QuickChick: Property-Based Testing for Coq

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

INRIA Team: Prosecco

Location: 23 Avenue d'Italie, Paris, France

Language: English (no French)

Existing skills or strong desire to learn:

- Coq proof assistant,
- property-based testing (e.g. QuickCheck),
- functional or logic programming.

Context

Devising mechanisms for enforcing strong safety and security properties is often very challenging. This makes it hard to be confident in the correctness of such mechanisms without fully machine-checked proofs; e.g., in the Coq proof assistant [1]. However, carrying out these proofs while designing the mechanisms can be an exercise in frustration, with a great deal of time spent attempting to verify broken definitions, and countless iterations for discovering the correct lemmas and strengthening inductive invariants. To prevent failed proof attempts and thus reduce the overall cost of producing a formally verified enforcement mechanism, we advocate the use of *property-based testing* for systematically finding counterexamples. Property-based testing can find bugs in definitions and conjectured properties early in the design process, postponing proof attempts until we are reasonably confident that the design is correct. Moreover, testing can be very helpful during the proof process for quickly validating potential lemmas, inductive invariants, or simply the current proof goal. In order for this methodology to work well though, testing has to be integrated with the proof assistant. The long term goal of this line of research is to devise QuickChick, the first property-based testing framework for Coq. We believe that such a tool could considerably improve the proving experience in Coq, and thus have a significant and long lasting impact on the growing Coq community.

As a first step in this direction, I collaborated with researchers from University of Pennsylvania, Chalmers, and MSR Cambridge, on building the initial prototype of a random testing framework for Coq similar to QuickCheck [14]. We used this framework to test noninterference for increasingly sophisticated dynamic information flow control (IFC) mechanisms for low-level abstract machines [22]. These case studies showed that property-based random testing can quickly find a variety of missing-taint and missing-exception bugs, and incrementally guide the design of a correct version of the IFC mechanisms. We found that both the strategy for generating random programs and the precise formulation of the noninterference property are critically important for good results.

We plan to extend this initial prototype, building on the existing research literature on property-based testing (e.g., random testing [6, 9, 14, 19], exhaustive testing with small instances [9, 23, 29], narrowing-based testing [9, 13, 26, 29], etc.), as well as related work on generating executable code from inductive definitions [4, 5, 16, 30], automatically generating data that satisfies the preconditions of the tested

property [10,12,20], integrating testing and proof automation [28], exploiting model finders [7,31], etc. However, our ambition goes beyond bringing some of these existing techniques to Coq and integrating them into a cohesive testing framework. Our research goal is to extend the state of the art in property-based testing by working out several promising new ideas that could become QuickChick's killer features. Two of these ideas, smart mutation and the deep integration with Coq/SSReflect, are explained in more detail below and constitute the main focus of this proposal.

1 Smart mutation

The first idea is to use *smart mutation* to evaluate and improve the effectiveness of property-based testing and to obtain confidence not just that no bugs are found, but that no bugs are left.[1] As a first step we are focusing our attention on testing the soundness of static and dynamic enforcement mechanisms (e.g., type systems, dynamic IFC systems, etc.). In this case, we are looking at smart mutation strategies that can only make the enforcement mechanism more permissive and that cover whole classes of interesting pointwise bugs. This enables the following iterative workflow: We start with the best guess for the mechanism and enforced property, and use property-based testing to find and fix bugs in the definitions and conjectured property. When we can no longer find any bugs this way, we use mutation to evaluate how good our testing is by generating a large number of more permissive enforcement mechanisms and separately testing each of them. If testing finds a counterexample to a mutant that mutant is killed, and if testing kills all generated mutants then we are done: we get high confidence that no bugs are left and our mechanism is sound with respect to the tested property. On the other hand, if some mutants escape testing, we inspect them one by one and try to manually construct a counterexample against any of them. In case we can manually find such a counterexample that was missed by testing, then our testing is clearly not good enough, and we need to improve it until the new counterexample is found automatically, e.g., by fixing bugs in the generator, or changing the random probability distribution, or coming up with a smarter generator. In practice we observed that improving testing based on a counterexample often kills a large number of other live mutants, and also sometimes finds new counterexamples against the current non-mutated design, which we fix before continuing. At the end, either all mutants are killed, or we are left with some live mutants to which we cannot find counterexamples by hand. These remaining live mutants are probably correct, and since we are only generating more permissive mutants we can use them to produce a better enforcement mechanism by applying these positive mutations to our current design, and iterating the whole process.

We believe that this workflow would be extremely effective in practice. However, in order to support this workflow, we need to be able to automatically generate smart mutants. While syntactic code mutation has been used to empirically evaluate the effectiveness of testing for decades [24], the existing techniques do not generate only more permissive mutants, which in our case means one would have to manually weed thorough lots of mutations which are simply stupid, and for which not finding a counterexample doesn't produce any benefit (probably just frustration). In recent experiments with testing noninterference (a followup of [22]) we found that splitting the IFC enforcement mechanism out of the operational semantics and into a separate rule table[2] allows us to easily generate all "missing check" and "missing taint" mutants.[3] This is guaranteed to lead to a more permissive mechanism, allows us to cover all interesting mistakes one can make in this setting, and worked very well in our experiments.

The big open problem is generalizing this idea beyond IFC to arbitrary properties and mechanisms. One simple thing to try first would be writing the enforcement mechanism as an inductive relation in Coq and mutating this relation by dropping or weakening premises. While such relations are not directly executable, and thus not directly testable, a Coq plugin can often extract an executable variant [16,30],

[1] While Edsger W. Dijkstra famously stated that "Testing shows the presence, not the absence of bugs" [18], we believe that additionally testing the testing infrastructure by systematically introducing large classes of bugs and making sure they are all found can produce high confidence in the results.

[2] This kind of split is useful beyond testing, as shown by our recent symbolic rule machine [3].

[3] For "missing taint" we exploit the fact that IFC labels form a lattice [27].

and could be extended to additionally apply our mutations beforehand. One could try this out on the System $F_{<:}$ [2, 11] type system, or some other mechanized metatheory benchmarks [25].

2 Deep integration with Coq/SSReflect

As discussed above we want QuickChick to be deeply integrated into the Coq proving process. However, an important limitation of our current prototype is that it only works for executable specifications written as Coq functions, which doesn't match usual Coq practice of defining things inductively. We could lift this limitation using the Coq plugin by Delahaye, Dubois, et al. [16, 30] for producing executable variants of inductively defined relations. More ambitiously, we would like to take advantage of QuickChick at any point during the proof process, by freely switching between declarative and efficiently executable definitions. We believe we can achieve this by integrating testing into the normal workflow of small-scale reflection proofs, as supported by the SSReflect extension for Coq [21]. However, while traditional SSReflect proofs use execution 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 executable and often not efficiently executable either. We believe we can support efficient testing during SSReflect proofs by exploiting a recent refinement framework by Dénès et al. [15, 17], which allows maintaining a correspondence and switching between proof-oriented and computation-oriented views of objects and properties.

Finally, testing can also help during automated proof search, as illustrated by recent work on ACL2s [28]. Moreover, automatically finding the witness of an existential quantifier is the same problem as generating data that satisfies the preconditions of the tested property, for which solutions have already been proposed [10, 12, 20]. This kind of usage requires to integrate testing more seamlessly into Coq though, so that the result of testing and the produced counterexamples are available to Coq tactics and terms. While our current prototype implements most things in Coq, it uses extraction to generate Haskell programs that are then executed efficiently outside of Coq,[4] and we could envision exploiting recent progress on efficient Coq reduction [8] to complement extraction. The random testing mode of the resulting internal testing framework would need to use a deterministic random number generator implemented in Coq, with the initial seed introduced in the environment through a plugin. Additional care needs to be taken when using random testing inside tactics and proofs, which need to be stable across multiple runs and different machines. Potential solutions include capturing the random seed or the produced counterexamples and saving them inside the proof script, or replacing random testing with exhaustive testing with small instances [9, 23, 29] and narrowing-based testing [9, 13, 26, 29] when possible.

References

[1] *The Coq Proof Assistant*, 2012. Version 8.4.

[2] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. *TPHOLs*. 2005.

[3] 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. *POPL*, 2014. To appear.

[4] S. Berghofer, L. Bulwahn, and F. Haftmann. Turning inductive into equational specifications. *TPHOLs*. 2009.

[5] S. Berghofer and T. Nipkow. Executing higher order logic. *TYPES*. 2002.

[4] This also exploits unsafe features of Haskell like tracing and printing in the middle of the computation, which are useful for debugging, but which would be hard to replicate in Coq.

[6] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. *SEFM*. 2004.

[7] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. *ITP*. 2010.

[8] M. Boespflug, M. Dénès, and B. Grégoire. Full reduction at full throttle. *CPP*. 2011.

[9] L. Bulwahn. The new Quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. *CPP*. 2012.

[10] L. Bulwahn. Smart testing of functional programs in Isabelle. *LPAR*. 2012.

[11] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of System F with subtyping. *Information and Computation*, 109(1/2):4–56, 1994.

[12] M. Carlier, C. Dubois, and A. Gotlieb. FocalTest: A constraint programming approach for property-based testing. In *Software and Data Technologies*, volume 170 of *Communications in Computer and Information Science*, pages 140–155. Springer, 2013.

[13] J. Christiansen and S. Fischer. EasyCheck - test data for free. *FLOPS*. 2008.

[14] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ICFP*. 2000.

[15] C. Cohen, M. Dénès, and A. Mörtberg. Refinements for free! *CPP*, 2013. To appear.

[16] D. Delahaye, C. Dubois, and J.-F. Étienne. Extracting purely functional contents from logical inductive types. *TPHOLs*. 2007.

[17] M. Dénès, A. Mörtberg, and V. Siles. A refinement-based approach to computational algebra in Coq. *ITP*. 2012.

[18] E. W. Dijkstra. Software engineering techniques on page 16. Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 October 1969.

[19] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. *TPHOLs*. 2003.

[20] S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. *PPDP*. 2007.

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

[22] C. Hriţcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos. Testing noninterference, quickly. *ICFP*, 2013.

[23] D. Jackson. *Software Abstractions: Logic, Language, and Anlysis*. The MIT Press, 2011.

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

[25] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: On the effectiveness of lightweight mechanization. *POPL*, 2012.

[26] F. Lindblad. Property directed generation of first-order test data. *TFP*, 2007.

[27] B. Montagu, B. C. Pierce, and R. Pollack. A theory of information-flow labels. *CSF*. 2013.

[28] H. Raju Chamarthi, P. Dillinger, M. Kaufmann, and P. Manolios. Integrating testing and interactive theorem proving. *ACL2*, 2011.

[29] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and Lazy SmallCheck: automatic exhaustive testing for small values. *Haskell*. 2008.

[30] P.-N. Tollitte, D. Delahaye, and C. Dubois. Producing certified functional code from inductive specifications. *CPP*. 2012.

[31] T. Weber. Bounded model generation for Isabelle/HOL. *ENTCS*, 125(3):103–116, 2005.