Evaluating PBT Frameworks in OCaml

ABSTRACT

Property-based testing (PBT) is an effective way of finding bugs in programs by automatically generating test cases to check user-defined properties. It is especially powerful for testing functional codebases, where it exploits immutability, purity, and the strong typing information available. Although the PBT space contains a wide variety of frameworks with a plethora of approaches to generating inputs, there is a lack of tools that compare the effectiveness of the frameworks. One such tool, ETNA [6], was recently presented to empirically evaluate and compare PBT techniques in various frameworks, focusing on the Haskell and Coq testing ecosystems. This research extends ETNA to OCaml, replicating case studies from the Haskell and Coq literature. We compare the bug-finding capabilities of three popular QuickCheck-style OCaml PBT frameworks as well as one AFL-based fuzzing approach and describe current work in tackling constrained OCaml properties via Coq by leveraging prior research on automatic generators for QuickChick [3].

1 INTRODUCTION

Property-based testing was introduced and popularized in Haskell through QuickCheck [1], allowing users to write executable specifications of their code and have them checked on a large number of automatically generated test cases. This approach is extremely useful in ensuring software reliability by discovering edge cases and unexpected behaviors that traditional unit testing might miss.

Various factors influence the effectiveness of PBT, such as the quality of the properties, but arguably the most critical factor is the variety of ways in which test cases are generated, which allows property-based testing to gain its advantage over traditional testing solutions and more accurately test the limits and edges of a user's code base. Many frameworks are inspired by QuickCheck, which uses randomized generation, though alternatives like input space enumeration and feedback-guided generation are also gaining traction. The style of generation can significantly affect the result of a PBT system. Even once a generation style is chosen, though, there is a diverse variety of frameworks for users to choose from. For example, Haskell has QuickCheck and Hedgehog; OCaml has QCheck and Crowbar, to name a few, each with its own techniques for dictating how inputs are generated.

Once the choice of framework and generation style is complete, the user still has countless options to consider when writing their generators. From input size to input shape to the values themselves, the user must blindly experiment with several trials to find one that best suits their task. Until recently, there were various existing performance evaluations, but a lack of comparisons between evaluations. ETNA introduced a system to empirically evaluate various generators across different frameworks to allow users to finally approach PBT solutions from a leveled playing field.

1.1 ETNA Background

A critical point of designing properties for PBT is preconditions. For example, when testing a system of binary search trees, our properties should only apply to valid BSTs, not arbitrary binary trees. A simple solution is to follow the data definition of the tree type to create an arbitrary binary tree, and then filter out those that are not valid BSTs. Shi et al. [6] call this approach *type-based*, as the generation of the test cases is guided by the type definition. However, as the workload becomes more and more sophisticated, this filtering approach falls apart. The chance of a random tree being a valid red-black tree is far smaller. The chance of a random lambda calculus expression being type-correct is even lower. This issue gives rise to *bespoke* generators, designed with the preconditions in mind to only generate valid test cases. As the input space grows in complexity, this approach requires far more user ingenuity, so many approaches lie between the poles of type-based and bespoke generators.

To measure the effectiveness of a generator, ETNA uses the approach of mutation testing, which artificially injects user-defined bugs (referred to as mutants) into the tested system and checks how well the testing can detect them. A *workload* in ETNA contains several properties to test, several mutants to test against, and a few generators to compare against each other. A *task* in ETNA refers to a single mutant applied to the source code and a single property tested to find the mutant. It runs many different tasks with various generators from different frameworks, collecting information on how quickly the bug is found, if at all. It then creates visualizations of the relative effectiveness of these different generators by generalizing the trial execution procedure.

1.2 Expansion to OCaml

This research expands ETNA's domain to OCaml frameworks. In Haskell, most PBT frameworks are almost syntactically identical to QuickCheck [1], such as LeanCheck [4] and SmallCheck [5]. On the other hand, OCaml provides developers with a wide variety of frameworks, with QCheck having a fully-implemented QuickCheck-like monadic system of generators, Crowbar being a simple wrapper to AFL's backend, and Base_quickcheck leveraging the Core standard library replacement. This diversity further motivates the need for evaluation.

This expansion contains the same workloads introduced in the original ETNA paper, implemented in OCaml using QCheck, Crowbar, and Base_quickcheck. Crowbar supports a fully random mode and an AFL-powered mode, both of which are benchmarked. Identical type-based and bespoke generator implementations were compared using properties of binary search trees, red-black trees, and the simply-typed lambda calculus.

In addition, since Coq code can interface with OCaml code via extraction, we implemented a semi-automatic shim for testing OCaml properties using Coq's QuickChick library [2]. Since past research has shown that Coq's support for inductive types can be used to automatically *generate generators* for a given property [3], this also opens the way for interesting future work in targeting OCaml preconditions with automatically derived bespoke generators.

2 RESULTS

Though ETNA supports changing the timing of tasks, we have kept the timeouts consistent with the original ETNA paper at 60 seconds per task. Each task is run on each generator 10 times with the results averaged for consistency. The task bucket charts classify tasks ranging from "solved instantly" to "unsolved" with progressively lighter shades. Figure 1 shows the chart for an example generator that solves 14 tasks within 0.1s but does not solve four other tasks at all.

Figure 1: Example bucket chart, and the bucket ranges.

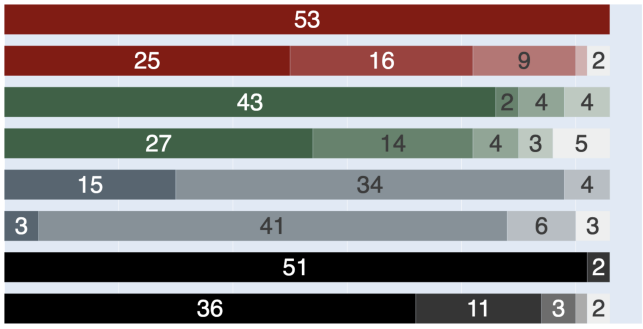

Figure 2: BST.
= QCheck, = Crowbar Fully-random,
= Crowbar AFL, = Base_quickcheck.
The first and second buckets for each framework represent the bespoke and type-based generators respectively.

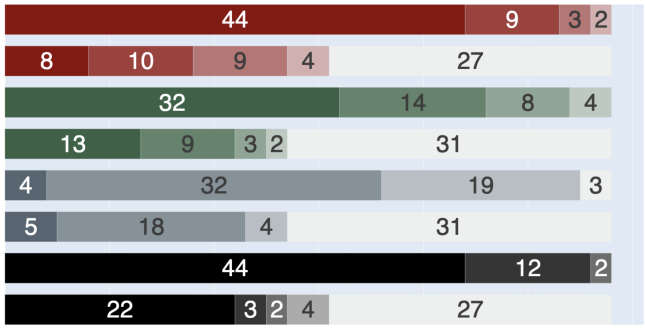

Figure 3: RBT

The results suggest that, for our relatively simple workloads, Base_quickcheck outperforms the other frameworks in almost all situations. For both the BST and RBT workloads, the bespoke generator first generated a list of key-value pairs and then piped them through a correct `insert` function. In our testing, the sizing distribution for the trees was most optimal by default for Base_quickcheck.

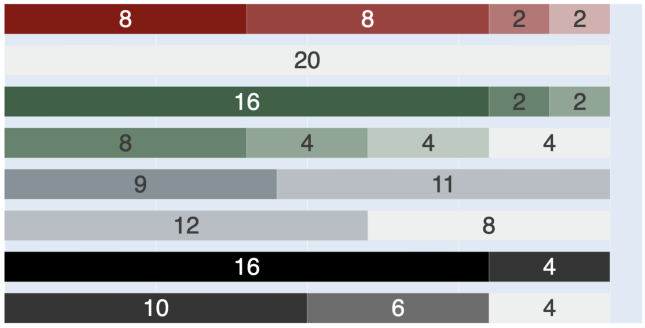

Figure 4: STLC

For the other frameworks, we had to address and fine-tune implementation details by hand to achieve optimal results. QCheck's size distribution was heavily bimodal, with many trees containing one or two nodes, and most others containing several thousand. Without manually tuning the sizing distribution it performed significantly worse. Similarly, Crowbar's source has a hard-coded upper size bound of 100; this causes its in-built `list` generator to only generate useful (not padded with trailing zeros) lists of *up to length six* due to its logarithmically scaling implementation, which could not find the majority of the bugs. We re-implemented a list generator that used Crowbar's `bind` operator to achieve lists of up to length 33, which we used for our evaluation.

Our results show that AFL through Crowbar's interface performed the worst out of all frameworks by over one order of magnitude. This is somewhat expected due to the simplicity of our workloads: AFL's instrumentation draws too much relative overhead. To look into this, we locally explored AFL and found that its instrumentation may not fully work through Crowbar. OCaml's AFL's documentation shows an example that fuzzes a four-character secret string. An OCaml program that used `AflPersistent.run` found the secret nearly ten times faster than an identical one that used `Crowbar.add_test`, suggesting a buggy implementation in Crowbar. We have submitted this, as well as Crowbar's earlier sizing issue, for review.

3 ONGOING WORK

All three presented workloads have properties with many relatively simple edge cases. We plan on designing a workload with a more complex or finite set of edge cases to see if AFL's instrumentation outperforms traditional PBT in such a situation.

We are also working on refining the pipeline allowing us to run Coq's QuickChick on OCaml properties via extraction. This will not only let us benchmark QuickChick's performance but will also allow us to utilize Coq-exclusive inductive types to *generate generators* which can rival bespoke ones [3]. This paper's tooling will allow us to quantitatively evaluate the Coq extraction overhead and the "free" generators' performances.

REFERENCES

[1] Koen Claessen and John Hughes. 2011. QuickCheck: a lightweight tool for random testing of Haskell programs. *SIGPLAN Not.* 46, 4 (may 2011), 53–64. https://doi.org/10.1145/1988042.1988046

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

[3] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2017. Generating good generators for inductive relations. *Proc. ACM Program. Lang.* 2, POPL, Article 45 (dec 2017), 30 pages. https://doi.org/10.1145/3158133

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

[5] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. *SIGPLAN Not.* 44, 2 (sep 2008), 37–48. https://doi.org/10.1145/1543134.1411292

[6] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C. Pierce, and Leonidas Lampropoulos. 2023. Etna: An Evaluation Platform for Property-Based Testing (Experience Report). 7, ICFP, Article 218 (aug 2023), 17 pages. https://doi.org/10.1145/3607860