Optimal Parameter Selection for Property-Based Testing

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

Instittition: 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),
- optimization and approximation algorithms,
- probabilities and statistics.

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 called QuickChick.[2] There are, however, important remaining challenges and research opportunities that are

[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.

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

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.

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.

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 are currently 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. We are also exploring 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, and of Fetscher et al. (2015) on generating well-typed terms from the definition of a type-system. We will improve on that work by allowing the user to customize the probability distribution, by providing better efficiency, and by scaling up to generating data satisfying more intricate properties.

Internship Topic: Optimal Parameter Selection for Property-Based Testing

Optimal parameter selection is a pervasive concern in property-based testing, and the goal of this internship is to provide an efficient algorithmic solution to this problem. For instance, when defining a generator (using QuickCheck combinators or the generator language mentioned above) one can choose weights for local probabilistic choices that eventually determine the final probability distribution of the generated data. The relation between the final probability distribution and these local choices is, however, complicated, because of filtering (generating data and discarding the one that doesn't satisfy certain conditions), recursion (where usually the probability of doing a recursive call decreases as the recursion level increases), etc. Users currently resort to running experiments, gathering information about the obtained probability distribution, and tweaking the local weights until they obtain a reasonable probability distribution; a tedious and time consuming manual process. In this internship we will develop an efficient optimization algorithm that given a desired probability distribution on the outputs, automatically selects the local weights that come closest to it.

Some experiments with a naive simulated annealing prototype show that this is possible in simple cases, such as generating machine instructions (Hriţcu et al., 2013), where we try to make the distribution of instruction opcodes nearly uniform. This naive prototype needs, however, a very large number of samples (hundreds of thousands) to precisely learn the obtained probability distribution at each step, which makes the whole process take hours. Using a less naive distribution learning algorithm (Kearns et al., 1994) and an optimization method that converges faster and is less sensitive to distribution learning errors should already improve things significantly. Moreover, we can try to handle distributions with very large domains by taking advantage of recent sub-linear collision-based algorithms for testing properties of distributions or approximating the distance between two distributions (Rubinfeld, 2012). While these techniques treat the generator as a black-box and only use sampling to dynamically learn properties of its final probability distribution, it would be interesting to also consider white-box techniques. We could for instance decompose the problem of learning properties of the whole generator into learning properties of small parts that are independent of the choice of weights and thus only have to be learned once. In some simple cases, such as uniformly generating machine instructions, we can compute the filtering probabilities and use the desired distribution to analytically compute the best weights. When this is not possible we could statically analyze the code of the generator and obtain symbolic approximations of its probabilistic behavior.

More ambitiously, we would like a framework in which we can specify multiple (Diakonikolas, 2011) optimization constraints on the generator, not just on the final probability distribution. For instance, we might want to generate machine code programs that execute for as long as possible (Hriţcu et al., 2013) or that exercises as many semantic rules as possible. Other quantities we would like to simultaneously optimize are generation and test execution time. The framework will try to find good values not just for the local probabilistic choices, but for all parameters the user specifies as well as parameters that are internal to the generation framework (e.g., the generator language described above), but which are the moment instantiated globally based on informal heuristics. Finally, a meta-optimization procedure could dynamically optimize the parameters of the parameter selection framework itself, leading to an adaptive mechanism that interleaves meta-optimization and parameter optimization with running tests and gathering statistics for the next optimization round.

We will use realistic case studies to assess the success of our framework: the hardware-assisted security monitors of Azevedo de Amorim et al. (2014a,b) and type-checkers for simple and dependent types (Barras and Werner, 1997; 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.

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.

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.

I. Diakonikolas. *Approximation of Multiobjective Optimization Problems*. PhD thesis, May 2011.

B. Fetscher, K. Claessen, M. H. Pałka, J. Hughes, and R. B. Findler. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. To appear in ESOP, 2015.

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.

M. J. Kearns, Y. Mansour, D. Ron, R. Rubinfeld, R. E. Schapire, and L. Sellie. On the learnability of discrete distributions. In F. T. Leighton and M. T. Goodrich, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, pages 273–282. ACM, 1994.

R. Rubinfeld. Taming big probability distributions. *ACM Crossroads*, 19(1):24–28, 2012.

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.