Luck: a Language for Effective, Dependable Property-Based Testing

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

Institition: INRIA Paris, Prosecco Team

Location: 2 rue Simone Iff, 75012 Paris, France

Language: English

Existing skills or strong desire to learn:

- functional programming (e.g. ML or Haskell);
- property-based testing (e.g. QuickCheck);
- optional: formal verification in Coq;
- optional: logic programming, constraint solving, probabilities, semantics, program synthesis

Introduction

Context Property-based testing (PBT) and formal verification are two complementary techniques for reducing the prevalence of software errors. PBT is a systematic technique that promises to both reduce the cost and increase the thoroughness of testing. It allows software testers to write executable specifications capturing expected properties of the system under test, or a small part of it, and it semi-automatically tests these properties on large sets of test data, e.g., randomly generated according to some probability distribution. When a counterexample is found it is automatically shrunk to a minimal one, which is displayed to the user. QuickCheck [5], the first popular PBT tool, targets Haskell and is widely used in the functional programming languages community. QuickCheck has been ported to all mainstream programming languages and has spurred significant industrial interest.

While PBT is effective at finding errors, no amount of testing is enough to guarantee the absence of errors. Only formal verification can provide the strong guarantees required for safety- and security-critical software, for which errors can have disastrous real-world consequences. Continuous progress in formal verification in the past decades has culminated with many recent milestones (e.g., seL4, CompCert, RockSalt, CertiKOS, miTLS) which show beyond any doubt that (with enough effort) formal verification can scale up to sizable software artifacts, such as compilers, operating system kernels, and cryptographic protocols.

Despite this great progress, formal verification in a proof assistant is still prohibitively expensive for mainstream adoption. Carrying out a formal proof 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 goal of this project is *to achieve the lower cost of PBT and the stronger formal guarantees of verification in one single system*. For this we will tightly integrate PBT and formal verification in the Coq proof assistant and achieve a virtuous cycle in which testing helps verification and verification helps testing. Testing will decrease the cost of verification 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. Easily understandable counterexamples will guide users not just for producing correct system designs, but also for constructing the formal evidence of their correctness during the verification process, by quickly validating proof goals, potential lemmas, and inductive invariants. All this will dramatically decrease the number of costly failed proof attempts in Coq developments, which 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. Finally, our solution will be fully general, and thus also be useful to regular software developers for testing outside proof assistants.

Luck Since in practice dependable PBT can itself be prohibitively difficult, the main objective of this project is to reduce the cost of PBT, while maintaining its full generality and programmability. We achieve this by a combination of programming language design and code synthesis. In particular we introduce Luck, a novel domain-specific language for property-based generators that leads to an order of magnitude reduction in code size using ideas from functional logic programming and constraint solving, provides automation of common patterns, yet is fully customizable, and keeps the user fully in control. Syntactically, we use a simple probabilistic lambda calculus with algebraic datatypes extended with "unknown values" (logical variables). When all values a Luck program uses are known, the program evaluates in the standard way, producing a single result. When some values used by the program are unknown, Luck randomly instantiates these unknowns to the extent needed [2]. Wrong instantiations can cause costly backtracking, so we allow the user to delimit expressions for which instantiation should be delayed and for which our language gathers and propagates constraints before committing to an instantiation. For instance, if `x` is

an unknown in the expression `low < x && x < high` we only want to instantiate `x` after processing both constraints. To account for instantiation and backtracking the semantics of Luck expressions is defined in terms of co-inductive computation trees, where branching is probabilistic. Work is already quite advanced on defining this semantics, on working out the metatheory, and on building a prototype Luck interpreter [14, 10]. Still, more research is needed both on the theory and practice of Luck.

Problem 1: A denotational semantics for Luck While we are currently giving an operational semantics to Luck, we would like to explore devising a denotational semantics for it. For this we could for instance take inspiration in recent work on semantics for probabilistic programming languages [1]. Our goal will be to prove that the new denotational semantics agrees with the existing operational one and to investigate whether soundness and completeness with respect to the boolean semantics of predicates is easier to establish for the denotational semantics. Finally, we aim to formalize these proofs in the Coq proof assistant.

Problem 2: Synthesizing Luck programs While Luck eliminates the duplication between generators and checkers for the same property, if one wants to obtain formal guarantees one currently needs to manually relate Luck code to the high-level property that it tests. In recent work, we have proposed a general verification methodology for proving that executable testing code is testing the right Coq property [12]. In order to reduce user effort, we will investigate classes of properties for which we can use program synthesis to produce correct-by-construction programs from their declarative specifications. For this we will take inspiration from logic programming [4, 3, 6, 13] and deductive synthesis [7]. We will use existing examples from the literature to assess the feasibility of our solution [4, 3, 6, 13, 9, 11, 8].

More problems In addition to the two problems above, other challenges on the Luck project include integrating Luck with Coq and/or Haskell, building an efficient compiler for Luck, devising an optimal parameter selection framework for Luck generators, and performing more ambitious case studies.

References

[1] Workshop on probabilistic programming semantics (PPS), 2016.

[2] S. Antoy. A needed narrowing strategy. *JACM*. 2000.

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

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

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

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

[7] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. *POPL*. 2015.

[8] B. Fetscher, K. Claessen, M. H. Palka, J. Hughes, and R. B. Findler. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. *ESOP*. 2015.

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

[10] L. Lampropoulos, B. C. Pierce, C. Hriţcu, J. Hughes, Z. Paraskevopoulou, and L. Xia. Making our own Luck: A language for random generators. Draft, 2015.

[11] 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*. 2011.

[12] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. *ITP*. 2015.

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

[14] L. Xia and C. Hriţcu. Integrating functional logic programming with constraint solving for random generation of structured data. Inria Internship Report, 2015.