsify is consistently slower and has a longer tail. For a fraction of tasks, Falsify's shrinking time is several orders of magnitude larger than the others, whereas QuickCheck and Hedgehog exhibit smaller variation across tasks. This pattern is consistent with our *shrinking effort* measurements, i.e., the number of executions used to explore alternatives during shrinking. The libraries expose knobs for tuning shrinking budgets, but the parameters are not directly comparable: QuickCheck's budget bounds total executions, whereas Hedgehog and Falsify bound failing executions. The graphs use each library's default shrinking budget. We also ran no-shrinking (`budget = 0`) and fixed-budget (`budget = 100`) configurations. The no-shrinking runs let us check the bug-finding overhead of enabling shrinking, while the fixed-budget runs were intended to standardize effort. The latter did not achieve comparable effort across libraries because the budget parameters count different events, so we do not draw conclusions from those runs here.

For correct-by-construction generators, RBT puts QuickCheck and Hedgehog close to each other against a slower Falsify. On BST, QuickCheck is significantly faster than both Hedgehog and Falsify. On STLC and $F_{<:}$, the ordering is clearer: QuickCheck is fastest, Hedgehog is next, and Falsify is slowest.

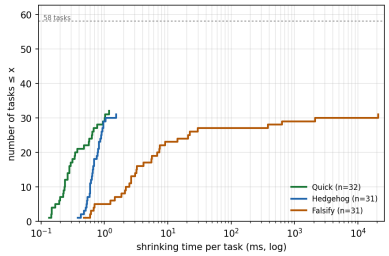
In an attempt to normalize over shrinking effort, we have also measured time per edit distance between the original counterexample against the shrunk counterexample as presented in Figure 3.14. The per-edit results are largely consistent with the absolute compar-



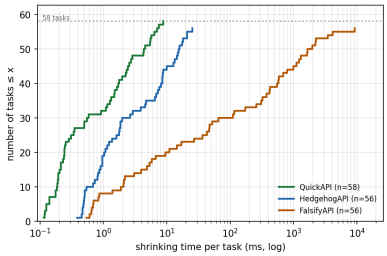
(a) BST, type-based

(b) BST, API

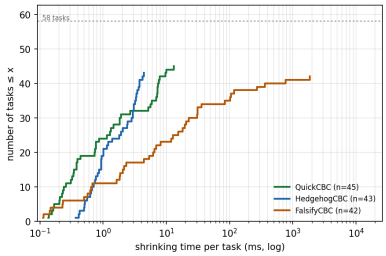
(c) BST, CBC



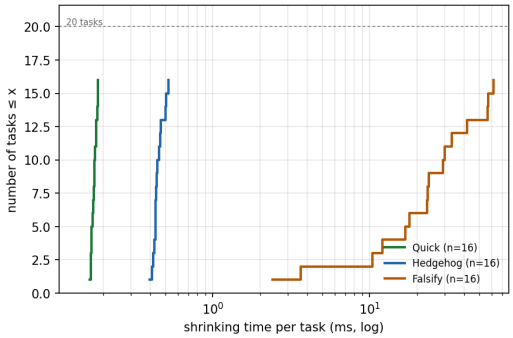
(d) RBT, type-based



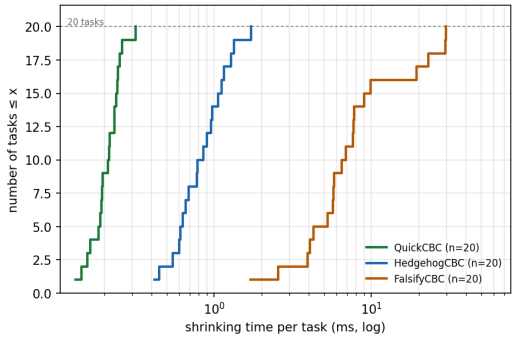
(e) RBT, API



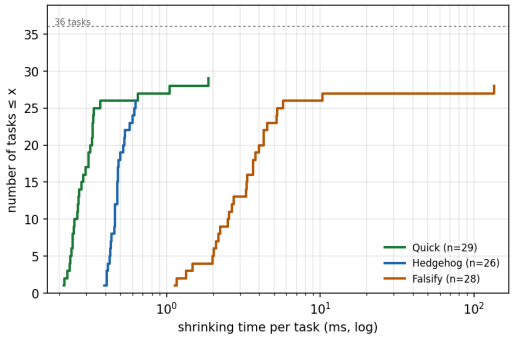
(f) RBT, CBC



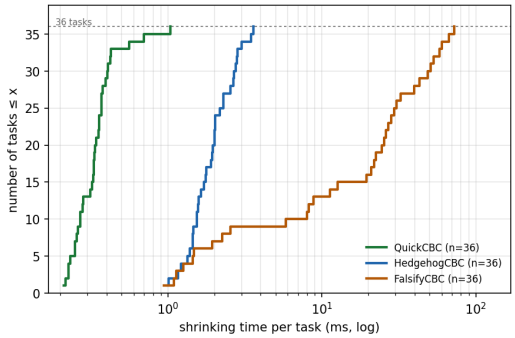
(g) STLC, type-based



(h) STLC, CBC



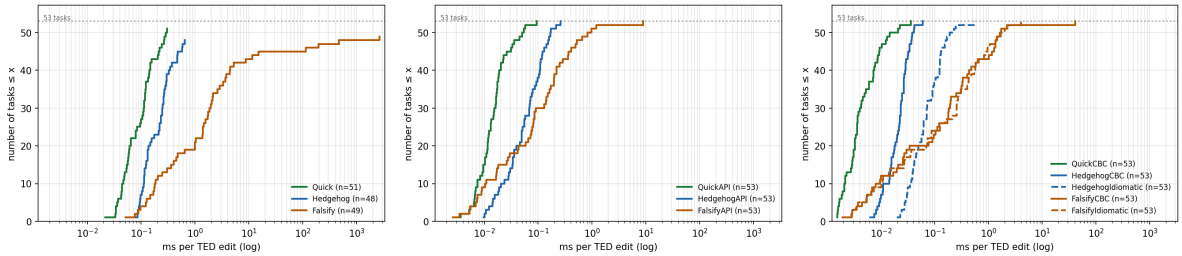
(i) $F_{<:}$, type-based



(j) $F_{<:}$, CBC

Figure 3.13: CCP of per-task shrinking wall-clock time (ms, log scale)

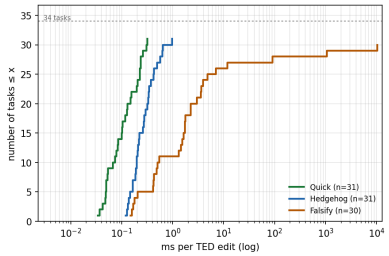
ison for QuickCheck and Hedgehog. For Falsify, the two diverge sharply in the API-based campaigns: its API-based generators start from far larger pre-shrink counterexamples (me-



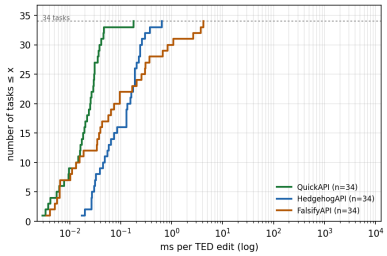
(a) BST, type-based

(b) BST, API

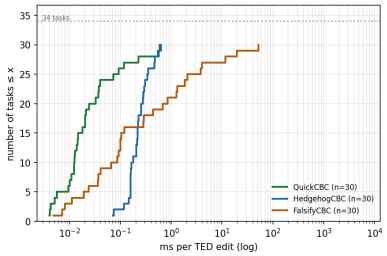
(c) BST, CBC



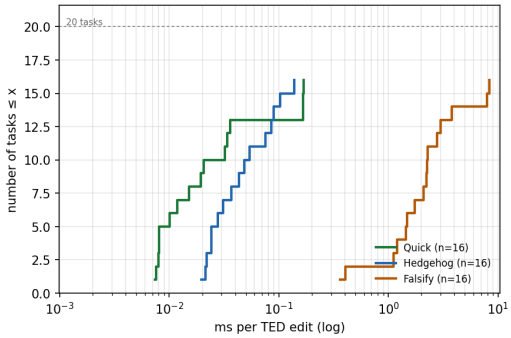
(d) RBT, type-based



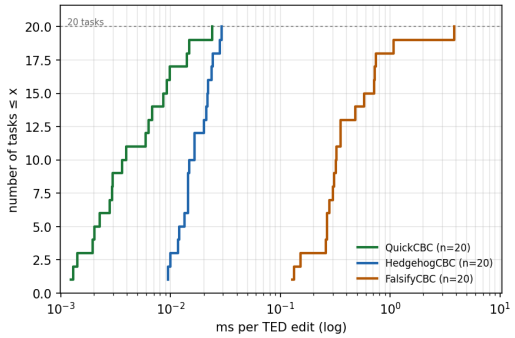
(e) RBT, API



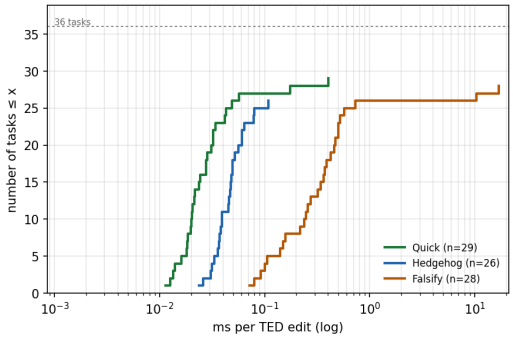
(f) RBT, CBC



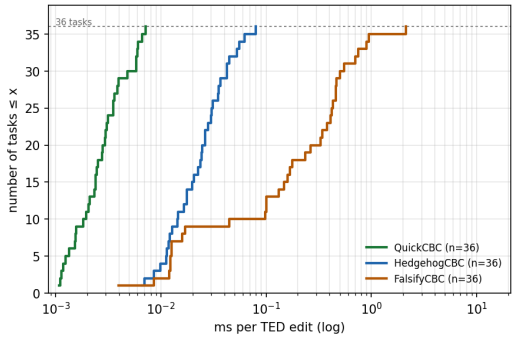
(g) STLC, type-based



(h) STLC, CBC



(i) $F_{<:}$, type-based



(j) $F_{<:}$, CBC

Figure 3.14: CCP of shrinking time per unit of progress: milliseconds spent shrinking divided by the tree edit distance reduced (log scale).

dian pre-shrink TED = 150 vs = 13-18 for the others), so the absolute comparison conflated higher shrinking effort with slower shrinking. Normalizing per edit collapses Falsify's gap to QuickCheck from 49x/95x (BST/RBT) to 6.2x/2.4x. In the type-based and correct-by-construction campaigns, where Falsify does not start from larger counterexamples, the per-edit and absolute comparisons agree.

3.5 Discussion

We now discuss the implications of these results in context of real world PBT usage. Namely, an important shortcoming of the original ETNA experiments is the lack of separation of bug-finding and shrinking time, specifically because bug-finding is a search for an "unknown unknown"; the testing process is searching for a bug that potentially does not exist. Once the testing uncovers a bug, a structure that is inherently finite, and hence has a large, complex but ultimately finite amount of search space; the shrinking is searching for a "known unknown" with an acceptable false negative result, a bug that is not truly minimal, but still a useful starting ground for the user. The implication, if one is to accept this argument, is that it is not acceptable to worsen bug-finding performance at the expense of better shrinking performance, given that it is justifiable to spend more time on a found bug than searching for a bug that might not exist. Our results, combined with these arguments, support a narrower conclusion: on these ETNA workloads, QuickCheck-style structural shrinking remains highly competitive, and often preferable, on the measured axes of bug-finding time, shrinking time, and distance to known minima. This should not be read as a general recommendation against integrated shrinking. Hedgehog and Falsify reduce the need to write separate shrinkers and

can preserve generator invariants by construction; those usability benefits are real, but they are not directly measured by our experiments.

This result must also be taken in context; the representativeness of ETNA workloads in terms of precondition sparsity can significantly affect structural shrinking. Moreover, a large chunk of the experiment results in this section depend on the execution time. ETNA-Haskell workloads are small programs; the large search space for both generation and shrinking indicates their quality in comparing effectiveness of the algorithms for generation and shrinking, but any indications of performance are potentially biased by the domination of the time spent in the PBT library as opposed to executing the tests themselves. As execution times get longer for workloads, the PBT library performance is less critical in the overall execution, whereas the quality of the algorithms will be more important.

An important metric for this discussion is *sample-efficiency*: how many candidates a shrinker must test during the shrinking phase, relative to the progress it makes. A sample-efficient shrinker should scale better as property execution time grows, because it spends fewer executions on candidates that do not reproduce the failure. One argument for integrated shrinking is that it should improve sample-efficiency for properties with preconditions, because generator invariants are preserved by construction. Our measurements partly support this argument for Hedgehog: for correct-by-construction generators, Hedgehog preserves a 26–56% failure rate across shrinking steps, compared to 5–12% for QuickCheck and roughly 2–3% for Falsify. The stark contrast between Hedgehog and Falsify is important: integrated shrinking can avoid invalid candidates introduced by structural shrinking, but integrated shrinking alone does not guarantee high sample-efficiency. Moreover, it is only one part of the trade-off; in these workloads, Hedgehog's higher failure rate during shrinking does

not translate into better shrinking time or consistently better final counterexamples.

Limitations and Threats to Validity

As we worked on our measurements, we have seen that presenting concrete measurements that provide insights is a genuinely complex task. The generators we use are based on the original QuickCheck generators written for ETNA, and are therefore likely to favor QuickCheck. We attempted to resolve this possibility by going through Hedgehog and Falsify repositories to find idiomatic alternatives, which resulted in the idiomatic CBC generators for BST that we report on.

Measuring performance in Haskell requires forcing strictness in the runners, which might result in scenarios where our measurements do not completely agree with empirical real world usage without the forced strictness. The whole experimentation suite is available as a reproducible benchmark in order to allow for outside contribution to correct any measurement errors that might have been the result of an oversight on the measurements. It is also very hard to control for effort across libraries, we simply could not set constant shrinking effort across libraries, and therefore opted for the default configurations for our final experiments.

Lastly, tree edit distance to ground truth minimums can be a noisy metric. For instance, it treats symmetric counterexamples (A-B vs B-A) differently, when they might in fact be equivalent. The minimum also relies on the exhaustive search algorithm, a different enumeration strategy might (and does) reach different inputs. As such, we also report on counterexample sizes in the statistical tests in Appendix A.1 which we observed resulted in broadly similar results to the edit distance experiments. Additionally, one might argue that the edit distance

metric may be closer to the structural notion of size exposed by QuickCheck-style shrinkers than to the randomness-buffer or choice-tree search spaces used by integrated shrinkers. The ultimate measure of shrinking effectiveness is debugging time, which would require a comprehensive user study that we deemed way out of scope for this paper.

3.6 Related and Future Work

ETNA, as I mentioned in the beginning of the chapter, is a reference to every crossword-puzzler's favorite Italian volcano. The design was heavily inspired by two existing benchmark suites in the fuzzing space: LAVA Lava and Magma Hazimeh:2020:Magma. 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 FixReverter offered a middle ground, generalizing real bug-fixes into patterns and applying them to multiple locations in a program.

Since we published ETNA, a number of studies on PBT have used it for evaluation. "Fail Faster" [98] has used the OCaml workloads for optimizing OCaml PBT libraries, Bennet [99] has extended ETNA for C for evaluation of random testing for separation logic, "Testing Theorems" [100] and "Tuning Random Generators" [101] use Rocq workloads, "QuickerCheck" [57] uses the Haskell workloads. In addition to works presented in this thesis, I have also worked with Ivan Mladenov on "QuickerChick", his student research competition submission on optimizing QuickChick.

As these studies were performed, we've kept working on both making ETNA more us-

able by doing the required engineering, and also getting feedback from the people using ETNA. I have also spent time horizontally growing ETNA as much as possible, porting the existing workloads in Haskell and Rocq to OCaml, Racket, Rust, and Python. There are a number of future directions I see ETNA going forward.

The first is improving presentation of results and analysis. The statistical tests and cumulative count graphs in the shrinking evaluation are one such step, the detailed HTML report produced by `etna report` is another avenue. An important missing aspect is the ability to do in-depth analysis by providing information at multiple granularities. Ideally, ETNA should be directly integrated with a Tyche [102] style visualization framework. We are yet to integrate code coverage analysis in any shape, which can be especially useful when we start evaluating more fuzzing tools.

Another is a breadth-oriented study for horizontally growing ETNA. Large language models (LLMs) show promising results for automated bug injection from historical bug fixes. Several studies [103–106] have investigated using LLMs to generate properties or propose bugs, but reinjection of historical bugs is left unexplored.

The closest related work in terms of PBT shrinking is the study of the Hypothesis reducer [16]. That work evaluates test-case reduction using case studies, mean input size, and the number of executions in the shrinking step. We build on that perspective by measuring distance to known minima, separating bug-finding time from shrinking time, and comparing several Haskell libraries and generator families on the same ETNA workloads.

Shrinking is part of a broader family of techniques that iteratively simplify an input or program while preserving a property of interest. In *test-case reduction*, the preserved property is a triggering failure: delta debugging introduced iterative simplification of failing in-

puts [10], followed by optimized techniques leveraging the hierarchical structure of the test case [107], the formal syntax [108], or the tree structure [109]. Researchers have also proposed probabilistic [110] and weighted [111] alternatives to delta-debugging, as well as domain-specific reducers for compiler testing [27,112], SQL queries [113] and dependency graphs [114]. Related lines simplify programs under different objectives: *program trimming* preserves equi-safety to scale static analyzers [115], while *program and container debloating* preserves intended functionality to reduce attack surface [116, 117]. Across these settings, evaluation typically reports size reduction and reduction time on benchmark suites of recorded failures or programs. Future work should broaden the benchmark suite beyond ETNA's Haskell workloads, and perhaps investigate alternative algorithms for shrinking.

Chapter

4 Domain-Specific Property-Based Testing

After the initial chapters on a quest for standardization, this chapter investigates a domain-specific PBT scenario that falls out of the standardization framework I have worked so far. Different domains require different tools and building blocks, many of which are possibly collapsible under a single framework; the greater question is the effectiveness of such a solution rather than its feasibility. This chapter talks about a particular instantiation of stateful property-based testing for testing databases, describing a paradigm we proposed as part of our DBTest 2026 paper "DIRT: Database-Integrated Random Testing" [19] on testing Turso [118], a SQLite-compatible open source database.

I started building a domain-specific property-based testing framework within Turso in the early days of its development. Ethan Chou's master's thesis work on model-free random testing [47] was closely related to the problems I was solving, so we started working together. He implemented parts of the random generator and the shrinker, and has been instrumental in the evaluations. We formalized the technique and conceptualized the paradigm with Harry Golstein and Leo Lampropoulos. Turso has been in active development as we continued to work on the project, and many Turso contributors, including but not limited to Diego Reis, Pedro Muniz, Mikaël Francoeur and Pavan Nambi have implemented parts of the random testing infrastructure; however, they did not work on the paper itself.

4.1 DIRT: Database-Integrated Random Testing

Database Management Systems (DBMSs) are complex systems, and inevitably, such complexity leads to bugs. To discover bugs, DBMSs have historically used random testing, with approaches ranging from binary fuzzing with AFL [4] to structured query generation in SQLSmith [119], and to the powerful test oracles of SQLancer [34]. Within prior work, SQLancer is distinguished by using specialized test oracles that reveal logic bugs in DBMSs, and it has proven extremely successful at finding bugs in mature databases.

Unfortunately, off-the-shelf testing frameworks are not as effective at finding actionable bugs in DBMSs that are actively being developed and have many missing features. When the space of randomly generated inputs is larger than the system under test supports, many of the generated tests lie in the unimplemented portion of the system, registering as false positives. Naturally, such false counterexamples are of no help to developers of DBMSs.

One way to alleviate this issue is to tailor the testing framework to the DBMS under test, ensuring that the testing framework only tests implemented features. Unfortunately, existing frameworks are not built with this flexibility in mind. In an off-the-shelf tester, such tailoring would require developers of the DBMS to dive into the internals of an external project, modify it to suit their needs, and maintain those changes as the DBMS evolves.

In our work, we proposed DIRT, *database-integrated random testing*, an alternative testing paradigm for testing databases during development. DIRT embeds the testing tool within the database itself, allowing for the generators to naturally evolve alongside the DBMS, therefore avoiding false positives by construction. At the same time, we leveraged this tight integration to empower developers to write their own oracles by developing a domain-specific language

(DSL) for expressing properties customized to particular features being developed.

We studied this problem in the context of Turso, an open-source SQLite3-compatible OLTP database engine in active development. With an average of more than 30 commits a day at the time of our study, Turso was a rapidly evolving target for testing. DIRT helped find many different bugs in the database; we have confirmed and studied 23 of these bugs. In comparison, running SQLancer-SQLite3 with minimal modifications in Turso led to a false positive rate of 96.5% with only one new bug, and running SQLancer-Turso, an existing independently developed SQLancer integration for Turso, without modifications led to a false positive rate of 58% with six newly discovered bugs.

4.1.1 Fuzzing and Property-Based Testing

In Fuzz Testing [3], the primary focus is on smart and effective test-case generation that can work out of the box with minimal user inputs. The key idea is to leverage *runtime feedback*, usually in the form of branch coverage [4,120], though not always [8], to keep track of inputs that exhibit interesting behavior (e.g. uncovered new paths) and then mutate them in the hopes of discovering even more interesting inputs. The most common oracle for deciding if an input is a bug or not is simple: does the program crash? Other oracles include fuzzing against a model or a different implementation (known as differential fuzzing [121,122]) or checking for excessive execution times [33].

Property-based testing (PBT), on the other hand, primarily focuses on empowering users to write their own oracles [85,123]. PBT frameworks usually offer DSLs for expressing oracles in the form of universally quantified executable predicates, as well as infrastructure for writing *generators* - programs that generate test inputs to test such predicates [85,124]. Al-

though traditionally these communities have been mostly distinct, the lines between them are increasingly blurred. In fact, fuzzing can be viewed as an instance of PBT where the property is fixed (e.g. the program does not crash) and PBT can be improved by extending its generation with coverage-guided capabilities. Such approaches have been tried recently with great success [12, 29], and in the context of database testing, both SQLancer and DIRT follow such a hybrid viewpoint.

4.1.2 SQLancer

SQLancer [34] is a multi-year research project by Rigger et al. that set the bar for automated random testing of databases since its inception in 2020. It hosts an extensible core with adapters for many production DBMSs including but not limited to ClickHouse, Apache Datafusion, MySQL, PostgreSQL, and SQLite. Within the last five years, SQLancer has not only grown with respect to the breadth of databases it supports, but it has also widened its arsenal of oracles. It started with Pivoted Query Synthesis (PQS) [44], a rather "simple" containment property over databases that has found at least 121 unique logic bugs in production databases. Two metamorphic oracles followed: Non-Optimizing Reference Engine Construction (NoREC) [45], which found 51 optimization bugs, and Ternary Logic Partitioning (TLP) [46], which discovered 77 novel logic bugs. SQLancer currently supports Query Plan Guidance (QPG) [125] for feedback-guided generation, Cardinality Estimation Restriction Testing (CERT) [126] for finding performance bugs in DBMSs, Differential Query Plans (DQP) [127] for detecting bugs in join optimizations, and Constant Optimization Driven Database System Testing (CODDTest) [128] for finding logic bugs in optimizations. As SQLancer focuses on testing large classes of behaviors across a variety of databases, each oracle amounts

to a significant research contribution in a new research paper.

Integrating a new DBMS into SQLancer is a time-consuming process. At a minimum, SQLancer integration requires implementing AST connectors, generation APIs for the relevant queries, and the oracles to use for detecting bugs. SQLancer also has a notion of *expected errors*, bugs that the users can deem as expected, so SQLancer does not report them as errors and continues testing.

4.1.3 Specifying Correctness Oracles

Rather than expecting database developers to modify SQLancer to suit their particular needs, we instead wanted to offer them flexible and extensible abstractions so that they can test their database throughout its development. DIRT was created to provide such random testing infrastructure that evolves alongside the DBMS and produces actionable bug reports for the database developers. This evolution involves not only tailoring the input space of SQL generation to the currently implemented portion of Turso for reducing false positives, but also tailoring the correctness oracles to new features as they are added, decreasing false negatives.

In this section, I introduce *generation actions*, an imperative formulation for properties that do not just describe *what* the property tests, but *how* to test it. We begin by using a very simple commutativity property to convey the basic ideas and notation, then detail the oracles that were actually developed to test Turso, and conclude the section with a description of our generation strategy. For concreteness, let us consider the following hypothetical equivalence relation leveraging the commutativity of $\wedge$: for any database *db* and any two predicates p and q that can range over variables from the database, if we evaluate $\mathsf{SELECT}(p \wedge q)$ and $\mathsf{SELECT}(q \wedge p)$ in *db* they should yield equivalent results:

$$\forall db, p, q. \quad \text{variables}(p) \subseteq db \wedge \text{variables}(q) \subseteq db$$
$$\Rightarrow \quad \text{SELECT}(p \wedge q) \equiv \text{SELECT}(q \wedge p)$$

Testing such a property would entail generating an arbitrary database, two expressions p and q that only mention variables from that database, and then evaluating it repeatedly. That is, we have precisely described *what* we want to test, the exact conditions in which the test is valid, but left the *how* to a database-agnostic framework. Instead of quantifying over db, p, and q and constraining them with post hoc relations, we propose a simple abstraction for describing *how* db, p, and q are generated with respect to the constraints. We call this abstraction for describing properties *Generation Actions* (*GA*).

```
gen property db =
    t ⟵ pick db.tables
    c ⟵ pick t.columns
    v ⟵ genOf expression c.type
    p := t.c = v
    q ⟵ gen expression (t,c)
    ! r1 := SELECT (p AND q)
    ! r2 := SELECT (q AND p)
    ! assert(r1 == r2)
```

Each generation action is parameterized by the type it returns and the context it can use (implemented as a trait in Rust). The snippet I present above expresses the same high-level commutativity property as a generation action that returns a `property` parameterized by a context that contains a database `db`. Rather than independently generating p and q, we fix some of the details of generation. We `pick` an arbitrary table from the database, `pick` a column from that table, and generate an expression `v` of that column's type, before constructing an explicit equality check `t.c = v`. For `q`, we generate an arbitrary `expression` that can refer to both `t` and `c`. We use ⟵ to bind the results of generation and := for regular let-style binding. We express

interactions with the database, explicitly annotated with a (!). In this example, the only interactions are defining the two different ways of conjuncting p and q and then `assert`ing their equality, but interactions can generally be queries or assertions.

An interaction not shown in this example that we used in our oracles is the ability to inject simulated *faults*. DBMSs interact heavily with the underlying OS, file system, and network, all of which have unpredictable and chaotic failure modes. As such, the same set of interactions (e.g. `CREATE-INSERT-DELETE`) might result in different results depending on any invisible failures in the middle. FoundationDB [129] is heavily praised for its use of simulation testing, injecting simulated faults within an otherwise correctly working testing environment. We also introduced fault injection in DIRT as part of the DSL for generation actions, as exemplified by the header initialization bug studied in §4.1.5.1.

4.1.3.1 Definitions of Oracles in DIRT

In this subsection, I present five oracles written as sequences of generation actions. Three of them are adaptations of SQLancer oracles: Pivoted Query Synthesis (PQS), Non-Optimizing Reference Engine Construction (NoREC), and Ternary Logic Partitioning (TLP). I first define PQS [44] as a universally quantified proposition in Figure 4.1, then show its implementation as a generation action in Figure 4.2. PQS states that, given a set of tables in the database, `SELECT`ing for a row constructed from the contents of those tables should contain that row.

Figure 4.1 and Figure 4.2 show two formulations of PQS, Figure 4.1 is the propositional formulation with universally quantified variables that define *what* PQS is, and Figure 4.2 is a GA formulation of PQS with two tables/columns as generation actions that define *how* PQS is

$$
\begin{aligned}
&db : \text{Database} \\
&t_1, t_2, \ldots, t_n : \text{Table} \\
&c_1, c_2, \ldots, c_n : \text{Column} \\
&p_1, p_2, \ldots, p_n : \text{Expression} \\
&r : \text{Row} \\
&\forall db, t_1, \ldots, t_n, c_1, \ldots, c_n, p_1, \ldots, p_n, r. \\
&t_1, \ldots, t_n \in db \\
&\land c_1 \in t_1 \land c_2 \in t_2 \land \cdots \land c_n \in t_n \\
&\land r.c_1 \in t_1.c_1 \land r.c_2 \in t_2.c_2 \land \cdots \land r.c_n \in t_n.c_n \\
&\land p_1(r) = \text{TRUE} \land p_2(r) = \text{TRUE} \land \cdots \land p_n(r) = \text{TRUE} \\
&\Rightarrow r \in \text{SELECT } * \text{ FROM } t_1.c_1, \ldots, t_n.c_n \text{ WHERE } p_1 \land \cdots \land p_n
\end{aligned}
$$

Figure 4.1: Pivoted Query Synthesis as a universally quantified property.

```
gen property db:
    (t1, t2) ⟵ (pick db.tables, pick db.tables)
    (c1, c2) ⟵ (pick t1.columns, pick t2.columns)
    (r1, r2) ⟵ (gen row t1, gen row t2)
    r  := (r1.c1, r2.c2)

    ! INSERT INTO t1 VALUES r1
    ! INSERT INTO t2 VALUES r2

    p1 ⟵ gen expression (t1, r1)
    p2 ⟵ gen expression (t2, r2)

    ! RS := SELECT r FROM t1, t2 WHERE p1 AND p2
    ! assert(r in RS)
```

Figure 4.2: Pivoted Query Synthesis as a generation action.

tested, modeling the implementation of PQS that tests Turso today. The propositional formulation quantifies over a database and a sequence of tables, columns, and expressions, under the constraint that the expressions will return `TRUE` when tested against a row `r`. The corresponding GA closely follows along, but ensures these constraints are satisfied by construction. We start by picking tables `t1, t2` from the state `db.tables`, followed by picking columns from the respective table. `gen row t` and `gen expression (t, r)` are dependent generation primitives, where the former generates a row based on the table `t`, and the latter generates an expression

that will evaluate to `TRUE` for row `r` of the table `t`.

In Figure 4.3, I provide definitions of SQLancer oracles as GAs in addition to the other oracles we implemented for DIRT. Figure 4.3a demonstrates how to write Non-Optimizing Reference Engine Construction Generation (NoREC) [45] as a GA. Figure 4.3b presents the GA for `WHERE Extended` case of Ternary Logic Partitioning (TLP) [46] oracle from SQLancer. We can express other properties that are not present in SQLancer, including those fundamental to key-value stores such as *Deleted rows should not be in the table* presented in Figure 4.3c. Some generated properties require additional validity conditions. For example, Figure 4.3d composes two generated `SELECT` queries with `UNION ALL`; this composition is only valid when both queries return the same number of columns, so the generation action includes an explicit `assume`.

4.1.4 Query Generation

We designed our query generation algorithm with three goals in mind. First, we wanted each automatically generated database interaction to respect database state (e.g. we should not select from a table that does not exist unless a user-written generation action explicitly calls for it). Second, we did not want to rely on querying the database itself to obtain the information necessary to ensure the first goal—doing so would assume that those queries ran successfully, defeating the purpose of testing. Finally, we wanted to allow database developers to specify, in a lightweight manner, aspects of the distribution of generated interactions that they deemed important, such as read/write heavy queries. To that end, we follow a generation-by-execution style approach [130], in which a model of the database as a key-value store is updated during generation as a shadow state. Unlike traditional model-based

```
gen property db:
    t ⟵ pick db.tables
    r ⟵ pick t.rows
    p ⟵ gen expression (t, r)
    ! RS1 = SELECT * FROM t WHERE p
    ! RS2 = SELECT p FROM t
    ! assert(RS1.length() == RS2.count(1))
```

(a) Non-Optimizing Reference Engine Construction (NoREC)

```
gen property db:
    t ⟵ pick db.tables
    p ⟵ gen expression (t)
    p' ⟵ gen expression
    ! RS1 = SELECT * FROM t WHERE p
    ! RS2 = SELECT * FROM t WHERE p AND p' UNION ALL
    SELECT * FROM t WHERE p AND (NOT p') UNION ALL
    SELECT * FROM t WHERE p AND (p' is NULL)
    ! assert(RS1.length() == RS2.count(1))
```

(b) Ternary Logic Partitioning (TLP) WHERE Extended

```
gen property db:
    t ⟵ pick db.tables
    r ⟵ pick t.rows
    p ⟵ gen expression (t, r)
    ! DELETE * FROM t WHERE P
    ! RS = SELECT * FROM t WHERE p
    ! assert(r not in RS)
```

(c) Deleted rows should not be in the table

```
gen property db:
    s1 ⟵ gen SELECT db.tables
    s2 ⟵ gen SELECT db.tables
    ! RS3 = s1 UNION ALL s2
    ! assume(s1.first().length() == s2.first().length())
    ! assert(s1.length() + s2.length() == RS3.length())
```

(d) UNION ALL preserves cardinality

Figure 4.3: Definitions of Oracles as Generation Actions

properties [123], this model is not used for differential testing, only for keeping track of the relevant information for correct generation. Algorithm 1 sketches our approach.

Algorithm 1 DIRT Generation Algorithm

```
e ← {read : R, write : W, create : C}          ▷ Workload distribution
interactions ← []
st ← {read : 0, write : 0, create : 0, tables : []}
while N > 0 do
    i ← gen Interaction(st, e)
    i.shadow(state)                            ▷ Update the shadow state
    interactions.push(i)
    N ← N − 1
end while
```

Database developers can specify three parameters, R, W, and C, describing the proportion of read, write, and create instructions that should appear in the generated interactions. The algorithm iteratively generates one or more interactions compatible with the current shadow state, updates the state to account for the new interactions, and repeats until enough interactions have been generated.

SQLancer does not keep a separate state [44], but instead uses the database APIs for querying the current state, such as the table names in `sqlite_master`, due to the implementation effort for the shadow model. We opted for the shadow state because it allows for complex reasoning over the database state for constructing arbitrary queries and properties, and because it gives us a canonical property over the database state: the shadow state is identical to the database at any given moment.

4.1.5 Evaluation

In order to evaluate the effectiveness of DIRT, we explored the answers to the following questions:

- RQ1: Does DIRT find bugs in Turso?
- RQ2: How does DIRT compare to the state-of-the-art?

4.1.5.1 RQ1: Does DIRT find bugs in Turso?

Bug	Description	Oracle	Module
466	TRUE not accepted as catch-all predicate	No Error	Query Compiler
548	Infinite loop when checkpointing on Linux	No Infinite Loop	I/O Subsystem
629	Query Optimizer broke TRUE in predicates	No Panic	Query Optimizer
662	SELECT with nested Boolean expressions sometimes gave no results	PQS	Query Optimizer
681	Storage Engine (B-tree) insert caused subtract with overflow	No Panic	Storage Engine
682	Faulty recursive binop logic caused SELECT to miss rows	PQS	Query Compiler
924	Storage Engine (B-tree) balancing caused page corruption when deleting	No Panic	Storage Engine
1040	LIKE operator did not work for non-text values	No Panic	Query Executor
1203	Storage Engine (B-tree) balancing error	No Panic	Storage Engine
1629	Storage Engine (B-tree) cell updates caused infinite loop in UPDATE	No Infinite Loop	Storage Engine
1734	DELETE did not emit conditional jumps if WHERE term was constant	Delete-Select	Query Compiler
1815	Storage Engine (B-tree) failed to balance when insert caused cell overflow	No Panic	Storage Engine
1818	Always read DB header and schema from file instead of memory page	No Panic	Page Manager
1975	Storage Engine (B-tree) expected table or index leaf page	No Panic	Storage Engine
1991	Use after free when validating B-tree balance	No Panic	Storage Engine
2024	SELECT ... LIMIT resulted in different rows from SQLite	Differential	Query Executor
2026	UPDATE then SELECT resulted in different rows from SQLite	Differential	Query Executor
2047	Overflow cell with divider cell was not found due to faulty validation	No Panic	Storage Engine
2074	SELECT hung with long text, CacheFull error	No Panic	Page Manager
2075	Large table with 128 columns handled incorrectly	No Panic	Page Manager
2088	Incorrect record header size calculation	No Panic	Page Manager
2106	Interior node replacement caused self-reference when depth exceeded 2	No Panic	Storage Engine
2116	Advance after post-delete balancing did not advance B-tree cursor	No Panic	Storage Engine

Table 4.1: List of confirmed unique Turso bugs found and reported by DIRT

We have collected 23 confirmed and fixed unique bugs found by DIRT in Turso over the course its development, evolving in complexity as Turso grows. Table 4.1 is the list of confirmed bugs. The bugs range from simple parser level bugs that could be classified as minor problems, to severe logic bugs deep in the core functionality of the database that could result in data loss. We have chosen three representative bugs as case studies, detailing the conditions in which they were discovered and triggered. The cases demonstrate the diversity of bugs found in Turso with respect to their origin (the bytecode compiler, the B-tree index, and the database header), to how they were discovered and reported (automated report, ourselves, and the developers), and to the oracles used (simple deletion properties, large indexed table generation, and developer-written properties).

Case 1: DELETE not emitting constant Where terms.

The first bug lies in the compiler of Turso. Turso translates SQL statements into bytecode and then runs that bytecode in a virtual machine (this is the same approach that SQLite uses [131]). The bytecode compilation is error-prone, as small changes to the compiled SQL expression can greatly affect the generated bytecode.

The bug stemmed from an error in the bytecode compilation of `DELETE`, specifically in the case of constant expressions in the `WHERE` terms. Constant expressions can be compiled to run before the main execution loop of the query, but instead the compilation step just skipped them. This caused `DELETE` with a constant that evaluates to `FALSE` to be incorrectly executed.

This bug was automatically reported by DIRT running in Turso CI and fixed within a week. The bug was discovered as a result of our PQS implementation in Turso augmented with validity preserving queries between the `INSERT` and `SELECT` statements. The random tester

added a naive `DELETE` operation that should not have affected the results of the `SELECT`, keeping the containment property intact, but the assertion that the inserted row should be in the result of the `SELECT` failed, triggering the bug report.

Case 2: Interior node replacement caused self-reference when depth exceeded 2.

The second bug manifested in the balancing logic of the B-tree implementation of Turso, but only because the generated interactions were paced to match the state of the Turso development. When Turso added database indexes as an experimental feature, we extended generation to support statements such as `SELECT DISTINCT`, `CREATE INDEX`, and compound operators such as `UNION` or `UNION ALL` to the space of generated inputs, and started generating larger tables to take advantage of the indexes. This resulted in triggering a crash failure in the core B-tree data structure that could result in data loss or corruption if not fixed. We reported this bug to the Turso developers, and the bug has since been fixed.

Case 3: Database Header Initialization.

The third selected bug highlights the importance of considering fault scenarios. SQLite uses write-ahead logs (WAL) in order to provide non-blocking reads and ACID transactions in the database. This strategy incurs an additional complexity to the persistent state of the DBMS as the database file might be out-of-date for certain operations, requiring the DBMS to read such information from WAL. This bug is the result of such a situation, where metadata like database size and schema is out-of-date at the time of reading, leading to incorrect operations. DIRT found this bug through an assertion failure in the freelist structure in the database header. As a result, the developer who reported the original bug wrote the first example of a *regression property*, added a new fault primitive `ReopenDatabase` that closes down

existing connections with the database and reopens them later, enabling the detection of any future bugs that might be a result of an error in the persistent state logic of Turso.

4.1.5.2 RQ2: How does DIRT compare to SQLancer?

We answered RQ2 by comparing the rate of true-positive and false-positive reports from three different random testing configurations: DIRT, the default SQLite3 integration of SQLancer with pragmas disabled (denoted SQLancer-SQLite), and the fork of a work-in-progress Turso integration of SQLancer by a Turso contributor (denoted SQLancer-Turso). Pragmas had to be removed for SQLancer-SQLite because most of them were unimplemented in Turso, causing all reported bugs to be false positives in our testing. We defined an actionable bug report from the tools as a sequence of SQL statements that *would have been* submitted and confirmed as a bug at the time of its original report. We used the 23 bug reports we analyzed as the ground truth, finding commits which are known to have bugs. The bugs span over 20 Turso commits, 6 of which do not provide Java bindings necessary for connecting with SQLancer. We ran all three tools on the remaining 14 commits in the commit history. We analyzed the results of each run and separated them into three bins: *True Positives* that would have been submitted and confirmed as bugs, *False Positives* that are reported as failures by the tools but do not constitute bugs as they are unimplemented features, and *No Bugs* that did not report a failure, provided in the stacked bar chart in Figure 4.4. SQLancer-SQLite achieves a true positive rate below 10% across all 14 commits. SQLancer-Turso performs better, with a 42% true positive rate, but still yields many more false positives than DIRT (58% vs. under 1%). DIRT also finds substantially more actionable bugs: 23 unique bugs, compared with six found by the two SQLancer configurations combined (§4.1.7). This does not mean that DIRT has bet-

ter generators or oracles than SQLancer. Rather, it shows that for rapidly evolving databases with many missing features, database-integrated testing yields more actionable bugs.

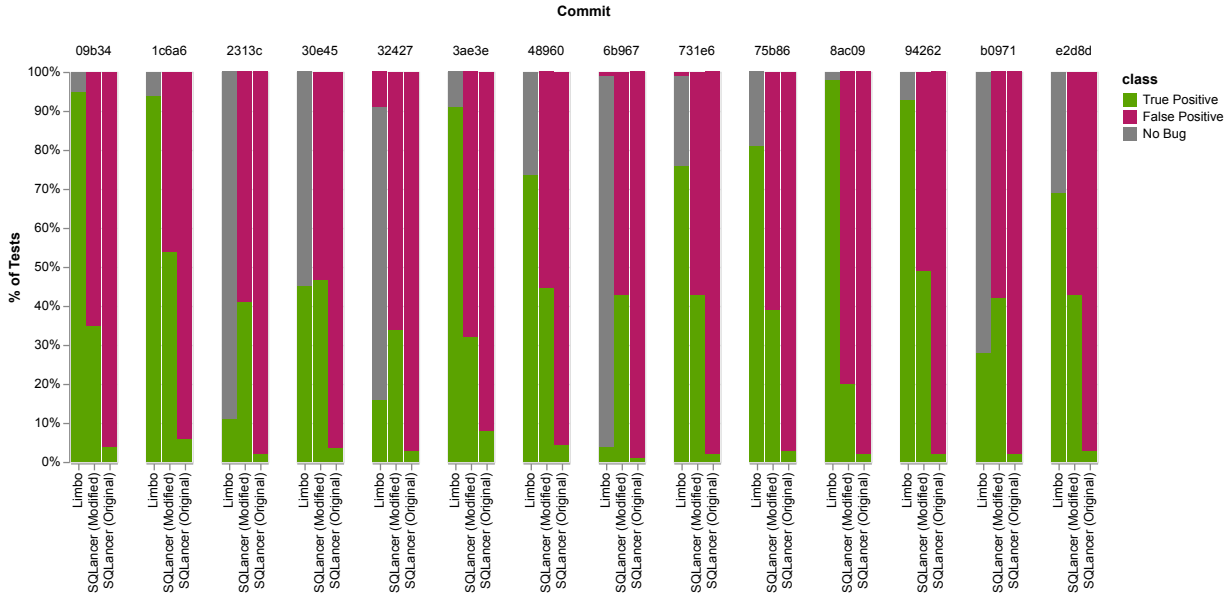


Figure 4.4: Comparison of true positives, false positives, and no-bug outcomes for DIRT, SQLancer-SQLite, and SQLancer-Turso.

4.1.6 Discussion

Why not integrate SQLancer into the development process instead of developing a random testing framework from scratch? As I discussed earlier, SQLancer is not designed for the constant evolution of the random testing infrastructure along with the project. It is a project with its own trajectory, development, new algorithms, generators, and oracles. DIRT aims to empower the database developers themselves by giving them a mechanism to test the database, instead of giving them bugs to solve. There are additional practical barriers: Java bindings for Turso are not complete at the moment, so we had to fix bugs in the SQLancer-Turso integration implemented by one of the Turso developers that relied on incomplete features that silently failed. Panics in Turso caused SQLancer to terminate, so it was not possible to minimize the

counterexamples with SQLancer reducers without further changes.

Given that SQLancer supports expected errors for reducing false positives, could we use expected errors for known bugs and false positives we found to discover more bugs? We could use expected errors, and we would have discovered more bugs in the process. In terms of any static evaluation target, tweaking SQLancer by progressively finding more bugs as found ones are marked as expected, it is always possible to make SQLancer find more bugs. The point, however, is that constant modification is not the expected and supported mode of operation when using SQLancer. Although it is phenomenal for off-the-shelf usage, its capabilities unfortunately act as a disadvantage against its use in actively developed projects with many missing features. We have shown that a project with a much smaller scope can adopt the ideas of input generation and oracles in SQLancer as well as other related work on database testing such as Apollo [122] or Thanos [132] and demonstrated that database developers can turn their domain expertise into writing properties as presented in the third case study in §4.1.5.1. We also observed that many crash failures are the result of inline assertions in Turso. These crashes imply logic bugs as they contradict invariants specified by the developers, blurring the distinction between crash failures and logic bugs in the literature. Such assertions do not replace SQLancer oracles or Turso properties because they do not have the ability to follow values through execution in a holistic way as properties do, so they can only reason about local invariants.

Role in Turso. At the time we started working on Turso, it had a small, unstructured random testing infrastructure that overwhelmingly focused on being able to support Deterministic Simulation Testing [129]. I/O was implemented in a way that could be easily simulated, which allowed for FAULTs to be integrated in the generated interactions. We contributed to the

project as external open source contributors, gradually proposing improvements to random testing infrastructure such as the Generation Actions DSL, most of the existing properties, stateful random generation, shrinking, additional oracles such as differential testing against SQLite and determinism checking, and implementing such proposals. The infrastructure has grown outside of our control at times, producing regressions through the development, along with many improvements by the maintainers and contributors to the project that contributed to the list of bugs found in Table 4.1. The SQLancer-Turso integration was almost entirely developed by Diego Reis, which we have used in our evaluations with small changes to their code.

Related work on adaptive DBMS testing.

Although we compared DIRT mainly against SQLancer as the basis of our oracles, SQLRight, Griffin, and SQLancer++ also adapt testing to DBMS behavior [133–135]. These systems primarily approach adaptation from the perspective of an external testing framework, with evaluations centered on broad effectiveness across multiple databases and minimal manual intervention. In contrast, DIRT studies adaptation as a database-integrated process: the testing infrastructure evolves together with a single DBMS under development, and developers directly shape generators, properties, and fault models as the implementation changes.

4.1.7 Bugs Found by SQLancer

- `UPDATE t SET (c0, c0)=(0, 0)`

  This expression results in a "Column specified more than once" error that was fixed soon after we reported it.

- `SELECT (0x0)`

  There was a parsing error for hexadecimals that resulted in an "invalid float literal" error that had been fixed before our experiments with SQLancer.

- `SELECT * FROM t WHERE c0 GLOB c0`

  There was a logic bug in the `GLOB` that panicked when the values passed were not `TEXT`. Executing this statement after inserting a `NULL` triggered the bug, which was fixed soon after we reported it.

- `INSERT INTO t VALUES ("a")`

  Using double quotes for string literals is discouraged, and SQLite3 even provides a run-time flag for disabling it. We reported this bug as Turso panicked for the provided statement.

- `INSERT INTO t VALUES ((0 BETWEEN 0 AND 0)), (0)`

  Constant `BETWEEN` expressions were not rewritten when used within `INSERT`, but the query compiler expected them to be rewritten. We reported this bug as Turso panicked for the provided statement.

- `INSERT INTO t(c2, c0) VALUES (0, 0), (0, 0)`

  There was a bug in calculating the column indexes when inserting values to a table in reverse order. SQLancer discovered the bug when the initial table `t` had a `NOT NULL` clause for the column `c0`, which promptly failed after executing the statement. We reported this bug after manually inspecting the result of the failure.

In all six cases, we saw the benefit of generating the entire possible input space for the SQL dialect of SQLite, which DIRT does not attempt to do, thereby missing such bugs. Our evaluation demonstrates the benefit of such comprehensive generation, but also shows that without tailoring to the database, the produced bug reports will be overwhelmingly dominated by false positives and reproductions of existing bugs yet to be fixed.

4.2 Discussion

We can draw a broader set of lessons from the DIRT study for domain-specific property-based testing. The complexity of the SQL generation, combining the breadth of the SQL language constructs with maintaining a separate model, is hard to manage and has been prone to bugs in the development phase. For instance, I have realized regressions in the generator that caused it to fall into states that it would not have found the bugs it previously did. This signals a call for producing tooling that would have caught generation regressions and drifts, either by historical bug injection for ground truth measurements or continuous monitoring of coverage to detect any sudden drops in the program coverage.

We have tested interactive (or human-in-the-loop) shrinking as a complement to automated shrinking methods, and our anecdotal results were that it can be a useful supporting technique for further minimization of reported counterexamples. This is related to the sample-efficiency discussion in the previous chapter (§ 3.5), execution of SQL statements against the database can be fairly costly, so sample-efficiency of shrinking is more crucial than the ETNA workloads. Humans can be surprisingly sample-efficient, which points to the usefulness of a perhaps more general application of interactive shrinking, but also signals

that investments in domain-specific shrinking can greatly affect the quality of the resulting counterexamples for domain-specific PBT.

Finally, stateful property-based testing offers a new dimension to the PBT that I believe is underexplored, the state! In Query Plan Guidance [125], for instance, the authors use the staleness of the query plans to determine that the current state is exhausted in terms of the opportunities for exploring new behavior and generate DML (Data Manipulation Language) and DDL (Data Definition Language) statements such as `INSERT`, `DELETE`, `UPDATE`, `CREATE TABLE` or `DROP TABLE` instead of DQL (Data Query Language) statements construct via `SELECT`. I find it possible that we could apply similar insights into other domains to use domain-specific information that we can obtain through its tooling to guide the generation and mutation process.

5 Chapter
Conclusions and Future Work

The work I present in this thesis aims to enable reliable, scalable, and reproducible experimentation and research for the next generation of random testing tools. We were able to undertake experiments that would otherwise have taken weeks to setup in a matter of hours and days due to the combination of tools and techniques I've worked during the development of this thesis.

We are in the midst of a paradigm shift in the development of software. Program synthesis from natural language specifications via generative AI has given rise to a proliferation of code generation. The leverage one gains from such generation, however, is bounded by their ability to verify it, putting verification of correctness to the center of this debate. I wrote this thesis in the midst of this shift, and I have witnessed the rising interest in random testing during these last 5 years.

Property-based testing is particularly interesting within this new paradigm of code synthesis because it emphasizes specifications as a central concept. Given a complete specification of a system, the futurist in me wants to claim we could synthesize the system from scratch. In addition to developing property-based testing tools themselves, I would like to use random testing for autonomous systems optimization, translation, and synthesis; develop tools and theories advancing these layers on top of testing. I believe, however, we are far from this reality, not due to model capabilities, but rather due to our lack of understanding what makes property-based testing effective.

During my thesis work, I have worked on building the infrastructure for collecting met-

rics and measuring statistics, devising which metrics to collect and statistics to compute and empirically validating them. Today, we can measure the quality of a generator produced by an algorithm for an ETNA workload; can we go further to understand the quality of a generator written for an arbitrary project using PBT? Can we automatically tune internal parameters of a generator as needed? We produced measures for evaluating shrinking strategies, can we generalize them, use them to find better algorithms across different ecosystems? Our measurements treat mutation-based random testing as a direct alternative to blackbox generation-based random testing, what kind of fine granular metrics can we devise for better understanding the effectiveness of the former? Finally, how can we leverage Programmable PBT to augment the current property runners, can we have fault localization in PBT, can we switch between symbolic and concrete inputs, can we parallelize work only when needed?

We may be able to do all of these things, devise metrics, develop new techniques and algorithms only if we are able to evaluate them properly. I plan to horizontally grow ETNA to as many languages, ecosystems and workloads as possible; to make experimentation across the board as seamless as possible, evaluate any and all features of a PBT framework, supercharge the development for the future of property-based testing.

A Appendix
Supplementary Material

A.1 Complete Statistical Comparison for Shrinking Evaluation

The tables below give, for every (workload, generator family), the Friedman omnibus test across tasks and the post-hoc Holm-corrected pairwise Wilcoxon signed-rank tests, for five metrics: bug-finding time,[1] tree edit distance to the ground-truth minimum,[2] counterexample size, shrink time, and time per edit.[3] Per-task values are trial medians; all metrics are lower-is-better. Δ is the median per-task difference (first minus second library) and r the matched-pairs rank-biserial effect size (negative $\Rightarrow$ first library better). The reported N is the number of tasks on which all three libraries have a value for that metric.

Table A.1: Statistical comparison of bug-finding and shrinking metrics for BST.

Comparison	median Δ	r	p	p_{Holm}
Type-based generators				
Bug-finding time (ms)				

[1] Bug-finding time is the wall-clock time spent before the first failing execution. Tasks for which any of the compared libraries never produces a failing trial are excluded from this metric, so its N can differ from the shrinking metrics.

[2] Tree edit distance to ground truth, and time per edit, both require the minimal counterexample established by the exhaustive LeanCheck search. This exists for every task of BST, STLC, and $F_{<:}$, but for only 34 of RBT's 58 tasks—the remaining 24 are too deep for exhaustive search. Tasks without a ground truth are excluded from these two metrics, lowering their N relative to shrink time.

[3] Time per edit (ms/ΔTED) is undefined when shrinking produces no reduction in edit distance to the ground truth ($d \leq 0$); such tasks are excluded from this metric only, which can lower its N slightly even where ground truth is complete.

Table A.1, BST, continued

Comparison	median Δ	r	p	p_{Holm}
Friedman ($N = 48$)	$\chi^2 = 51.5$		< 0.001	—
Quick vs. Hedgehog	−56.3	−0.69	< 0.001	< 0.001
Quick vs. Falsify	−13.6	−0.80	< 0.001	< 0.001
Hedgehog vs. Falsify	+16.0	+0.58	< 0.001	< 0.001
TED to ground truth				
Friedman ($N = 48$)	$\chi^2 = 10.0$		0.007	—
Quick vs. Hedgehog	+0.0	−0.37	0.105	0.315
Quick vs. Falsify	+0.0	−0.37	0.253	0.505
Hedgehog vs. Falsify	+0.0	+0.20	0.372	0.505
Counterexample size				
Friedman ($N = 48$)	$\chi^2 = 1.0$		0.607	—
Quick vs. Hedgehog	+0.0	−1.00	0.317	0.952
Quick vs. Falsify	+0.0	−1.00	0.317	0.952
Hedgehog vs. Falsify	+0.0	−0.33	0.655	0.952
Shrink time (ms)				
Friedman ($N = 48$)	$\chi^2 = 80.8$		< 0.001	—
Quick vs. Hedgehog	−0.2	−0.92	< 0.001	< 0.001
Quick vs. Falsify	−4.0	−0.99	< 0.001	< 0.001
Hedgehog vs. Falsify	−3.9	−0.96	< 0.001	< 0.001

Table A.1, BST, continued

Comparison	median Δ	r	p	p_{Holm}
Time per edit (ms)				
Friedman ($N = 48$)	$\chi^2 = 76.0$		< 0.001	—
Quick vs. Hedgehog	−0.1	−1.00	< 0.001	< 0.001
Quick vs. Falsify	−1.3	−0.99	< 0.001	< 0.001
Hedgehog vs. Falsify	−1.2	−0.93	< 0.001	< 0.001
API-based generators				
Bug-finding time (ms)				
Friedman ($N = 53$)	$\chi^2 = 24.0$		< 0.001	—
QuickAPI vs. HedgehogAPI	−2.4	−0.87	< 0.001	< 0.001
QuickAPI vs. FalsifyAPI	−2.3	−0.62	< 0.001	< 0.001
HedgehogAPI vs. FalsifyAPI	+0.1	+0.05	0.740	0.740
TED to ground truth				
Friedman ($N = 53$)	$\chi^2 = 12.9$		0.002	—
QuickAPI vs. HedgehogAPI	+0.0	−0.63	0.006	0.017
QuickAPI vs. FalsifyAPI	+0.0	+0.01	0.976	0.976
HedgehogAPI vs. FalsifyAPI	+0.0	+0.43	0.038	0.076
Counterexample size				
Friedman ($N = 53$)	$\chi^2 = 0.1$		0.949	—
QuickAPI vs. HedgehogAPI	+0.0	+0.17	0.785	1.000

Table A.1, BST, continued

Comparison	median Δ	r	p	p_{Holm}
QuickAPI vs. FalsifyAPI	+0.0	+0.29	0.516	1.000
HedgehogAPI vs. FalsifyAPI	+0.0	+0.20	0.705	1.000
Shrink time (ms)				
Friedman ($N = 53$)	$\chi^2 = 100.1$		< 0.001	—
QuickAPI vs. HedgehogAPI	−0.5	−1.00	< 0.001	< 0.001
QuickAPI vs. FalsifyAPI	−9.2	−1.00	< 0.001	< 0.001
HedgehogAPI vs. FalsifyAPI	−8.7	−0.99	< 0.001	< 0.001
Time per edit (ms)				
Friedman ($N = 53$)	$\chi^2 = 34.7$		< 0.001	—
QuickAPI vs. HedgehogAPI	−0.0	−0.93	< 0.001	< 0.001
QuickAPI vs. FalsifyAPI	−0.1	−0.77	< 0.001	< 0.001
HedgehogAPI vs. FalsifyAPI	−0.1	−0.38	0.017	0.017
Correct-by-construction generators				
Bug-finding time (ms)				
Friedman ($N = 53$)	$\chi^2 = 1.8$		0.397	—
QuickCBC vs. HedgehogCBC	−0.0	+0.04	0.794	0.794
QuickCBC vs. FalsifyCBC	−4.2	−0.61	< 0.001	< 0.001
HedgehogCBC vs. FalsifyCBC	−4.2	−0.61	< 0.001	< 0.001
TED to ground truth				

Table A.1, BST, continued

Comparison	median Δ	r	p	p_{Holm}
Friedman ($N = 53$)	$\chi^2 = 54.2$		< 0.001	—
QuickCBC vs. HedgehogCBC	−5.0	−0.96	< 0.001	< 0.001
QuickCBC vs. FalsifyCBC	−3.0	−0.98	< 0.001	< 0.001
HedgehogCBC vs. FalsifyCBC	+0.0	−0.33	0.066	0.066
Counterexample size				
Friedman ($N = 53$)	$\chi^2 = 42.1$		< 0.001	—
QuickCBC vs. HedgehogCBC	−6.0	−1.00	< 0.001	< 0.001
QuickCBC vs. FalsifyCBC	+0.0	−1.00	< 0.001	< 0.001
HedgehogCBC vs. FalsifyCBC	+0.0	−0.12	0.539	0.539
Shrink time (ms)				
Friedman ($N = 53$)	$\chi^2 = 81.6$		< 0.001	—
QuickCBC vs. HedgehogCBC	−0.5	−0.87	< 0.001	< 0.001
QuickCBC vs. FalsifyCBC	−15.6	−1.00	< 0.001	< 0.001
HedgehogCBC vs. FalsifyCBC	−14.6	−0.89	< 0.001	< 0.001
Time per edit (ms)				
Friedman ($N = 53$)	$\chi^2 = 72.1$		< 0.001	—
QuickCBC vs. HedgehogCBC	−0.0	−0.97	< 0.001	< 0.001
QuickCBC vs. FalsifyCBC	−0.2	−0.97	< 0.001	< 0.001
HedgehogCBC vs. FalsifyCBC	−0.1	−0.76	< 0.001	< 0.001

Table A.2: Statistical comparison of bug-finding and shrinking metrics for RBT.

Comparison	median Δ	r	p	p_{Holm}
Type-based generators				
Bug-finding time (ms)				
Friedman ($N = 28$)	$\chi^2 = 38.8$		< 0.001	—
Quick vs. Hedgehog	−173.1	−0.86	< 0.001	< 0.001
Quick vs. Falsify	−16.5	−0.74	< 0.001	< 0.001
Hedgehog vs. Falsify	+76.7	+0.86	< 0.001	< 0.001
TED to ground truth				
Friedman ($N = 28$)	$\chi^2 = 6.6$		0.038	—
Quick vs. Hedgehog	+0.0	−0.41	0.205	0.615
Quick vs. Falsify	+0.0	+0.04	0.932	1.000
Hedgehog vs. Falsify	+0.0	+0.17	0.607	1.000
Counterexample size				
Friedman ($N = 28$)	$\chi^2 = 0.0$		1.000	—
Quick vs. Hedgehog	+0.0	+0.00	1.000	1.000
Quick vs. Falsify	+0.0	+0.00	1.000	1.000
Hedgehog vs. Falsify	+0.0	+0.00	1.000	1.000
Shrink time (ms)				
Friedman ($N = 28$)	$\chi^2 = 52.3$		< 0.001	—
Quick vs. Hedgehog	−0.3	−1.00	< 0.001	< 0.001

Table A.2, RBT, continued

Comparison	median Δ	r	p	p_{Holm}
Quick vs. Falsify	−2.8	−1.00	<0.001	<0.001
Hedgehog vs. Falsify	−2.5	−0.98	<0.001	<0.001
Time per edit (ms)				
Friedman ($N=28$)	$\chi^2=46.5$		<0.001	—
Quick vs. Hedgehog	−0.2	−1.00	<0.001	<0.001
Quick vs. Falsify	−1.5	−1.00	<0.001	<0.001
Hedgehog vs. Falsify	−1.3	−0.95	<0.001	<0.001
API-based generators				
Bug-finding time (ms)				
Friedman ($N=56$)	$\chi^2=54.3$		<0.001	—
QuickAPI vs. HedgehogAPI	−32.2	−0.97	<0.001	<0.001
QuickAPI vs. FalsifyAPI	−42.6	−0.84	<0.001	<0.001
HedgehogAPI vs. FalsifyAPI	+0.3	+0.29	0.060	0.060
TED to ground truth				
Friedman ($N=34$)	$\chi^2=1.8$		0.415	—
QuickAPI vs. HedgehogAPI	+0.0	+0.44	0.070	0.139
QuickAPI vs. FalsifyAPI	+0.0	+0.61	0.026	0.077
HedgehogAPI vs. FalsifyAPI	+0.0	−0.28	0.284	0.284
Counterexample size				

Table A.2, RBT, continued

Comparison	median Δ	r	p	p_{Holm}
Friedman ($N = 56$)	$\chi^2 = 13.5$		0.001	—
QuickAPI vs. HedgehogAPI	+0.0	+0.41	0.146	0.146
QuickAPI vs. FalsifyAPI	+0.0	−0.43	0.066	0.133
HedgehogAPI vs. FalsifyAPI	+0.0	−0.89	< 0.001	0.002
Shrink time (ms)				
Friedman ($N = 56$)	$\chi^2 = 110.0$		< 0.001	—
QuickAPI vs. HedgehogAPI	−1.4	−1.00	< 0.001	< 0.001
QuickAPI vs. FalsifyAPI	−48.9	−1.00	< 0.001	< 0.001
HedgehogAPI vs. FalsifyAPI	−47.9	−0.99	< 0.001	< 0.001
Time per edit (ms)				
Friedman ($N = 34$)	$\chi^2 = 25.9$		< 0.001	—
QuickAPI vs. HedgehogAPI	−0.1	−0.99	< 0.001	< 0.001
QuickAPI vs. FalsifyAPI	−0.0	−0.71	< 0.001	< 0.001
HedgehogAPI vs. FalsifyAPI	−0.0	−0.18	0.379	0.379
Correct-by-construction generators				
Bug-finding time (ms)				
Friedman ($N = 42$)	$\chi^2 = 54.1$		< 0.001	—
QuickCBC vs. HedgehogCBC	−60.1	−0.98	< 0.001	< 0.001
QuickCBC vs. FalsifyCBC	−26.0	−0.88	< 0.001	< 0.001

Table A.2, RBT, continued

Comparison	median Δ	r	p	p_{Holm}
HedgehogCBC vs. FalsifyCBC	+42.6	+0.94	<0.001	<0.001
TED to ground truth				
Friedman ($N=30$)	$\chi^2=21.4$		<0.001	—
QuickCBC vs. HedgehogCBC	−0.5	−0.57	0.025	0.038
QuickCBC vs. FalsifyCBC	−1.5	−0.80	<0.001	0.002
HedgehogCBC vs. FalsifyCBC	−1.0	−0.57	0.019	0.038
Counterexample size				
Friedman ($N=42$)	$\chi^2=9.7$		0.008	—
QuickCBC vs. HedgehogCBC	+0.0	−0.55	0.078	0.156
QuickCBC vs. FalsifyCBC	+0.0	−0.82	0.015	0.045
HedgehogCBC vs. FalsifyCBC	+0.0	−0.15	0.633	0.633
Shrink time (ms)				
Friedman ($N=42$)	$\chi^2=34.5$		<0.001	—
QuickCBC vs. HedgehogCBC	−0.4	+0.12	0.516	0.516
QuickCBC vs. FalsifyCBC	−5.0	−0.94	<0.001	<0.001
HedgehogCBC vs. FalsifyCBC	−5.0	−0.88	<0.001	<0.001
Time per edit (ms)				
Friedman ($N=30$)	$\chi^2=39.5$		<0.001	—
QuickCBC vs. HedgehogCBC	−0.2	−0.88	<0.001	<0.001

Table A.2, RBT, continued

Comparison	median Δ	r	p	p_{Holm}
QuickCBC vs. FalsifyCBC	−0.1	−0.99	<0.001	<0.001
HedgehogCBC vs. FalsifyCBC	+0.0	−0.26	0.213	0.213

Table A.3: Statistical comparison of bug-finding and shrinking metrics for STLC.

Comparison	median Δ	r	p	p_{Holm}
Type-based generators				
Bug-finding time (ms)				
Friedman ($N=16$)	$\chi^2=1.6$		0.444	—
Quick vs. Hedgehog	−7.8	−0.16	0.597	1.000
Quick vs. Falsify	+47.6	+0.13	0.669	1.000
Hedgehog vs. Falsify	+143.6	+0.19	0.528	1.000
TED to ground truth				
Friedman ($N=16$)	$\chi^2=14.5$		<0.001	—
Quick vs. Hedgehog	−2.5	−0.78	0.006	0.017
Quick vs. Falsify	−1.0	−0.56	0.058	0.117
Hedgehog vs. Falsify	+1.0	+0.54	0.086	0.117
Counterexample size				
Friedman ($N=16$)	$\chi^2=8.2$		0.017	—

Table A.3, STLC, continued

Comparison	median Δ	r	p	p_{Holm}
Quick vs. Hedgehog	−2.0	−0.88	0.008	0.024
Quick vs. Falsify	+0.0	−0.79	0.054	0.107
Hedgehog vs. Falsify	+0.0	+0.53	0.133	0.133
Shrink time (ms)				
Friedman ($N = 16$)	$\chi^2 = 32.0$		< 0.001	—
Quick vs. Hedgehog	−0.3	−1.00	< 0.001	< 0.001
Quick vs. Falsify	−23.5	−1.00	< 0.001	< 0.001
Hedgehog vs. Falsify	−23.3	−1.00	< 0.001	< 0.001
Time per edit (ms)				
Friedman ($N = 16$)	$\chi^2 = 25.1$		< 0.001	—
Quick vs. Hedgehog	−0.0	−0.35	0.231	0.231
Quick vs. Falsify	−2.1	−1.00	< 0.001	< 0.001
Hedgehog vs. Falsify	−2.1	−1.00	< 0.001	< 0.001
Correct-by-construction generators				
Bug-finding time (ms)				
Friedman ($N = 20$)	$\chi^2 = 24.1$		< 0.001	—
QuickCBC vs. HedgehogCBC	−4.1	−0.99	< 0.001	< 0.001
QuickCBC vs. FalsifyCBC	−1.5	−0.94	< 0.001	< 0.001
HedgehogCBC vs. FalsifyCBC	+0.9	+0.42	0.105	0.105

Table A.3, STLC, continued

Comparison	median Δ	r	p	p_{Holm}
TED to ground truth				
Friedman ($N = 20$)		$\chi^2 = 14.7$	< 0.001	—
QuickCBC vs. HedgehogCBC	−4.5	+0.00	1.000	1.000
QuickCBC vs. FalsifyCBC	+5.0	+0.54	0.033	0.066
HedgehogCBC vs. FalsifyCBC	+7.8	+1.00	< 0.001	< 0.001
Counterexample size				
Friedman ($N = 20$)		$\chi^2 = 14.7$	< 0.001	—
QuickCBC vs. HedgehogCBC	−4.2	+0.01	0.968	0.968
QuickCBC vs. FalsifyCBC	+5.8	+0.61	0.022	0.044
HedgehogCBC vs. FalsifyCBC	+7.8	+1.00	< 0.001	< 0.001
Shrink time (ms)				
Friedman ($N = 20$)		$\chi^2 = 40.0$	< 0.001	—
QuickCBC vs. HedgehogCBC	−0.6	−1.00	< 0.001	< 0.001
QuickCBC vs. FalsifyCBC	−6.5	−1.00	< 0.001	< 0.001
HedgehogCBC vs. FalsifyCBC	−5.9	−1.00	< 0.001	< 0.001
Time per edit (ms)				
Friedman ($N = 20$)		$\chi^2 = 36.4$	< 0.001	—
QuickCBC vs. HedgehogCBC	−0.0	−0.95	< 0.001	< 0.001
QuickCBC vs. FalsifyCBC	−0.3	−1.00	< 0.001	< 0.001

Table A.3, STLC, continued

Comparison	median Δ	r	p	p_{Holm}
HedgehogCBC vs. FalsifyCBC	−0.3	−1.00	<0.001	<0.001

Table A.4: Statistical comparison of bug-finding and shrinking metrics for $F_{<:}$.

Comparison	median Δ	r	p	p_{Holm}
Type-based generators				
Bug-finding time (ms)				
Friedman ($N=26$)	$\chi^2=46.7$		<0.001	—
Quick vs. Hedgehog	−1055.2	−1.00	<0.001	<0.001
Quick vs. Falsify	−115.4	−0.89	<0.001	<0.001
Hedgehog vs. Falsify	+776.8	+1.00	<0.001	<0.001
TED to ground truth				
Friedman ($N=26$)	$\chi^2=12.5$		0.002	—
Quick vs. Hedgehog	−2.5	−0.88	<0.001	0.001
Quick vs. Falsify	−2.2	−0.81	0.001	0.002
Hedgehog vs. Falsify	+0.0	+0.45	0.079	0.079
Counterexample size				
Friedman ($N=26$)	$\chi^2=29.5$		<0.001	—
Quick vs. Hedgehog	−3.0	−0.96	<0.001	<0.001

Table A.4, $F_{<:}$, continued

Comparison	median Δ	r	p	p_{Holm}
Quick vs. Falsify	−2.2	−0.94	< 0.001	0.001
Hedgehog vs. Falsify	+0.5	+0.88	0.002	0.002
Shrink time (ms)				
Friedman ($N = 26$)		$\chi^2 = 52.0$	< 0.001	—
Quick vs. Hedgehog	−0.2	−1.00	< 0.001	< 0.001
Quick vs. Falsify	−2.7	−1.00	< 0.001	< 0.001
Hedgehog vs. Falsify	−2.5	−1.00	< 0.001	< 0.001
Time per edit (ms)				
Friedman ($N = 26$)		$\chi^2 = 48.3$	< 0.001	—
Quick vs. Hedgehog	−0.0	−0.91	< 0.001	< 0.001
Quick vs. Falsify	−0.3	−1.00	< 0.001	< 0.001
Hedgehog vs. Falsify	−0.2	−1.00	< 0.001	< 0.001
Correct-by-construction generators				
Bug-finding time (ms)				
Friedman ($N = 36$)		$\chi^2 = 14.9$	< 0.001	—
QuickCBC vs. HedgehogCBC	−2.1	−0.75	< 0.001	< 0.001
QuickCBC vs. FalsifyCBC	−0.7	−0.59	0.001	0.003
HedgehogCBC vs. FalsifyCBC	+0.4	+0.49	0.010	0.010
TED to ground truth				

Table A.4, $F_{<:}$, continued

Comparison	median Δ	r	p	p_{Holm}
Friedman ($N = 36$)	$\chi^2 = 56.0$		< 0.001	—
QuickCBC vs. HedgehogCBC	−60.2	−1.00	< 0.001	< 0.001
QuickCBC vs. FalsifyCBC	−52.0	−1.00	< 0.001	< 0.001
HedgehogCBC vs. FalsifyCBC	+7.2	+0.40	0.035	0.035
Counterexample size				
Friedman ($N = 36$)	$\chi^2 = 56.0$		< 0.001	—
QuickCBC vs. HedgehogCBC	−46.0	−1.00	< 0.001	< 0.001
QuickCBC vs. FalsifyCBC	−39.5	−1.00	< 0.001	< 0.001
HedgehogCBC vs. FalsifyCBC	+5.2	+0.39	0.039	0.039
Shrink time (ms)				
Friedman ($N = 36$)	$\chi^2 = 62.0$		< 0.001	—
QuickCBC vs. HedgehogCBC	−1.6	−1.00	< 0.001	< 0.001
QuickCBC vs. FalsifyCBC	−21.8	−1.00	< 0.001	< 0.001
HedgehogCBC vs. FalsifyCBC	−19.7	−0.92	< 0.001	< 0.001
Time per edit (ms)				
Friedman ($N = 36$)	$\chi^2 = 68.2$		< 0.001	—
QuickCBC vs. HedgehogCBC	−0.0	−1.00	< 0.001	< 0.001
QuickCBC vs. FalsifyCBC	−0.2	−1.00	< 0.001	< 0.001
HedgehogCBC vs. FalsifyCBC	−0.2	−0.95	< 0.001	< 0.001

A Appendix
Bibliography

[1] jmid. pbt-frameworks. `https://github.com/jmid/pbt-frameworks`. Accessed: 2026-03-30.

[2] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279. ACM, 2000.

[3] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[4] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. Retrieved 2024-07-01 16:50:18+02:00.

[5] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*, WOOT'20, USA, 2020. USENIX Association.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Trans. Software Eng.*, 45(5):489–506, 2019.

[7] Caroline Lemieux and Koushik Sen. Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 475–485, New York, NY, USA, 2018. Association for Computing Machinery.

[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.

[9] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. FuzzFactory: domain-specific fuzzing with waypoints. *Replication Package for "FuzzFactory: Domain-Specific Fuzzing with Waypoints"*, 3(OOPSLA):174:1–174:29, October 2019.

[10] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[11] David R. MacIver. Hypothesis: Property-based testing for python. `https://hypothesis.works/`, 2016.

[12] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 329–340, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Stephen Dolan. Property fuzzing for ocaml. `https://github.com/stedolan/crowbar`, 2017.

[14] Harrison Goldstein and Benjamin C. Pierce. Parsing Randomness. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):128:89–128:113, October 2022.

[15] Koen Claessen and Michał Pałka. Splittable pseudorandom number generators using cryptographic hashing. In *ACM SIGPLAN Symposium on Haskell*, pages 47–58. ACM, 2013.

[16] David Maciver and Alastair F. Donaldson. Test-case reduction via test-case generation: Insights from the hypothesis reducer (tool insights paper). In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 13:1–13:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[17] Alperen Keles, Justine Frank, Ceren Mert, Harrison Goldstein, and Leonidas Lampropoulos. Programmable property-based testing, 2026.

[18] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C. Pierce, and Leonidas Lampropoulos. Etna: An evaluation platform for property-based testing (experience report). *Proc. ACM Program. Lang.*, 7(ICFP), aug 2023.

[19] Alperen Keles, Ethan Chou, Harrison Goldstein, and Leonidas Lampropoulos. Dirt: Database-integrated random testing. In *Proceedings of the 2026 11th International Workshop on Testing Database Systems*, DBTest '26, page 23–30, New York, NY, USA, 2026. Association for Computing Machinery.

[20] John Hughes. How to specify it! - a guide to writing properties of pure functions. In *Symposium on Trends in Functional Programming*, 2019.

[21] Leonidas Lampropoulos and Benjamin C. Pierce. *QuickChick: Property-Based Testing In Coq*. Software Foundations series, volume 4. Electronic textbook, August 2018.

[22] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy Smallcheck: automatic exhaustive testing for small values. In *1st ACM SIGPLAN Symposium on Haskell*, pages 37–48. ACM, 2008.

[23] Jacob Stanley. Hedgehog: Release with confidence. `https://hackage.haskell.org/package/hedgehog/`, 2019.

[24] leancheck: Enumerative property-based testing. `https://hackage.haskell.org/package/leancheck`.

[25] Edsko de Vries. falsify: Internal shrinking reimagined for haskell. In *Proceedings of the 16th ACM SIGPLAN International Haskell Symposium*, Haskell 2023, page 97–109, New York, NY, USA, 2023. Association for Computing Machinery.

[26] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294, 2011.

[27] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 335–346, 2012.

[28] Edsko de Vries. falsify: Internal shrinking reimagined for haskell. In *Proceedings of the 16th ACM SIGPLAN International Haskell Symposium*, Haskell 2023, page 97–109, New York, NY, USA, 2023. Association for Computing Machinery.

[29] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. Coverage guided, property based testing. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications* (*OOPSLA*), October 2019.

[30] Rohan Padhye, Caroline Lemieux, and Koushik Sen. Jqf: coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 398–401, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Zac Hatfield-Dodds and contributors. Hypofuzz: Adaptive fuzzing of hypothesis tests.

[32] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2155–2168, New York, NY, USA, 2017. Association for Computing Machinery.

[33] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 254–265, New York, NY, USA, 2018. Association for Computing Machinery.

[34] SQLancer contributors. SQLancer: Automated testing to find logic and performance bugs in database systems.

[35] Jinsheng Ba and Manuel Rigger. Testing database engines via query plan guidance. In *The 45th International Conference on Software Engineering* (*ICSE'23*), May 2023.

[36] Andreas Löscher and Konstantinos Sagonas. Targeted property-based testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 46–56, New York, NY, USA, 2017. Association for Computing Machinery.

[37] Harrison Goldstein, John Hughes, Leonidas Lampropoulos, and Benjamin C. Pierce. Do judge a test by its cover: Combining combinatorial and property-based testing. In *Proceedings of the European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science, 2021.

[38] Segev Elazar Mittelman, Alvin Resnick, Ivan Perez, Alwyn Goodloe, and Leonidas Lampropoulos. Don't go down the rabbit hole: Reprioritizing enumeration for property-based testing. In *Proceedings of the ACM SIGPLAN International Symposium on Haskell*, 2023.

[39] Robert Krook, Nicholas Smallbone, Bo Joel Svensson, and Koen Claessen. Quickercheck: Implementing and evaluating a parallel run-time for quickcheck, 2024.

[40] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 78–88, New York, NY, USA, 2012. ACM.

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

[42] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, December 2018.

[43] Eirini Arvaniti. Automated random model-based testing of stateful systems. Diploma thesis, National Technical University of Athens, School of Electrical and Computer Engineering, July 2011.

[44] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[45] Manuel Rigger and Zhendong Su. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 1140–1152, New York, NY, USA, 2020. Association for Computing Machinery.

[46] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[47] Ethan Khan Chou. Model-free stateful random testing. Master's thesis, University of Maryland, College Park, 2025. Advisor: Leonidas Lampropoulos.

[48] Richard J. Boulton, Andy Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in hol. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156. North-Holland Publishing Co., June 1992.

[49] Jacob Prinz, Alex Kavvos, and Leonidas Lampropoulos. Deeper shallow embeddings. In *13th International Conference on Interactive Theorem Proving (ITP)*, Lecture Notes in Computer Science, 2022.

[50] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 143–156, New York, NY, USA, 2008. ACM.

[51] Yao Li and Stephanie Weirich. Program adverbs and tlön embeddings. *Proceedings of the ACM on Programming Languages*, 6, 07 2022.

[52] Josie Holmes, Alex Groce, Jervis Pinto, Pranjal Mittal, Pooria Azimi, Kevin Kellar, and James O'Brien. TSTL: The template scripting testing language. *International Journal on Software Tools for Technology Transfer*, 20(1):57–78, 2018.

[53] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, volume 187 of *ICSE '24*, pages 1–13, Lisbon, Portugal, 2024. Association for Computing Machinery.

[54] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, page 199–208, New York, NY, USA, 1988. Association for Computing Machinery.

[55] Bogdan Popa. Rackcheck: Property-based testing for racket. `https://docs.racket-lang.org/rackcheck/index.html`, 2021.

[56] Leonidas Lampropoulos. *Random Testing for Language Design*. PhD thesis, University of Pennsylvania, 2018.

[57] Robert Krook, Nicholas Smallbone, Bo Joel Svensson, and Koen Claessen. Quickercheck: Implementing and evaluating a parallel run-time for quickcheck. In *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages*, IFL '23, New York, NY, USA, 2024. Association for Computing Machinery.

[58] Andreas Löcher and Konstantinos Sagonas. Automating targeted property-based testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 70–80, 2018.

[59] Cameron Bytheway. Bolero: A fuzzing and property testing front-end framework for rust.

[60] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. Generative type-aware mutation for testing SMT solvers. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–19, 2021.

[61] Jingling Sun, Ting Su, Jiayi Jiang, Jue Wang, Geguang Pu, and Zhendong Su. Property-based fuzzing for finding data manipulation errors in android apps. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1088–1100. ACM, 2023.

[62] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Validity fuzzing and parametric generators for effective random testing. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 266–267. IEEE / ACM, 2019.

[63] Alperen Keles, Jessica Shi, Nikhil Kamath, Tin Nam Liu, Ceren Mert, Harrison Goldstein, Benjamin C. Pierce, and Leonidas Lampropoulos. Etna: An evaluation platform for property-based testing, 2026.

[64] Jinfu Chen, Shengran Wang, Saihua Cai, Chi Zhang, Haibo Chen, Jingyi Chen, and Jianming Zhang. A novel coverage-guided greybox fuzzing based on power schedule optimization with time complexity. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[65] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. Boosting fuzzer efficiency: an information theoretic perspective. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 678–689. ACM, 2020.

[66] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196, December 1996.

[67] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020.

[68] Yao Li and Stephanie Weirich. Program adverbs and tlön embeddings. *Proc. ACM Program. Lang.*, 6(ICFP):312–342, 2022.

[69] Jacques Carrete, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

[70] Kazutaka Matsuda, Samantha Frohlich, Meng Wang, and Nicolas Wu. Embedding by unembedding. *Proc. ACM Program. Lang.*, 7(ICFP), August 2023.

[71] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. Association for Computing Machinery.

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

[73] Catalin Hritcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. Testing noninterference, quickly. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2013.

[74] Catalin Hritcu, Leonidas Lampropoulos, Antal Spector-Zabusky, Arthur Azevedo Amorim, Maxime Denes, John Hughes, Benjamin C. Pierce, and Dimitrios Vytiniotis. Testing noninterference, quickly. In *Journal of Functional Programming (JFP)*, 2016.

[75] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C. Pierce, and Leonidas Lampropoulos. Etna: An evaluation platform for property-based testing (experience report). *Proc. ACM Program. Lang.*, 7(ICFP), August 2023.

[76] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, 2016.

[77] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), December 2020.

[78] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 72–82, New York, NY, USA, 2014. Association for Computing Machinery.

[79] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery.

[80] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[81] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. FIXREVERTER: A realistic bug injection methodology for benchmarking fuzz testing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3699–3715, Boston, MA, August 2022. USENIX Association.

[82] Alperen Keles. etna-cli, 2026.

[83] Alperen Keles. etna-index, 2026.

[84] Alperen Keles. etna-schemas, 2026.

[85] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000.

[86] Alperen Keles. A declarative framework for hand-crafted mutation analysis and management, 2026.

[87] Agustín Mista and Alejandro Russo. Generating random structurally rich algebraic data type values. In *Proceedings of the 14th International Workshop on Automation of Software Test*, AST '19, page 48–54. IEEE Press, 2019.

[88] Casey Klein and Robert Bruce Findler. Randomized testing in PLT Redex. In *Workshop on Scheme and Functional Programming (SFP)*, 2009.

[89] Michael H. Palka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 91–97, New York, NY, USA, 2011. ACM.

[90] Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. Effect-driven quickchecking of compilers. *Proc. ACM Program. Lang.*, 1(ICFP), aug 2017.

[91] Tram Hoang, Anton Trunov, Leonidas Lampropoulos, and Ilya Sergey. Random testing of a higher-order blockchain language (experience report). In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2022.

[92] Benjamin C. Pierce. *Software Foundations*. Software Foundations series, volume 4. Electronic textbook, August 2018.

[93] John Hughes, Benjamin C. Pierce, Thomas Arts, and Ulf Norell. Mysteries of Dropbox: Property-based testing of a distributed synchronization service. In *International Conference on Software Testing, Verification and Validation (ICST)*, April 2016.

[94] Rudy Matela. Tools for discovery, refinement and generalization of functional properties by enumerative testing, 2017.

[95] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.

[96] Tim Henderson. Zhang-shasha: Tree edit distance in python. `https://github.com/timtadh/zhang-shasha`, 2016.

[97] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30, 2006.

[98] Cynthia Richey, Joseph W. Cutler, Harrison Goldstein, and Benjamin C. Pierce. Fail faster: Staging and fast randomness for high-performance pbt, 2026.

[99] Zain K Aamer and Benjamin C. Pierce. Bennet: Randomized specification testing for heap-manipulating programs. *Proc. ACM Program. Lang.*, 9(OOPSLA2), October 2025.

[100] Segev Elazar Mittelman, Harry Goldstein, and Leonidas Lampropoulos. Testing theorems, fully automatically. In *OOPSLA'26*. Association for Computing Machinery, 2026.

[101] Ryan Tjoa, Poorva Garg, Harrison Goldstein, Todd Millstein, Benjamin C. Pierce, and Guy Van den Broeck. Tuning random generators: Property-based testing as probabilistic programming. *Proc. ACM Program. Lang.*, 9(OOPSLA2), October 2025.

[102] Harrison Goldstein. Tyche: In Situ Exploration of Random Testing Effectiveness (Demo), October 2023.

[103] Vasudev Vikram, Caroline Lemieux, Joshua Sunshine, and Rohan Padhye. Can large language models write good property-based tests?, 2024.

[104] Khashayar Etemadi, Marjan Sirjani, Mahshid Helali Moghadam, Per Strandberg, and Paul Pettersson. Llm-based property-based test generation for guardrailing cyber-physical systems, 2025.

[105] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. PropertyGPT: LLM-driven formal verification of smart contracts through retrieval-augmented property generation, 2024.

[106] Lucas Jing, Xinqi Wang, Liao Zhang, and Simon S. Du. Pbt-bench: Benchmarking ai agents on property-based testing, 2026.

[107] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*, pages 142–151. ACM, 2006.

[108] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*, pages 361–371. ACM, 2018.

[109] Satia Herfert, Jibesh Patra, and Michael Pradel. Automatically reducing tree-structured test inputs. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 861–871. IEEE Press, 2017.

[110] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. Probabilistic delta debugging. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 881–892. ACM, 2021.

[111] Xintong Zhou, Zhenyang Xu, Mengxiao Zhang, Yongqiang Tian, and Chengnian Sun. WDD: Weighted delta debugging. In *2025 IEEE/ACM 47th International Conference on Software Engineering*, pages 1592–1603. IEEE, 2025.

[112] Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1017–1032. ACM, 2021.

[113] Li Lin, Zongyin Hao, Chengpeng Wang, Zhuangda Wang, Rongxin Wu, and Gang Fan. SQLess: Dialect-agnostic sql query simplification. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 743–754. ACM, 2024.

[114] Christian Gram Kalhauge and Jens Palsberg. Binary reduction of dependency graphs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 556–566. ACM, 2019.

[115] Kostas Ferles, Valentin Wüstholz, Maria Christakis, and Isil Dillig. Failure-directed program trimming. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 174–185. ACM, 2017.

[116] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: Automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 476–486. ACM, 2017.

[117] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 380–394. ACM, 2018.

[118] Turso Contributors. Turso, 2026.

[119] Andreas Seltenreich. Sqlsmith. `https://github.com/anse1/sqlsmith`, 2015.

[120] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery.

[121] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. Property-based testing in practice. In *Proceedings of the IEEE/ACM 46th*

International Conference on Software Engineering, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[122] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woon-Hak Kang. APOLLO: automatic detection and diagnosis of performance regressions in database systems. *PVLDB*, 13(1):57–70, 2019.

[123] John Hughes. How to specify it! a guide to writing properties of pure functions. In *Trends in Functional Programming: 20th International Symposium, TFP 2019, Vancouver, BC, Canada, June 12–14, 2019, Revised Selected Papers*, pages 58–83, Berlin, Heidelberg, 2019. Springer-Verlag.

[124] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[125] Jinsheng Ba and Manuel Rigger. Testing database engines via query plan guidance. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, pages 2060–2071. IEEE Press, 2023.

[126] Jinsheng Ba and Manuel Rigger. Cert: Finding performance issues in database systems through the lens of cardinality estimation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[127] Jinsheng Ba and Manuel Rigger. Keep it simple: Testing databases via differential query plans. *Proc. ACM Manag. Data*, 2(3), May 2024.

[128] Chi Zhang and Manuel Rigger. Constant optimization driven database system testing. *Proc. ACM Manag. Data*, 3(1), February 2025.

[129] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 2653–2666, New York, NY, USA, 2021. Association for Computing Machinery.

[130] Catalin Hritcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. Testing noninterference, quickly. *SIGPLAN Not.*, 48(9):455–468, September 2013.

[131] Richard D Hipp. SQLite, 2020.

[132] Ying Fu, Zhiyong Wu, Yuanliang Zhang, Jie Liang, Jingzhou Fu, Yu Jiang, Shanshan Li, and Xiangke Liao. Thanos: DBMS Bug Detection via Storage Engine Rotation Based

Differential Testing . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 655–666, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[133] Yu Liang, Song Liu, and Hong Hu. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *Proceedings of the 31st USENIX Security Symposium (USENIX 2022)*, Boston, MA, aug 2022.

[134] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. Griffin: Grammar-free dbms fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[135] Suyang Zhong and Manuel Rigger. Scaling automated database system testing, 2026.