QuickChick: Property-Based Testing for Coq

Maxime Dénès[1], Cătălin Hriţcu[2], Leonidas Lampropoulos[1], Zoe Paraskevopoulou[2,3], Benjamin C. Pierce[1]

[1]University of Pennsylvania [2]Inria Paris-Rocquencourt [3]NTU Athens

April 4, 2014

Co-designing software or hardware systems and their formal proofs is an appealing idea, with the expectation that the rigor enforced by formal methods will percolate the whole design. In practice 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. We believe that property-based testing (PBT) can dramatically decrease the number of failed proof attempts and reduce the overall cost of producing formally verified systems. Despite the existence of experimental tools [Wil11], Coq is still lagging behind proof assistants like Isabelle, which provides several mature PBT tools (e.g. [Bul12]). We aim to improve the PBT support in Coq, while also investigating several innovations we could add into the mix like polarized mutation testing and a language-based approach to custom generation. We are also exploring whether PBT could bring more confidence to the implementation of Coq itself.

1 A random testing framework for Coq

As a first step, we implemented a prototype[1] PBT framework for Coq, very similar to QuickCheck [CH00]. We then took a previous development that was using QuickCheck to test non-interference for increasingly sophisticated dynamic information flow control mechanisms for low-level code [Hri+13] and ported it to Coq. In particular, the abstract machines are now formalized in Coq, making it possible to test and then prove properties on the same artifacts. Our prototype has most of the features of QuickCheck. It relies on Coq's extractor to Haskell, but we are still exploring the design space. In particular, we may port it to OCaml, or even try to do most of the computations inside Coq.

One issue when integrating PBT into the normal Coq proving process is that our current prototype only works for executable specifications, which doesn't match the regular practice of using inductive definitions. We could try to lift this limitation using an existing plugin [DDÉ07; TDD12] for producing executable variants of inductively defined functional relations. More ambitiously, we would like to use PBT at any point during a proof, by freely switching between declarative and efficiently executable definitions. We believe we can achieve this by integrating PBT into small-scale reflection proofs, as supported by SSReflect. 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 support efficient testing in SSReflect by exploiting a recent refinement framework [DMS12; CDM13], which allows maintaining a correspondence and switching between proof-oriented and computation-oriented views of objects and properties.

2 Testing Coq itself

We are also leading a separate research effort, motivated by the fact that the trust brought by a Coq proof is always modulo potential implementation bugs in its kernel. While this may seem not to be a practical

[1]Our prototype is available online: `https://github.com/lemonidas/QuickChick`

matter of concern, the verification of critical software meant to be used in industry increases the need for guarantees on the tools used to carry out this verification. For example, certification authorities in avionics have precise requirements on such tools. Although only a limited number of critical bugs have been found in Coq's kernel over the years, this number is not zero.

We are investigating the use of PBT to detect some of these bugs. For example, the first author recently found (manually) a critical bug in a function implementing restrictions on pattern matching. Although a full executable specification of this function would probably be nothing less than its implementation, the bug could have been caught by checking if the result of the function is preserved by the reduction rules of the calculus, on randomly generated terms. We also want to check syntactic properties of the calculus actually implemented by Coq, like Church-Rosser or subject reduction, on which consistency and normalization proofs often rely. Some counterexamples to these properties are known for given fragments of the calculus (e.g. subject reduction with co-inductive types), the question being if unknown examples could be found as well.

The main difficulty is that testing such properties most often relies on generating well-typed CIC terms. Which might sound overambitious, because the Curry-Howard isomorphism tells us that it amounts to proof search. However, unlike automated theorem proving, we can generate a term in any type that we like, i.e. we can also choose the theorem. Also, our goal is not to implement a press-button procedure that would tell if an inconsistency has been found somewhere (it would probably not be realistic), but rather a framework that could aid developers when studying parts of the kernel that may be problematic.

We are reusing some ideas from the existing work [Pał+11], which has been successful in the context of GHC (the Glasgow Haskell Compiler). In particular, combining constructs of the language (abstractions, applications,...) and constants already present in the environment. Of course dependent types (and full polymorphism) bring new challenges, like more complex dependencies between subgoals created during the generation. We believe that implementation techniques for higher-order logic programming languages could give interesting answers to some of these challenges.

References

[Bul12] L. Bulwahn. "The New Quickcheck for Isabelle - Random, Exhaustive and Symbolic Testing under One Roof". *CPP*. 2012.

[CDM13] C. Cohen, M. Dénès, and A. Mörtberg. "Refinements for free!" *CPP*. 2013.

[CH00] K. Claessen and J. Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". *ICFP*. 2000.

[DDÉ07] D. Delahaye, C. Dubois, and J.-F. Étienne. "Extracting Purely Functional Contents from Logical Inductive Types". *TPHOLs*. 2007.

[DMS12] M. Dénès, A. Mörtberg, and V. Siles. "A Refinement-Based Approach to Computational Algebra in Coq". *ITP*. 2012.

[Hri+13] C. Hritcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. A. de Amorim, and L. Lampropoulos. "Testing noninterference, quickly". *ICFP*. 2013.

[Pał+11] M. H. Pałka, K. Claessen, A. Russo, and J. Hughes. "Testing an optimising compiler by generating random lambda terms". *AST*. ACM. 2011.

[TDD12] P.-N. Tollitte, D. Delahaye, and C. Dubois. "Producing Certified Functional Code from Inductive Specifications". *CPP*. 2012.

[Wil11] S. Wilson. "Supporting dependently typed functional programming with proof automation and testing". PhD thesis. University of Edinburgh, 2011.