QuickChick: Property-Based Testing for Coq

Advisor: Cătălin Hriţcu ⟨catalin.hritcu@gmail.com⟩

Institition: INRIA Paris-Rocquencourt, Prosecco Team

Location: 23 Avenue d'Italie, Paris, France

Language: English

Existing skills or strong desire to learn:

- functional programming (e.g. OCaml or Haskell),
- property-based testing (e.g. QuickCheck),
- interactive theorem proving in the Coq proof assistant,
- optional: SSReflect, logic programming, constraint programming, probabilistic programming

Research Context

Designing complex systems that provide strong safety and security guarantees is challenging (e.g. programming languages, language compilers and runtimes, reference monitors, operating systems, hardware, etc). Proof assistants such as Coq (The Coq team, 1984-now) are invaluable for showing formally that such systems indeed satisfy the properties intended by their designers. However, carrying out formal proofs while designing even a relatively simple system can be an exercise in frustration, with a great deal of time spent attempting to prove things about broken definitions, and countless iterations for discovering the correct lemmas and strengthening inductive invariants.

The long-term goal of this project[1] is to reduce the cost of producing formally verified systems by integrating property-based testing (PBT) with the Coq proof assistant. Ideally, our solution will achieve the best of testing and proving, by producing easily understandable counterexamples and guiding users towards correct system designs and corresponding formal evidence of their correctness. The use of PBT will dramatically decrease the number of failed proof attempts in Coq developments by allowing users to find errors in definitions and conjectured properties early in the design process, and to postpone verification attempts until they are reasonably confident that their system is correct. PBT will also help during the verification process by quickly validating proof goals, potential lemmas, and inductive invariants. Our solution will provide automation of common patterns, yet keep the user fully in control. These improvements will lower the barrier to entry and increase adoption of the Coq proof assistant. Moreover, integrating PBT with Coq will provide an easier path going from systematic testing to formal verification, by encouraging developers to write specifications that can be only tested at first and later formally verified. It will also allow PBT users to verify that they are testing the right properties and to evaluate the thoroughness of their testing. Achieving all this requires improvements to the state-of-the-art both in PBT and formal verification research, which we discuss in the next section.

While property-based testing has already been integrated with relative success into other proof assistants such as Isabelle (Bulwahn, 2013) and ACL2 (Chamarthi et al., 2011), the logic of Coq is much richer, which raises additional challenges. Also, these previous efforts were aimed at full automation, leaving no space for user customization or interaction, which is, in our experience, crucial for thorough testing that finds interesting bugs and drives the design and verification of nontrivial systems.

As a necessary first step towards the final goal of this project we have ported the QuickCheck framework (Claessen and Hughes, 2000; Hughes, 2007) from Haskell to Coq, producing an prototype Coq plugin

[1] This project is a collaboration between Cătălin Hriţcu and Maxime Dénès from INRIA Paris-Rocquencourt, Benjamin Pierce and Leonidas Lampropoulos from University of Pennsylvania, John Hughes from Chalmers, and Zoe Paraskevopoulou from ENS Cachan.

called QuickChick.[2] There are, however, important remaining challenges and research opportunities that are unique to integrating property-based testing in the Coq theorem prover. The scientific objectives of this project consist in addressing these challenges and seizing these opportunities. We will measure success by performing several realistic case studies.

Scientific Objectives

Challenge: significant effort required for using testing during Coq verification.

Currently, a QuickChick user has to write efficiently executable variants of the proof-oriented artifacts she uses for verification, and to formally relate the two via additional Coq proofs. The artifacts that have to be given equivalent efficient implementations include both the system (e.g. a type system, a dynamic monitor) and the properties under test (e.g. progress, preservation, noninterference). We call the executable variant of the property under test a *checker* for that property. Checkers are, however, not sufficient for testing conditional properties with sparse pre-conditions; for instance generating random lists and then filtering out the ones that are not sorted leads to extremely inefficient testing. In such cases the user has to additionally provide *property-based generators* that efficiently generate only data satisfying the sparse pre-conditions (e.g. only sorted lists). Our aim is to decrease the human effort required for using testing during the normal Coq proving process by automating the tedious but boring parts of these tasks.

Objective 1: streamline relating declarative and efficiently executable artifacts. We will devise convenient proof techniques for showing the equivalence between declarative and efficiently executable artifacts. For this we will focus on small-scale reflection proofs, as supported by the SSReflect extension to Coq (Gonthier and Mahboubi, 2009). However, while traditional SSReflect proofs use evaluation to remove the need for some reasoning in small proof steps, the objects defined in the SSReflect library and used in proofs are often not fully and efficiently executable. We believe we can overcome this limitation by exploiting a recent refinement framework by (Cohen et al., 2013; Dénès et al., 2012), which allows maintaining a correspondence and switching between proof-oriented and computation-oriented views of objects and properties. In some special cases it is furthermore possible to use the declarative artifact (e.g., an inductive definition that describes a logic program) to automatically generate the executable implementation together with its correctness proof (Berghofer and Nipkow, 2002; Delahaye et al., 2007; Tollitte et al., 2012). We plan to integrate these existing techniques into QuickChick, but our main objective is a general and easy-to-use proof framework for relating user-defined declarative and efficiently executable artifacts that are arbitrarily far apart from each other.

Objective 2: a language for property-based generators. Writing property-based generators by hand is very effective, but it often requires duplicating the structure of the corresponding checkers' code, after which the two can easily run out of sync, leading to testing bugs. We have recently started designing a new domain-specific language, in the style of lenses (Hofmann et al., 2012), for writing property-based generators. An expression in this language denotes both a checker and a property-based generator for the same property. For this we extend the syntax of propositional logic with algebraic datatypes, pattern matching, and structural recursion. In order to support negation we make the semantic interpretation of an expression be a pair of complementary probability distributions. We are working on devising an efficient evaluation engine that exploits the logical structure of the statement to minimize the amount of backtracking and uses laziness to exploit sharing between computations. Once this is done we will integrate our language into Coq using the powerful notation mechanism. We will also explore various approaches for controlling the obtained probability distributions. The closest related work in this space is a recent framework by Claessen et al. (2014) for generating constrained random data with uniform distribution. We will improve on that work by allowing the user to customize the probability distribution, by providing better efficiency for generating simply-typed λ-calculus terms, and by scaling up to generating data satisfying more intricate properties, as needed for the case studies discussed below: indistinguishable abstract machine states (as required by noninterference), dependently-typed terms, etc.

Opportunity: integrating testing within Coq enables proving formal statements about the testing itself. Objective 1 is mainly targeted at automating proofs; the QuickChick user will in most cases still have to write efficient checkers and property-based generators. Objective 2 is targeted at allowing the user to write both a

[2] `https://github.com/QuickChick`

checker and a generator as one single program with dual semantics. Despite this help, the QuickChick user can still make mistakes that result in testing the wrong property. Moreover, the code written by the user will depend on the correctness of the sophisticated execution engine from Objective 2 and of the QuickChick framework itself. Testing errors can conceal important bugs and thus reduce its benefits, and are especially hard to find and debug in the presence of randomness—our generators are probabilistic programs. The fact that all testing code written by the user and the large majority of our code is written in Coq itself enables us to prove its correctness. In a recent experiment (Paraskevopoulou and Hriţcu, 2014) we devised a prototype verification framework on top of QuickChick in which one can show formally that checkers and property-based generators are correct with respect to the declarative properties they are supposed to test. The main novelty is that we provide a systematic way to replace complex reasoning about probabilities with reasoning solely about the *set of outcomes* a generator can produce with non-zero probability. We have used this verification methodology to prove the correctness of most QuickChick combinators, with respect to the (axiomatic) set of outcomes semantics of a small number of primitive ones. We have also applied our methodology on a red-black tree example and made good progress on a more complex noninterference example. These encouraging preliminary results indicate that this verification methodology is modular, scalable, and requires minimal changes to existing code.

Objective 3: automate verification of user checkers and generators. Verification in our framework is at the moment a manual process, still we believe the sets of outcomes abstraction is highly suitable for automatic verification. We will develop specialized Coq tactics for automating the most common patterns occurring in our proofs. We will achieve further automation using SMTCoq, an interface for calling SMT solvers in Coq without compromising soundness (Keller, 2013).

Objective 4: verify the QuickChick implementation. Beyond the case studies below, we will also apply our verification framework to the QuickChick implementation itself, producing the first formally verified PBT framework. We will start in the sets of outcomes model and highly increase the automation of our current proofs using the results of Outcome 3. We will also reduce the number of primitives that are given an axiomatic semantics to the absolute minimum. We will extend the verification to also cover counterexample shrinking. More ambitiously, we will repeat this verification while taking into account probabilities. Since it was first released in 2000 and until very recently QuickCheck used the splittable random number generator in Haskell that generates highly correlated randomness.[3] The fix proposed by Claessen and Pałka (2013) is based on a provable cryptographic construction; however their proofs are only on paper. As a stretch goal our verification will include a Coq proof of Claessen and Pałka's construction using the CertiCrypt framework for cryptographic proofs (Barthe et al., 2009).

Objective 5: verify the execution engine of the property-based generator language from Objective 2. We will verify the correctness of the efficient execution engine for generators from Objective 2, again, first with respect to the sets of outcomes abstraction, and then in a probabilistic setting.

Opportunity: systematic evaluation of testing quality. The integration with Coq raises new opportunities for evaluating and improving the effectiveness of PBT. While the verification framework above guarantees that the checkers and generators are testing the right property, it does not ensure that they are testing it *well*. While gathering statistics and checking code coverage are useful techniques for estimating the quality of testing, we want to add a new, more systematic, and potentially more reliable technique for this.

Objective 6: devise polarized mutation testing framework. The idea is to use the declarative Coq definitions to systematically mutate the artifact under test to introduce all pointwise bugs from an interesting class and to make sure that they are all found by testing. Once all introduced bugs are found and no new bugs are discovered in the non-mutated artifact we can obtain higher confidence that indeed no bugs are left and we can start proving. The main novelty over previous mutation testing work is that instead of blindly introducing syntactic changes that do not necessarily violate the tested property and waste precious human effort weeding them out, we only introduce real bugs by exploiting the logical structure of the Coq property and the declarative description of the artifact. If the tested property is tight then strengthening the predicates appearing in positive positions or weakening the predicates in negative ones is guaranteed to only introduce real bugs. For instance, we can add bugs to type progress by strengthening the step relation (e.g. dropping whole stepping rules) and to noninterference by weakening it (e.g. dropping information-flow side-conditions). Similarly, we can break type

[3] `https://ghc.haskell.org/trac/ghc/ticket/3620`

preservation either by strengthening the occurrence of the typing relation in the conclusion, or by weakening the occurrence in the premise. Initial experiments with manually introducing bugs in an information-flow abstract machine (Hriţcu et al., 2013), the simply-typed λ-calculus, and CompCert have been very encouraging. We discovered for instance that a generator for simply-typed λ terms (Pałka et al., 2011) was not generating shadowed variables, which caused it to miss a capturing bug in substitution. We also discovered that CSmith, a highly successful tool for testing C compilers (Yang et al., 2011), could not detect a bug we introduced in CompCert, causing it to perform tail-call optimization even for non-tail-recursive functions. Beyond working out some of these case studies in detail, the goal is to develop a general methodology and theory of polarized mutation testing, and to integrate it with the rest of QuickChick. We will also use this methodology to assess the success of our language for property-based generators (Objective 2).

Case Studies. We will use a series of realistic case studies to assess the success of our project.

Objective 7: test security monitors. In a separate line of research (Azevedo de Amorim et al., 2014a,b), we are developing a novel framework for hardware-assisted security monitors. While our goal is to produce verified monitors for important security properties such as noninterference, memory safety, isolation, control-flow integrity,etc., designing such monitors is hard, and our initial experiments have shown that testing can help speed up and guide the design process (Hriţcu et al., 2013).

Objective 8: test complex type-checkers including Coq and F*. We will start with the simply-typed λ-calculus because this will enable us to compare against previous work—e.g., by Claessen et al. (2014)—then we will try to extend our work to dependent types. In particular we will try to use the language-based approach from Objective 2 to devise the first efficient generators for dependently-typed terms, currently an open challenge. For this we will target the formalization of the Calculus of Constructions (CC) in Coq by Barras and Werner (1997). Because CC is a subset of Coq, generating CC terms will allow us to test the implementation of Coq itself, and to catch simple bugs that affect the soundness of Coq once in a while.[4] We will also try to apply our techniques to F*, a sophisticated type-checker based on dependent and refinement types (Swamy et al., 2013).

References

A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 165–178. ACM, Jan. 2014a.

A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: A framework for verified, tag-based security monitors. Under Review, Nov. 2014b.

B. Barras and B. Werner. Coq in Coq. Technical report, 1997.

G. Barthe, B. Grégoire, and S. Z. Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 90–101, 2009.

S. Berghofer and T. Nipkow. Executing higher order logic. In *International Workshop on Types for Proofs and Programs (TYPES)*, volume 2277 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2002.

L. Bulwahn. *Counterexample Generation for Higher-Order Logic Using Functional and Logic Programming*. PhD thesis, Technische Universität München, Feb. 2013.

H. R. Chamarthi, P. C. Dillinger, M. Kaufmann, and P. Manolios. Integrating testing and interactive theorem proving. In *10th International Workshop on the ACL2 Theorem Prover and its Applications*, volume 70 of *EPTCS*, pages 4–19, 2011.

K. Claessen and J. 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.

[4] `https://sympa.inria.fr/sympa/arc/coqdev/2014-04/msg00052.html`

K. Claessen and M. Pałka. Splittable pseudorandom number generators using cryptographic hashing. In *ACM SIGPLAN Symposium on Haskell*, pages 47–58. ACM, 2013.

K. Claessen, J. Duregård, and M. H. Pałka. Generating constrained random data with uniform distribution. In M. Codish and E. Sumii, editors, *Functional and Logic Programming*, volume 8475 of *Lecture Notes in Computer Science*, pages 18–34. Springer International Publishing, 2014.

C. Cohen, M. Dénès, and A. Mörtberg. Refinements for free! In *3rd International Conference on Certified Programs and Proofs (CPP)*, pages 147–162, 2013.

D. Delahaye, C. Dubois, and J.-F. Étienne. Extracting purely functional contents from logical inductive types. In *20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 4732 of *Lecture Notes in Computer Science*, pages 70–85. Springer, 2007.

M. Dénès, A. Mörtberg, and V. Siles. A refinement-based approach to computational algebra in Coq. In *3rd International Conference on Interactive Theorem Proving (ITP)*, volume 7406 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2012.

G. Gonthier and A. Mahboubi. A small scale reflection extension for the Coq system. Technical Report Research Report Number 6455, Microsoft Research INRIA, July 2009.

M. Hofmann, B. C. Pierce, and D. Wagner. Edit lenses. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 495–508, 2012.

C. Hriţcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos. Testing noninterference, quickly. In *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Sept. 2013.

J. Hughes. QuickCheck testing for fun and profit. In *9th International Symposium on Practical Aspects of Declarative Languages (PADL)*, volume 4354 of *Lecture Notes in Computer Science*, pages 1–32. Springer, 2007.

C. Keller. *A Matter of Trust: Skeptical Communication Between Coq and External Provers*. PhD thesis, École Polytechnique, June 2013.

M. H. Pałka, K. Claessen, A. Russo, and J. 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.

Z. Paraskevopoulou and C. Hriţcu. A Coq framework for verified property-based testing. Internship Report, Inria Paris-Rocquencourt, Sept. 2014.

N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. *Journal of Functional Programming*, 23(4):402–451, 2013.

The Coq team. *The Coq Proof Assistant*, 1984-now. Version 8.4.

P.-N. Tollitte, D. Delahaye, and C. Dubois. Producing certified functional code from inductive specifications. In *Second International Conference on Certified Programs and Proofs (CPP)*, volume 7679 of *Lecture Notes in Computer Science*. Springer, 2012.

X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 283–294. ACM, 2011.