RECRUITMENT CAMPAIGN 2014

PROPOSAL TO RECRUIT A PhD STUDENT ON GRANT

Research Team-Project: Prosecco, Inria Paris-Rocquencourt

Topic Title: Speeding up Theorem Proving with Property-Based Testing

Keywords:
programming languages, design, verification, mechanized metatheory, interactive theorem proving, Coq proof assistant, random testing, QuickCheck, generating highly-structured data, mutation testing, probabilistic programming, domain-specific languages, reducing proof effort

Research context (*4 000 characters max.*):

Designing complex systems that provide strong safety and security guarantees is challenging (e.g. programming languages, compilers, language runtimes, reference monitors, operating systems, hardware, etc). Proof assistants such as Coq 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. We believe that we can use *property-based testing (PBT)* to dramatically decrease the number of failed proof attempts and reduce the overall cost of producing formally verified systems. PBT can systematically explore the input space and find bugs in definitions and conjectured properties early in the design process, postponing proof attempts until one is reasonably confident that the design is correct. Moreover, PBT can be very helpful during the proof process for quickly validating potential lemmas, inductive invariants, or simply the current proof goal.

We have recently started an ambitious research program aimed at improving the state of the art in PBT and making it an invaluable part of the interactive theorem proving process, in collaboration with researchers from University of Pennsylvania (Benjamin C. Pierce, Maxime Dénès, Leonidas Lambropoulos, Antal Spector-Zabusky, etc.), Chalmers University (John Hughes), and Microsoft Research Cambridge (Dimitrios Vytiniotis). As a first step in this direction we built the initial prototype of a PBT framework for Coq, very much similar to QuickCheck (Claessen and Hughes, ICFP'00). We used this framework to test noninterference for increasingly sophisticated dynamic information flow control (IFC) mechanisms for low-level code (Hritcu et al., ICFP'13). This case study showed that random testing can quickly find a variety of 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. More recently, we used "polarized" mutation testing to systematically introduce all missing-taint and missing-check bugs and incrementally improve our testing strategy until all the bugs were found, which gave us high confidence in our testing. Afterwards we could much more easily prove in Coq that the proposed IFC mechanism is indeed correct (Azevedo de Amorim et al., POPL'14).

We plan to extend this prototype Coq testing framework, building on the existing PBT research literature (e.g., random testing, exhaustive testing with small instances, narrowing based testing,

constraint programming based testing, etc.), as well as related work on generating executable code from inductive definitions, automatically generating data that satisfies the preconditions of the tested property, integrating testing and proof automation, exploiting model finders, etc. However, having tried to use many of the existing tools ourselves, we have the feeling that making this work well in practice requires further extending the state of the art. We plan to do this by working out several promising new ideas, three of which constitute the focus of this PhD topic and are explained in more detail below: (1) developing a general methodology and framework for polarized mutation testing, (2) designing a language for custom test-data generators, and (3) integrating all this into the normal Coq/SSReflect proving process. We additionally plan to explore the practical applicability of these ideas to typical problems in formalized metatheory, in particular to soundness proofs for expressive type systems, program logics, and dynamic IFC mechanisms.

PhD research work description (*4 000 characters max.*):

1. A General Framework for Polarized Mutation Testing

The first idea is to test and debug the testing infrastructure itself (e.g. the test-data generator) by systematically mutating the artifact under test to introduce all pointwise bugs from an interesting class and making sure that they are 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. 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 property and 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 non-interference by weakening it (e.g. dropping IFC side-conditions). Similarly, we can break type preservation either by strengthening the occurrence of the typing relation in the conclusion, or by weakening the occurrence in the premise. Beyond working out these case studies in detail, the goal is to develop a general methodology and framework for polarized mutation testing.

2. Sampleable Predicates: A Language for Writing Custom Test-Data Generators

Most properties one encounters in practice are conditional (e.g. P implies Q). When the premises are sparse, effective testing requires writing custom random generators tuned to the specific property under test. This is, however, difficult and error prone. The second direction of this thesis is to design a new domain-specific language for writing custom test-data generators. An expression in this language denotes both a logical predicate and a random generator that only produces values satisfying the predicate. 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. For expressions where every variable is either fixed by previous choices or else unconstrained we can ensure an efficient evaluation strategy that does not involve backtracking. The goals here are to formalize a core language and prove a number of meta-theoretic properties, and then to extend this to a full-fledged language embedded into Coq. We would also like to explore various approaches for controlling the obtained probability distributions.

3. Deep Integration with Coq/SSReflect

Finally, we plan to integrate PBT and the ideas above into the normal Coq proving process. 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 the plugin by Delahaye et al. (TPHOLs'07) and Dubois et al. (CPP'12) for producing executable variants of inductively defined 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 by Dénès et al. (ITP'12) and Cohen et al. (CPP'13), which allows maintaining a correspondence and switching between proof-oriented and computation-oriented views of objects and properties.

Required skills and background:

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

Contact :

- **Catalin Hritcu (catalin.hritcu@inria.fr) and**
- **Karthikeyan Bhargavan (karthikeyan.bhargavan@inria.fr)**