Property-Based Testing in Haskell
An Analysis of QuickCheck usage in Open-Source Haskell Projects

Ye Zhao

Supervisor(s): Andreea Costea, Sára Juhošová

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Ye Zhao
Final project course: CSE3000 Research Project
Thesis committee: Andreea Costea, Sára Juhošová, Marco Zuñiga Zamalloa

Abstract

Property-Based Testing (PBT) with QuickCheck has become a cornerstone of reliable software development in Haskell, yet there is little systematic understanding of how developers employ it in production quality libraries. In this study, we perform an empirical analysis of QuickCheck usage in nine representative open-source Haskell projects (`aeson`, `attoparsec`, `bytestring`, `containers`, `hashable`, `lens`, `megaparsec`, `pandoc-type` , `text` and `vector`). We extract thousands of QuickCheck properties and manually sample 217 of them to classify property types such as invariants, roundtrip checks, algorithm correctness, and idempotence. We also analyze all extracted properties to identify patterns in generator and shrinking strategies. Our Results show that invariants and test-oracle tests dominate the sampled properties, while custom generators are used in 55.8% of all properties and custom shrinkers in only 22.12%. We discuss how these patterns vary across different domains (data structures, text processing, optics, hashing) and highlight the tension between default and custom test infrastructure. Finally, we reflect on the implications for tool support, education, and future research particularly automatic generator inference and improved shrinking utilities to lower the barrier to effective PBT in large codebases.

1 Introduction

In modern software engineering, ensuring the correctness and reliability of software systems remains a critical challenge. Traditional testing approaches, such as unit testing and integration testing, rely heavily on manually written example-based tests. However, such approaches are often insufficient for uncovering edge cases or providing strong correctness guarantees. Property-Based Testing (PBT) has emerged as a complementary technique, where instead of specifying individual test cases, developers describe properties that the software should satisfy, and the testing framework automatically generates numerous random inputs to check these properties.

One of the most mature and influential tools in this area is QuickCheck for Haskell, pioneered by Koen Claessen and John Hughes in 2000 [2]. QuickCheck introduced the concept of automatic test data generation and shrinking to simplify failing test cases, leveraging Haskell's strong type system and functional purity. Besides safe default generators, QuickCheck still allows users to write custom generators for constrained or recursive structures. Its shrinking heuristic tries nearby candidates (sub-lists, smaller numbers, shallower trees) until the smallest failing input is found. Its design has inspired similar tools in other languages, such as Hypothesis in Python, proptest in Rust, and jqwik in Java. However, recent studies highlight a lag in evaluation: even experienced developers often lack feedback on how well random generated inputs exercise the code under test, leading to high rejection rates and blind trust in "All tests passed" [8].
Beyond tooling concerns, why and how practitioners use PBT in the wild is still only partially understood. Goldstein et al. [7] report that developers lean on a handful of "high-leverage" idioms—round-trip, differential, and model-based properties—while generator authoring and shrinking remain pain points. Complementary work on Python's Hypothesis shows a similar concentration of property kinds and raises questions about external validity across languages [3]. While there is abundant documentation and tutorials on how to write property-based tests with QuickCheck, there is a lack of systematic analysis on how QuickCheck is actually used in open source projects of real-world Haskell.

Questions remain about the types of properties developers choose to write, the extent of generator customization, and how shrinking strategies are employed in practice. This project seeks to bridge this knowledge gap by conducting an empirical analysis of existing Haskell open-source projects. This analysis will answer a main research question:

How is Property-Based Testing with QuickCheck applied in real-world Haskell open-source projects?

To answer this question, we decompose it into following research questions:

RQ1: What types of properties are typically tested using QuickCheck?

RQ2: Which types of quantifiers and logical connectives are used in these properties?

RQ3: How does property-based testing complement other testing strategies such as unit tests?

RQ4: How and when are generators implemented?
RQ5: In which scenarios do developers explicitly define shrinking strategies?

The outcome will contribute to our understanding of how developers apply PBT in practice, potentially guiding better tooling, documentation, and education in the Haskell community and beyond.

The remainder of this paper is organized as follows. Section 2 details our methodology, covering repository selection and data-collection, the open-coding procedure and responsible research. Section 3 reports the empirical results, providing qualitative findings. Section 4 discusses those findings in depth, highlights limitations. Section 5 concludes with answers to our research questions and directions for future work.

2 Methodology

This section explains the approach undertaken in our research procedure. It includes the selection of target Haskell open-source repositories and methodology for data collection. Furthermore, we address the principles of responsible research, emphasizing transparency, reproducibility, and the mitigation of potential biases. The aim is to provide a clear and replicable account of how the evidence was gathered and analyzed to answer the stated research questions.

2.1 Repository Selection and Data Collection

To investigate the application of Property-Based Testing (PBT) with QuickCheck in real-world Haskell open-source projects, we first selected a set of prominent and actively maintained libraries. The criteria for project selection was designed to ensure both the relevance of the chosen projects to the domain of PBT and their representativeness of common Haskell development practices. The selected projects are listed in Table 1.

Table 1: Selected Repositories

Repository	Stars	Version
aeson	1.3K	2.2.3.0
attoparsec	524	0.14.4
bytestring[1]	298	0.13.0.0
containers	341	0.8
hashable	103	1.5.1.0
lens	2.1K	5.3.4
megaparsec	946	9.7.0
pandoc-types	112	1.23.1
text	415	2.1.2
vector	379	0.13.2.0

1 *bytestring* is excluded from the qualitative sample.

We first performed a broad search using the GitHub API to identify a candidate pool of repositories. We retained Haskell repositories if they had (i) commits within the last 12 months, (ii) import the Test. QuickCheck library, and (iii) a permissive license. This process initially produced over twenty diverse repositories, containing hundreds of QuickCheck test files. From this candidate pool, manual selection will ensure the chosen repositories are suitable and contain meaningful QuickCheck test cases. We considered functional domain coverage and community impact in this manual step so that we build a varied and influential sample rather than being an arbitrary selection. It finally determined a set of 10 repositories for analysis as listed in Table 1.

Once identified, the property-based test cases, along with their associated generator and shrinking code, will be extracted from these projects for further analysis. Inside each `test/` hierarchy we located QuickCheck declarations via a regex that matches either `testProperty` (the `tasty` registration API) or an explicit `property $` call. Every match, typically a `prop_` function, is treated as one *sampling unit*.

This extraction yielded a large corpus Property-Based Tests. Recognizing that some projects contain large clusters of similar tests and that a manual analysis of all properties is infeasible with time constraint, we decided to random sample from each repository. In this sampling procedure, we excluded the bytestring test suite. On one hand it has over thousand test cases, on the other hands, it registers many property in a way that the regex heuristic can not capture easily. To avoid systematic under-counting we removed that suite from the qualitative phase. All subsequent analyses therefore concern the remaining nine libraries. After exclusion, the random draw resulted in 217 tests.

2.2 Data Analysis

With the test cases collected, the coding analysis phase focuses on thoroughly examining each property. We applied open coding, a standard qualitative analysis technique, to inductively identify and classify key characteristics of QuickCheck property usage in the Haskell ecosystem. This research is a sub-project of a multi-environment project, therefore, we combined our findings in the early stage of the study, ensuring the consistency across different languages and frameworks analysis. We collaboratively developed a shared dictionary with existing literatures[2] and checklist of general dimensions, then adjust them carefully based on specific environment:

- Python/Hypothesis[4],
- Rust/proptest[1],
- Rust/QuickCheck[5],
- Java/jqwik[9].

For the Haskell and QuickCheck analysis, each extracted property will be manually labeled and categorized based on 4 dimensions:

- the type of property (as defined below),
- the complexity and logical structure (noting the presence of quantifiers or logical operators),
- the use of generators (distinguishing between default and custom implementations),
- the shrinking strategy (whether explicitly defined or relying on QuickCheck's default behavior).

The property intent labels are defined below:
Invariant. The transformation must leave some projection of the value unchanged for all inputs and branches (includes error-handling and boundary checks).
Test-Oracle Behaviour is validated by comparing against a trusted reference—either a standard-library function or a simpler "golden" implementation.
Hard-to-Prove / Easy-to-Verify Algebraic or structural laws that would be tedious to prove formally (e.g. monoid or lens laws) but are trivial to check on random data.
Round-Trip Applying an operation and its inverse—encode then decode, set then get—must yield the original value.
Different Paths, Same Destination Intended for checks that multiple execution routes converge to the same result
Idempotence Repeating an operation has no further effect: f (f x) == f x.

[2]https://fsharpforfunandprofit.com/posts/property-based-testing-2/

Structural Induction Explicit base-case and inductive-step properties over a recursive structure.

Although some QuickCheck properties naturally satisfy more than one of our categories, we assign exactly one label per property. This decision keeps frequency statistics interpretable and prevents double-counting. We use a simple priority rule:

- Round-Trip beats Invariant (round-trip is the clearer intention).
- Test-Oracle beats Idempotence or Invariant (the key idea is to compare with a reference).
- Hard-to-Prove / Easy-to-Verify wins whenever the property states an algebraic or structural law (e.g. lens), even if it also happens to be an invariant.

We applied these definitions and rules on sampled dataset to detect patterns in generator and shrinking configurations, ensuring consistency. The findings will be summarized through visualizations in Section 3 to clearly illustrate the distribution and characteristics of the observed patterns.

2.3 Responsible Research

Our data sources are open and license-friendly. All analysis objects are public repositories under permissive licenses such as BSD-3. These licences explicitly allow inspection, archival, and redistribution of source code for research purposes, provided copyright notices are retained. Our study therefore modified no code and operate all analysis on read-only clones.

No personally identifiable information is collected or processed during the research process, so no additional IRB(Institutional Review Board) approval is required.

We respect the work of developers. When citing code snippets, we annotate the library name, and file path to ensure that contributors receive transparent and traceable academic citations.We give fair presentation of results. When reporting observations, we only make objective descriptions based on the published version and do not evaluate individual developers.

We used Large Language Model(LLM) throught out the research workflow. It played a role as a writing assistant and secondary verifier after manual extraction and coding steps. We used OpenAI GPT-4 to generate preliminary paragraph drafts, phrasing suggestions, and title variants, then always reviewed and edited those content. We also use it for grammar and spell check in writing process. We viewed and approved all methodological decisions, data collection and analysis. The LLM assistant did not contribute original research ideas. There was no executable code pasted verbatim and no AI-generated code appears in the final dataset.

3 Results

This section presents the empirical results derived from the systematic analysis of QuickCheck property tests across

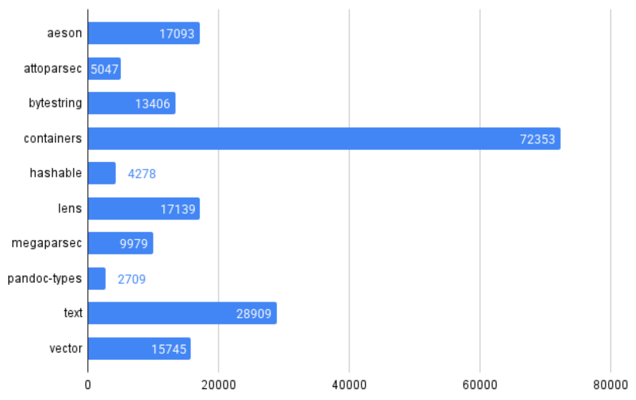

Figure 1: Lines Of Code in Haskell

the selected Haskell open-source projects. We begin by providing a general overview of PBT adoption. Then we go through the main findings related to the types of properties commonly tested, the prevalent use of quantifiers and logical connectives, and the observed patterns in generator and shrinking strategy implementations. Detailed breakdowns, supported by illustrative examples from the codebase, will highlight the practical application of QuickCheck features.

This study analyzes 9 repositories and examines all sampled tests to answer research questions.The data set resulting from our analysis of all 217 property-based tests is also available[6]. We present the results of our qualitative analysis below, organized by research question.

Table 2: Selected Repositories Tests Ratio

Repository Name	**Total PBT**	**Other tests**	**PBT Percentage**
aeson	287	1876	13.3 %
attoparsec	70	0	100 %
containers	1171	423	73.5 %
hashable	21	15	58.3 %
lens	25	55	31.3 %
megaparsec	386	573	40.3 %
pandoc-types	23	114	17.2 %
text	492	262	65.3 %
vector	171	2637	6.1 %

3.1 Property-Based Testing

Figure 1 plots the approximate lines-of-code (LOC) for each library's production code. Codebases vary by nearly an order of magnitude, from 2709 LOC in pandoc-types to 72353 LOC in containers. However, LOC alone does not explain PBT uptake: the tiny attoparsec suite is 100% PBT, while the similarly compact hashable mixes property and example tests, and the large containers project still manages a 73% PBT ratio. This suggests that domain characteristics and oracle availability, rather than sheer code size, drive QuickCheck adoption.

Across these libraries we observed a striking variation in the reliance on QuickCheck-style PBT versus conventional example-based tests (unit or integration). attoparsec employ PBT exclusively, eschewing any testCase-style or benchmark tests. At the opposite end of the spectrum, aeson and vector mix heavily in favor of traditional tests. In between, several core libraries show strong but not exclusive adoption of PBT: containers, text, and hashable. The parser combinator library megaparsec and the optics package illustrate a blended approach that leverages PBT for core algebraic laws but retains example-based checks for intricate error-reporting and performance scenarios. Finally, pandoc-types sits nearer to the Aeson/Vector end of the spectrum, reflecting its role as a high-level document AST library.

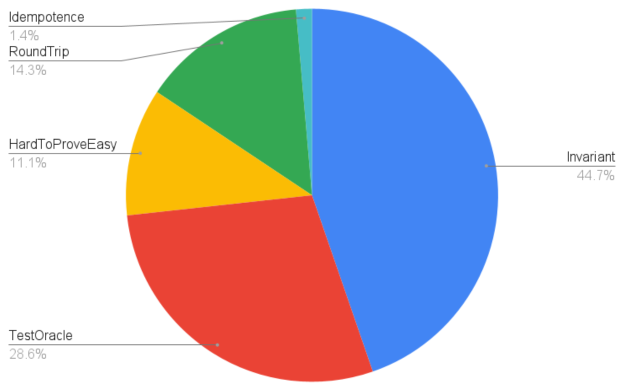

Figure 2: Pie Chart of Property Distribution

Table 3: property-type-distribution

Property Type	Count	%
Invariant	97	44.7 %
Test Oracle	62	28.5 %
RoundTrip	31	14.3 %
Hard to prove, easy to verify	24	11.1%
Idempotence	3	1.4 %
DifferentPaths	0	0.0 %
StructuralInduction	0	0.0 %

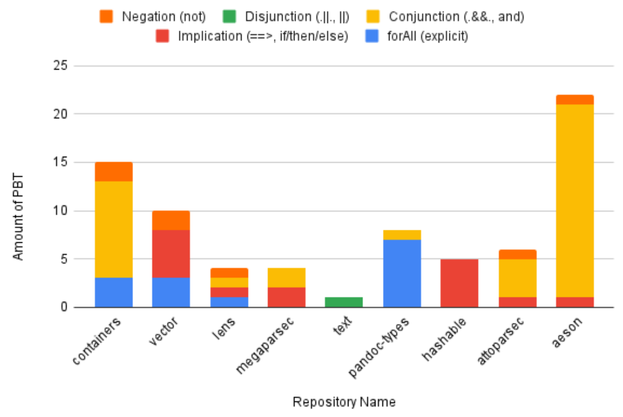

Figure 3: Quantifier and Connectives Distribution per Repository

From these repositories we selected a purposive sample of 217 PBT cases that included custom generators or shrinkers. The distribution is above. As shown in Figure 2 and Table 3: we found real-world QuickCheck practice is highly skewed toward three `high-leverage` idioms. `Invariant` takes account for 48.5% percent, followed by `Test Oracle` and `RoundTrip`. Algebraic laws that are `Hard to prove, easy to verify` contribute another 12%. `Idempotence` is also rare.

We found no clear instances of the `DifferentPaths` and `Structural Induction` patterns. Such tests if present are likely classified under broader invariant or round-trip properties.

These findings reflect the variety of property types developers write, with invariants and test-oracle tests being the most common.
Figure 3 (per-repository breakdown) and Figure 4 (corpus totals) show how often the main QuickCheck quantifiers and logical connectives occur in the 217-property sample.

Conjunction `.&&.` (or *and*) appears 38 times, accounting for almost two-thirds of all explicit logical operators. Authors typically chain two independent checks—for instance, an invariant plus a sanity predicate—rather than embedding deep Boolean logic.

Pre-condition implication is secondary (15 times) and close to the frequency of Explicit quantification(14 times). ==> (or an if . . . then . . . expression that plays the same role) is used 15 times. These properties guard against illegal inputs (e.g. empty vectors, out-of-range indices) before asserting the actual law. The pattern clusters in vector and containers, both of which expose partial operations that would otherwise throw exceptions.

Negation is occasional and disjunction almost absent. Unary `not` shows up 7 times, mainly to express failure branches. We found a single explicit disjunction(||), confirming the community preference for small, conjunctive specifications.

Library-level differences. `aeson` and `containers` contain the bulk of conjunctive clauses, reflecting richer composite laws (e.g. encode–decode round-trip and length preservation).`vector` contributes the most implications, guarding partial index operations. Parsing libraries (megaparsec, attoparsec) hardly use connectives at all—most of their properties are atomic round-trip checks.

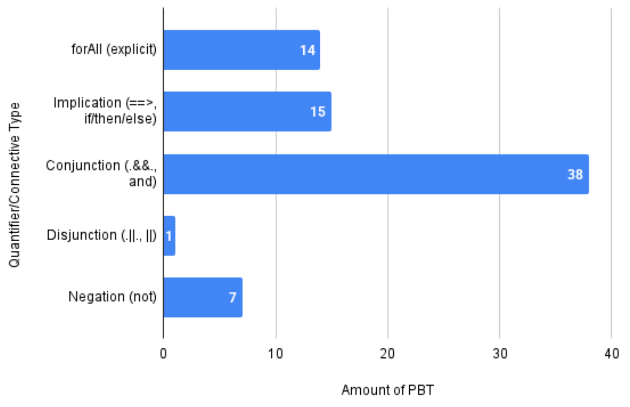

Figure 4: Quantifier and Connectives Distribution in Sampled Dataset

3.2 Generator and Shrinking

Generator and shrinker usage analysis further illustrated developer behavior: in this same 217-test sample, 121 tests (55.8 %) used custom input generators. However, we found the actual amount of defined `instance Arbitrary` was only 12. They, with a few inline `forall` generators are heavily reused by 121 tests.

Custom shrinkers appeared less frequently, and 48 tests (22.12 %) specify custom shrinking logic through forAllShrink or dedicated shrink functions. There was only 4 defined shrinker. Such custom strategies are concentrated in modules handling complex or recursively structured data. By contrast, the remaining two-thirds of tests rely on QuickCheck's default shrinkers, reflecting that shrinker customization remains a secondary priority, possibly due to complexity or perceived low return on investment.

4 Discussion

Building upon the empirical findings presented in Section 3, this section provides a comprehensive discussion of our results. We will systematically address each of the research sub-questions posed in the Introduction, interpreting the implications of our observations regarding property types, logical constructs, and generator/shrinking strategies. A key focus will be on understanding the how and why behind specific PBT application patterns identified in real-world projects. We will also analyze how QuickCheck-based property testing complements other software testing methodologies (e.g. benchmarking, traditional unit testing) within these projects. Finally, we acknowledge the limitations inherent in our study design and data collection, offering insights into potential biases and outlining promising avenues for future research in this domain.

4.1 Analysis of findings

Parsing and low-level data packages (e.g. attoparsec) embrace PBT almost exclusively, leveraging randomized inputs to flush out corner-case bugs in binary/text parsing. Higher-level libraries (aeson, vector, pandoc-types) balance PBT with targeted unit/integration tests. These projects interact with external file formats, I/O layers, and tiny error channels. Therefore, developers can benefit from handwritten tests to validate user-facing behavior (serialization options, exception messages, performance budgets) that QuickCheck's functional assertions cannot easily capture.
Middle-tier packages—containers, text, hashable, megaparsec, lens: PBT dominates the algebraic core, while example tests document delicate corner cases or performance hot-spots.

The data confirms that invariant-style checks form the backbone of QuickCheck practice in the surveyed libraries: nearly one half of the sampled properties simply assert that a transformation leaves some projection of the value unchanged. These micro-invariants are cheap to state, shrink quickly on failure, and rarely require custom generators-hence their popularity.

Round-trip, test-oracle and hard-to-prove-easy-to-verify properties together contribute roughly another third of the corpus patterns.These idioms are attractive, possibly because they allow developers to confirm non-trivial behavior by comparing against either an inverse function, a reference implementation, or an easily checked post-condition. Developers get strong assurances without writing formal proofs and QuickCheck supplies the diverse inputs.

Different-paths and idempotence appear sporadically, reflecting domain-specific needs (e.g. parser branch coverage, duplicate-free updates).
Logical structure mirrors the above: a large majority are single-clause properties and only a small part use an explicit pre-condition (==>) or combine multiple logical conjuncts. Simplicity keeps failure traces small and diagnosis fast.

Generator practice echoes this spectrum of complexity. For simple scalar or list-like types, automatically derived Arbitrary instances suffice. Custom generators concentrate in modules with recursive or highly constrained structures—balanced trees, generalized lens optics, or parser states—where default sampling would otherwise drown in invalid cases. Shrinker customization remains uncommon: only one quarter of properties define a custom shrink, suggesting that developers either accept the quality of QuickCheck's generic shrinkers or think the extra effort disproportionate to the expected debugging benefit.

4.2 Comparison with other environment

As mentioned before, this research is a sub-project of a multi-environment project, there are other our other language / framework pairs analysed. Across all five ecosystems tests tends to rely on a single assertion. Test Oracle patterns dominate in both Haskell and Python. Our 55.8 % rate is around 16 percent higher than Hypothesis and 26 percent above Rust quickcheck. The likely cause is Haskell's recursive, algebraic data structures: developers must hand-craft Arbitrary instances once default derivation fails. By contrast, Hypothesis ships rich built-in strategies, and QuickCheck's derivations handle most flat structures. Custom shrinkers are nearly absent in Hypothesis and jqwik, modestly used in Rust-qc (21 %), and highest in Haskell (22.12 %). QuickCheck's generic shrink tree works for scalars but performs poorly on deeply nested ADTs, motivating hand-written shrink functions. Hypothesis, in turn, embeds advanced tree-search heuristics that remove the need for manual shrinking.

4.3 Threats to Validity

Our qualitative results are based on a hand-random, stratified sample of 217 properties. While we preserved cross-project balance, it may still under-sample infrequent property categories. A different random draw—or a larger sample—could shift the observed proportions by a few percentage points. Future replications should either enlarge the sample or use adaptive sampling to force-include rare categories.

We discovered all properties with a regex-based method that looks for the patterns `prop_...`, `testProperty ...`, and `property $`. It ignored Template-Haskell splices, run-time registration via tasty, and properties hidden behind CPP flags. Though we tried to manually verify statistics, consequently, a small number of tests may have been missed. The under-count was severe enough in bytestring to force its exclusion (as mentioned in Section 2.1). Future work should replace the regex heuristic with a GHC API or HIE-AST pass to capture dynamically generated properties.

Our assigning PropertyType labels is partly subjective. Assigning each property a single Property-Type label involved judgement and may introduce subjectivity.

```haskell
prop_safeIndex :: Vector Int -> Int -> Property
prop_safeIndex v i =
  not (V.null v) ==>
    V.!? v i === fn (V.toList v) i
  where
    fn xs j = case drop j xs of
                (x:_) | j >= 0 -> Just x
                _              -> Nothing
```

Listing 1: Safe index property (vector/tests/Property_vector.hs)

We deliberately counted each property once, using the rule mentioned in Section 2.2, to avoid inflating percentages. This means our category totals represent the primary testing intention developers expressed, not an exhaustive set of behaviors each property happens to cover. The trade-off buys clarity at the cost of losing a small amount of nuance. Future work that explores multi-label or hierarchical coding could quantify those overlaps more precisely.

Beyond interpretive subjectivity, the coding process involves mundane yet fallible data entry: copying a property's file path, filling "custom generator" column, or pasting the shrinker flag. Although we double-checked rows and an audit sample was spot-verified, occasional slips (e.g. mis-counting a forAll occurrence or overlooking a locally defined shrink function) cannot be ruled out. Given the small absolute numbers in our sample, even a handful of such slips could impact percentages by one or two points.

We mitigate this risk by releasing the full coded data csv file, and inviting replication studies to re-label inconsistencies. In our judgement the residual error margin is not big enough to overturn the qualitative patterns reported, but readers should treat the exact percentages as approximate rather than absolute.

Finally, we examined only the latest release of nine high-impact libraries. Smaller domain-specific projects and historical evolutions of properties were out of scope, limiting external generalizability and obscuring temporal trends. Future work that mines additional application-level code and tracks property churn across commits would strengthen the longitudinal picture of QuickCheck practice.

5 Conclusions and Future Work

This paper has presented an empirical investigation into the practical application of Property-Based Testing with QuickCheck in real-world Haskell open-source projects. By analyzing a diverse set of influential libraries, we have explored the prevalent patterns, strategies, and challenges associated with PBT adoption in industrial-strength software development. Our findings confirm the widespread use of equivalence and invariant properties, the strategic deployment of custom generators and shrinking for complex data types and scenarios, and the complementary nature of PBT alongside other testing strategies. This research contributes a valuable empirical foundation to the understanding of effective PBT practices, offering actionable insights for developers aiming to leverage the full power of QuickCheck in their own projects.

5.1 Answers to the Research Questions

RQ1: What types of properties are typically tested using QuickCheck?

Property-based testing in Haskell library development is dominated by a handful of canonical patterns—particularly RoundTrip, Invariant, and Test Oracle, together three idioms account for around 87.5%. This suggests that QuickCheck excels when a simple oracle or inverse exists, echoing user studies that highlight the "high-leverage idioms" nature of these patterns[7].

RQ2: Which types of quantifiers and logical connectives are used in these properties?

Properties are normally a single equality or Boolean clause; only one in five carries an explicit pre-condition and barely one in ten chains multiple predicates. Practical value instead of theoretical elegance—drives this pattern choice.

RQ3: How does property-based testing complement other testing strategies such as unit tests?

Our data show a pragmatic division of labor rather than an outright replacement: When an API is pure and deterministic, developers prefer QuickCheck. A single invariant or roundtrip property explores thousands of cases that would be impractical to enumerate manually. When correctness depends on I/O side-effects, performance envelopes, or precise error messages, authors fall back on illustrative tests, sometimes derived from shrunk counter-examples—to lock in the behavior.

RQ4: How and when are generators implemented?

For most properties, QuickCheck's defaults will be sufficient. But in domains with recursive or highly-constrained data, default sampling would drown in invalid cases and thus custom generator is needed.

RQ5: In which scenarios do developers explicitly define shrinking strategies?

Custom shrinkers are less authored but once necessary, it's always in the same recursive domains that need custom generators. They are indispensable to achieving meaningful random exploration and efficient failure minimization.

5.2 Future research directions

From our data we can say that a few custom generator defined can cover a huge scope of property. Thus we think investigating approaches to derive custom generators automatically from data type definitions will be worthy to reduce developer effort.

Besides, only one quarter of the properties bother with a custom shrinker, yet many blog posts claim that good shrinking is a life-saver when a test fails. Conduct user studies on debugging time saved by custom shrinkers versus default, to quantify practical impact.

References

[1] Antonios Barotsis. *Property-Based Testing in Open-Source Rust Projects: A Case Study of the proptest Crate*. Bachelor Thesis, Delft University of Technology.

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

[3] Arthur Lisboa Corgozinho, Marco Tulio Valente, and Henrique Rocha. How developers implement property-based tests. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 380–384. IEEE, 2023.

[4] David de Koning. *Property-Based Testing in Practice using Hypothesis: In-depth study on how developers use Property-Based Testing in Python using Hypothesis*. Bachelor Thesis, Delft University of Technology, 2025.

[5] Max Derbenwick. *Property-Based Testing in Rust, How is it Used?: A case study of the quickcheck crate used in open source repositories*. Bachelor Thesis, Delft University of Technology.

[6] Max Derbenwick, Harald Toth, David de Koning, Antonios Barotsis, Ye Zhao, Andreea Costea, and Sára Juhošová. Property-based testing in the wild! 4TU.ResearchData, 2025. doi: 10.4121/368f63ab-10fc-4603-a15a-bde25e72e778.

[7] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. Property-based testing in practice. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*, 2024.

[8] Harrison Goldstein, Jeffrey Tao, Zac Hatfield-Dodds, Benjamin C. Pierce, and Andrew Head. Tyche: Making sense of pbt effectiveness. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, UIST '24, New York, NY, USA, 2024. Association for Computing Machinery.

[9] Harald Toth. *Property-Based Testing in the Wild!: Exploring Property-Based Testing in Java: An Analysis of jqwik Usage in Open-Source Repositories*. Bachelor Thesis, Delft University of Technology, 2025.