Deriving Test Input Generators Through Program Inversion

Sophie Bosio
Department of Informatics
University of Oslo
Oslo, Norway
sabosio@uio.no

***Abstract*—This extended abstract will detail the tentative plans for my master's thesis. My thesis topic will be using program inversion to derive test data input generators automatically for use in property-based testing.**

Property-based testing is a methodology based on properties - or assertions that should hold about the program in general. The programmer writes properties for their program, and uses existing tools that automatically generate and run tests based on these properties. However, one reason why it is not more widely adopted in industry scenarios, is because the tools can only generate test input data automatically for basic input types. Testing new types requires the programmer to write custom generators for those types, but writing good generators is difficult. The thesis work consists of using program inversion to automatically derive good test input generators for use with property-based testing tools. The overarching goal of the project is to make academic testing tools more easily adoptable for industry use.

I. Introduction

Property-based testing is a testing methodology that allows for comprehensive testing. More common methodologies such as unit and integration testing make it easy to begin writing tests, as they only require the programmer to provide a handful of usage examples. However, this often results in a false sense of correctness and misleads the programmer into thinking the program functions as intended. As it turns out, it is very difficult to predict and formulate the test scenarios in which errors *could* occur [2].

One of the most well-known tools for property-based testing is the library QuickCheck, originally developed for Haskell by Koen Claessen and John Hughes in 1999 [1]. It is a combinator library that generates test cases based on assertions - or properties - that should hold for a function or combination of functions. It heuristically attempts to generate a test case that falsifies the properties. One of the tool's useful features is its ability to reduce the failing test case to a minimal failing instance. This way, the feedback to the programmer is as concise as possible, facilitating debugging.

II. Motivation

Instead of examples of input and expected output, property-based testing requires the programmer to specify *properties* of the program. These properties are assertions that should hold in general. To test these assertions, we need a way to randomly generate test data and try them against the assertions. If failing tests are found, the programmer reviews the properties to find potential errors in their specification. If the properties are indeed correctly specified, the test results reveal software bugs, with the added knowledge of what scenarios provoke them.

Currently, automatic testing tools such as QuickCheck can generate and run tests, given a generator for the required input [1]. However, this presents its own issues. Firstly, writing a good generator is hard. Secondly, writing properties that test not only the input most commonly generated, but instead test the input most likely to *fail*, requires significant forethought on the part of the programmer. The result of these issues can be that testing is foregone entirely, because writing them becomes too tedious or involved. If instead, the generator could be derived directly from the program, we could circumvent this extra step, reducing the risk of programmer error and encouraging testing.

One promising avenue to automatically deriving generators is program inversion. Given a programming language with a construct specifically for tests - or program invariants - and a way to invert both functions and tests written in the language, it might be possible to derive an inverse of the test. Since the result of running a test on test input is a boolean value ('True' corresponding to a passed test and 'False' to a failed test), applying the test's inverse to the boolean value 'False' should result in test input that falsifies that test. However, this clearly cannot be a true function, since the inverse would return many different test inputs, as many inputs could falsify a single test. That is, the inverse of the test function is itself not a function, but a relation. This complicates matters.

While a perfect inverse may be difficult, or more often impossible, to derive, it may not be necessary. In fact, it may be sufficient to derive a function that only returns a *subset* of the failing test inputs, and that probabilistically selects input with a high likelihood of failing. In the case where input data can be generated, we can guarantee that there does exist failing examples. In the opposite case, we cannot guarantee program correctness, but it may increase our *confidence* that the program is feature compliant and functional in the relevant cases.

Out of all the input data generated, we want a way to find the most useful ones. In particular, some of these inputs may be excessively long or complex, which reduces the utility of the result. Therefore, we would like to shrink failing input to

978-1-7281-4387-3/19/$31.00 © 2019 IEEE

minimal failing input, removing redundancy.

Once this has been achieved, we should be able to integrate our generator with the property-based testing tool QuickCheck. The approach described above would allow the generator for new types of input to be derived automatically from the code, thus removing this extra step for the programmer.

The project, therefore, is an attempt at bridging the gap between sophisticated academic testing tools and a fast-paced industry environment, by making these tools easier to integrate into the workflow of real-world professional developers.

III. Methodology

The thesis work will largely consist of designing the programming language and implementing an interpreter capable of the program inversion described above. The goal of the project will be an operational programming language with an interpreter that interfaces with QuickCheck to run tests automatically. The language will be functional and implemented in Haskell.

The design of the language and the implementation of the interpreter will necessitate a preliminary understanding of existing tools, libraries, and technologies for property-based testing and program inversion and reversion. In particular of QuickCheck and its implementation. The most interesting aspects of QuickCheck in relation to this thesis are its heuristics for generating test input likely to falsify the tests and its mechanism for shrinking test cases to minimal failing ones.

With regard to the language design, relevant material exists on the subject of both program inversion and reversible programming languages. This thesis will draw inspiration particularly from the grammar-based approach to program inversion described by Matsuda et al. in their paper on the matter [4]. Programming languages that will serve as inspiration include the invertible functional language Jeopardy [3] and the reversible imperative language Janus [5].

The final thesis will detail the motivation for the project, provide necessary information about property-based testing and QuickCheck, the design choices of the language, the code for the generator derivation tool, as well as its documentation.

References

[1] Claessen, K., and Hughes, J. Quickcheck: A lightweight tool for random testing of haskell programs. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 46* (01 2000).

[2] Hughes, J. *Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane*, vol. 9600. 03 2016, pp. 169–186.

[3] Kristensen, J. T., Kaarsgaard, R., and Thomsen, M. K. Jeopardy: An invertible functional programming language, 2022.

[4] Matsuda, K., Mu, S.-C., Hu, Z., and Takeichi, M. A grammar-based approach to invertible programs. In *Programming Languages and Systems* (Berlin, Heidelberg, 2010), A. D. Gordon, Ed., Springer Berlin Heidelberg, pp. 448–467.

[5] Yokoyama, T. Reversible computation and reversible programming languages. *Electronic Notes in Theoretical Computer Science 253*, 6 (2010), 71–81. Proceedings of the Workshop on Reversible Computation (RC 2009).