QuickerChick

IVAN MLADENOV, University of Maryland, USA
ALPEREN KELES, University of Maryland, USA
LEONIDAS LAMPROPOULOS, University of Maryland, USA

Property-based testing (PBT) with QuickChick relies on extracting Rocq programs to OCaml. However, this extraction mechanism, while crucial for QuickChick to function, has significant performance implications. In this work, we describe how we optimized QuickChick, exploiting various opportunities offered by program extraction to significantly improve each test's extraction, compilation, and running time. We also evaluate these improvements using the ETNA benchmarking platform for PBT to assess how individual improvements impacted the overall test run time.

CCS Concepts: • **Software and its engineering → Software verification and validation**.

Additional Key Words and Phrases: Property-based testing, Rocq, OCaml, Program Extraction

1 Introduction

QuickChick [1] is the canonical library for property-based testing in Rocq. A programmer writes a property that they expect should hold, and QuickChick generates test cases to verify said property. Normally, a PBT library will run these tests in the language in which the tests were written — but not QuickChick! That's what makes QuickChick unique.

QuickChick needs to use a mechanism called *extraction* [2] to be able to execute tests. Rocq, as a proof assistant, lacks some of the side-effecting capabilities that are necessary for random testing (such as randomness or IO). Extraction allows QuickChick to convert a test written in Rocq directly into (hopefully) equivalent OCaml code, which can then be compiled and executed with its results reported back to the user within an interactive proving session. For example, consider the following simple property, which checks that the result of a `max` function is greater than both of its inputs:

```
Definition max (x : nat) (y : nat) : nat := if x <=? y then y else x.

Conjecture prop_maxGe: forall (x y : nat), max x y >= y /\ max x y >= x.

QuickChick prop_maxGe.
```

When a user runs the QuickChick command on the last line, they will get some output like this:

```
> QuickChecking prop_maxGe
>
> +++ Passed 10000 tests (0 discards)
```

Under the hood, this property is extracted to a monolithic OCaml file, which is then compiled to run the tests. Let's look at the size of said file:

```
$ wc -l prop_maxGe.ml
    2760 prop_maxGe.ml
```

Authors' Contact Information: Ivan Mladenov, University of Maryland, College Park, MD, USA; Alperen Keles, University of Maryland, College Park, MD, USA; Leonidas Lampropoulos, University of Maryland, College Park, MD, USA.

How does a single property end up generating such a large file? For the OCaml program to run, every single dependency—whether from QuickChick itself, the standard library, or the core library—of the property must also be extracted to the file. This introduces some obvious overhead. For one, the extraction scheme must recursively walk all library dependencies, making the extraction time longer. On top of that, every extracted dependency must be compiled to execute tests, i.e., building these large singleton files wastes time.

There's a silver lining to all of this: many of these dependencies are frequently reused functions and type definitions, which means one could pre-compile them into a library and link against it. This is precisely what we have done.

2 An Extracted Library

We developed an internal library for QuickChick, which contains OCaml equivalents to many of QuickChick's commonly used dependencies. Having these library definitions gave us many opportunities to squeeze out better performance.

The extraction library in Rocq allows users to hijack extraction by replacing terms or types directly with raw strings of OCaml code. This lets us replace any dependencies with direct references to our library definitions, ultimately shortening the recursive walk. With the example from before, we can see how much the file shrank by inlining our library functions:

```
$ wc -l prop_maxGe.ml
     218 prop_maxGe.ml
```

From our experiments, we saw a 1.5-3× improvement in extraction, along with a roughly 1.2× improvement in compile time. So despite the drastic reduction in file size, the compile time didn't reflect these changes as much as we'd hoped. As it turns out, the OCaml compiler is incredibly effective; rather, the slowdown was in the call to *ocamlbuild*. We thus changed the build to directly compile with *ocamlopt* and explicitly link against the library, resulting in a >2× improvement in compile time. We can see the improvements already through ETNA [4] experiments for binary search trees (BST), red-black trees (RBT), and the simply-typed lambda calculus (STLC):

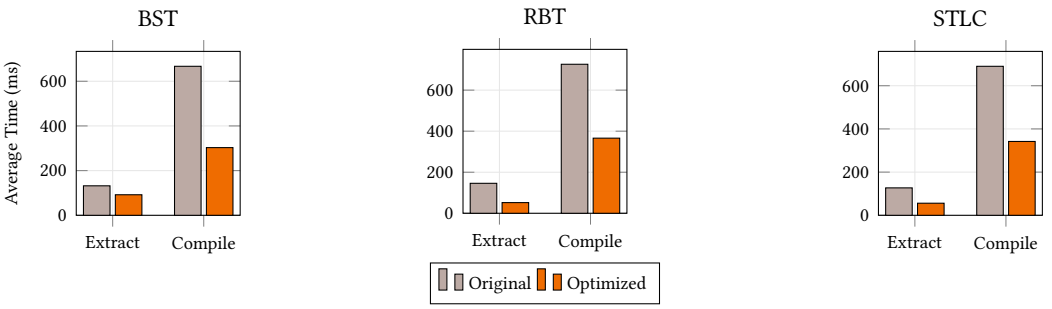

Fig. 1. Average extraction and compilation times (ms) for BST, RBT, and STLC benchmarks (ETNA)

Writing this library also gave rise to an opportunity not previously available: tail-call optimizations. Now that we're writing in OCaml rather than Rocq, we could rewrite naively recursive functions and let the compiler optimize them out. This change alone resulted in 1.2-1.8× improvement in run time. While looking for these inefficiencies, we realized that the function responsible for computing test case sizes was too expensive for how frequently it's called. Without significantly changing the distribution of test sizes, we were able to get yet another 2× speedup.

Lastly, it was worthwhile to experiment with different optimization levels and compilers. QuickChick was already using the native compiler, but we wanted to see how a compiler like *flambda* [3] would perform. Evidently, the compilation time was slightly worse with the aggressive optimizations. The question was whether the run-time benefits outweighed the compile-time penalty. The answer was a resounding yes. Because we're already pre-compiling the library functions, and on top of that we've cut down on the actual file size, the time spent compiling one property test is negligible. With that, we get a free performance boost in run time, another 1.5× speedup on average.

We can zoom out now and see what all of these run-time improvements amount to:

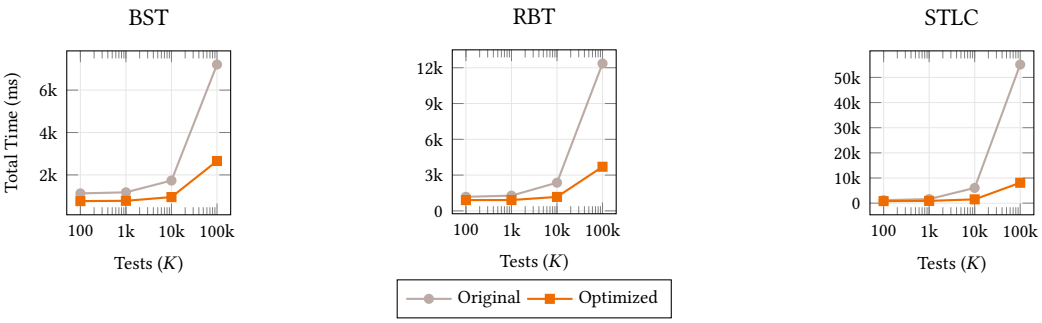

Fig. 2. Average total runtime (ms) vs test counts for BST, RBT, and STLC benchmarks (ETNA), semi-log

With all of our optimizations in hand, we achieved a 3-7× overall speedup for QuickChick. In all benchmarks, even with small test cases, the optimized library finished tests faster. No matter the price paid for compilation time, the overall time improvements lead to a snappier testing experience. Where previously executing 100,000 tests of certain properties could take close to a minute, with the combination of our optimizations, the same testing runs can finish in under 10 seconds.

3 Conclusion and Future Work

Ideally, we'd have a 1-1 equivalent of all QuickChick library functions written, hand-optimized, and compiled in OCaml. Because this library is extensible, we can keep translating Rocq functions to approach this goal, in the meantime improving performance and the user experience. Ultimately, this extracted library lays the groundwork for a better-performing QuickChick, where the gap between testing verified programs and efficient execution is narrowing.

References

[1] Leonidas Lampropoulos and Benjamin C. Pierce. 2018. *QuickChick: Property-Based Testing in Rocq.* https://softwarefoundations.cis.upenn.edu/qc-current/index.html

[2] Pierre Letouzey. 2002. *A New Extraction for Coq.* https://www.irif.fr/~letouzey/download/extraction2002.pdf

[3] ocaml. 2025. *OCaml - Optimization with Flambda.* https://ocaml.org/manual/5.4/flambda.html

[4] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C. Pierce, and Leonidas Lampropoulos. 2023. Etna: An Evaluation Platform for Property-Based Testing (Experience Report). *Proc. ACM Program. Lang.* 7, ICFP, Article 218 (Aug. 2023), 17 pages. doi:10.1145/3607860