Type-Based Test Generation for Haskell and Scala using Constraint Logic Programming

Rafael Fernández Ortiz

February 8, 2023

Abstract

Your abstract.Your abstract.Your abstract.Your abstract.Your abstract.Your abstract.Your abstract.Your abstract.Your abstract.Your abstract.Your abstract.Your abstract.Your abstract.Your abstract.Your abstract.Your abstract.Your abstract.Your abstract.

Contents

Chapter 1

Introduction

Software is a set of instructions that tell a computer what to do. It can be used for a wide range of tasks, from simple calculations to complex simulations. Confidence in software is important because it ensures that the software will perform as expected and not cause unintended consequences. Unfortunately, that confidence is not present most time.

Risks in software can come from a variety of sources, such as bugs, security vulnerabilities, or poor design. These risks can lead to errors, crashes, or even loss of data.

For critical systems, such as those used in the medical or aerospace industries, the stakes are even higher. These systems must be thoroughly tested and validated to ensure that they will not cause harm or failure in a critical situation. For example:

- Medical systems: Medical systems such as electronic health records (EHRs) and medical devices are critical systems that can have serious consequences if they fail. A bug in an EHR system could lead to incorrect patient information being displayed, potentially leading to a misdiagnosis or other medical errors.
- Aerospace systems: Aerospace systems such as aircraft navigation systems and flight control systems are critical systems that must operate reliably at all times. A bug in an aircraft navigation system could cause the plane to fly off course, leading to a crash.
- Industrial control systems: Industrial control systems (ICS) are used to control and monitor industrial processes such as manufacturing, power generation, and oil and gas production. An issue in an ICS could cause a malfunction in a manufacturing process, leading to costly downtime or even physical damage to the equipment, or even a cyber-attack on an ICS could cause a shutdown of the whole process causing a major disruption.

When we talk about software quality, we are talking about of how well a software system or application meets its specified requirements and is fit for its intended use. It is a multi-faceted concept that includes aspects such as whether the software performs the intended tasks, and whether it meets the needs of the end users. That is functionality. Also, it includes other factors such as reliability, usability, performance, and maintainability.

Ensuring that software has good functionality, or more generally, ensuring software quality is important because it helps to ensure that the software will be useful, effective and that it will perform as expected and not cause unintended consequences. for its intended purpose.

In order to guarantee the expected behavior and therefore, to have good software quality, testing and validation are critical for identifying and mitigating these issues.

1.1 Testing

Software testing (or simply testing) is the process of evaluating a system or its component(s) to find whether it satisfies the specified requirements. It consists of executing a program on a known pre-

selected set (test suite) of inputs (test cases) and inspecting whether the outputs match the expected results.

This process validates the semantic properties of a program's behavior. It's important to test software thoroughly before deployment to ensure that it functions as intended and to identify and fix any bugs or other issues. Testing is an essential step in the software development process and is critical for ensuring the quality and reliability of software.

Therefore, we can define in a relaxed way that a **test** is a set of executions on a given program using different input data for each execution; its purpose is to determine if the program functions correctly. A test has a negative result if an error is detected during the test i.e., the program crashes or **a property is violated**.

A test has a positive result if a series of tests produces no error, and the series of tests is "complete" under some coverage metric. When we say in software testing that a test is "complete", it refers to the level of coverage that the tests provide for the software being tested. In other words, that reflects a representative percentage of the reliability of the software with respect to expected behavior. However, we have to consider that "reliability" is relative and it is biased and subject to the chosen test cases, which is itself subject to the criteria of the tester.

A test has an "incomplete" result if a series of tests produces no errors but the series is not complete under the coverage metric. In summary, a test is focused on evaluating the software to find any issues and bugs.

There are different types of tests that can be used during the software development process, each with a different purpose and focus. Some of the main types of tests are:

- **Unit testing**: Unit testing is a type of testing that focuses on individual components of the software, such as individual functions or methods. The goal of unit testing is to ensure that each component behaves as expected. Unit tests are usually automated, and they are run as part of the development process to catch any issues early.
- **Integration testing**: Integration testing is used to ensure that different components of the software work together correctly. It tests the interactions between different parts of the software. Integration tests are usually automated, and they are run after the unit tests to ensure that the integrated system behaves as expected.
- **Acceptance testing**: Acceptance testing is used to ensure that the software meets the needs of the end users. It is typically done by the customer or other stakeholders to ensure that the software meets their needs. Acceptance tests can be automated or manual, and they are run after system tests to ensure that the system is ready to be deployed.

Sadly, trying to find counter-examples by testing that produces bug a behavior non-expected is most of the time a difficult task. In simpler software, testers could find most of those cases which produce counter-examples, designing test cases one by one. However, the design process reaches those cases that one knows by experience or intuition, leaving aside very interesting and not at all intuitive cases that can hardly be imagined. For this reason and because it can be a very tedious task, it would be ideal to automate the generation of test cases.

1.1.1 Property-Based Testing

Property-based testing is a technique that uses random inputs to test the properties of a system, rather than specific inputs. It helps to mitigate risks in the software industry by providing a way to test the software in a wide range of scenarios. This can help to identify bugs and other issues that may not be found using traditional testing techniques such as manual testing or unit testing.

The use of random inputs in property-based testing allows for a more thorough exploration of the software's behavior, making it more likely that any bugs or issues will be found. It also helps to ensure

that the software behaves correctly in a wide range of scenarios, which is especially important for critical systems.

Property-based testing helps to ensure that the software is robust and can handle unexpected inputs or edge cases. This is particularly important for systems where failure could have serious consequences. Also helps in testing the software performance and scalability, by testing the software with large inputs, it can identify potential performance issues that would be difficult to detect with other testing techniques.

In critical systems, for example, property-based testing can be used to test the software's behavior to ensure that it will not cause harm or failure in a critical situation and also can help to increase confidence in the software.

In a more specific way, the specification of one or more properties drives the testing process, which assures that the given program meets the stated property. For example, if an analyst wants to validate that a specific program correctly authenticates a user, a property-basted testing procedure tests the implementation of the authentication mechanisms in the source code to determine if the code meets the specification of *correctly authenticating the user*.

Property-based testing is complementary to software engineering life cycle methodologies. Analysis and inspection of design, requirements, and code help to prevent flaws from being introduced into source code. Property-based testing validates that the final product is free of specific flaws. Because property-based testing concentrates on generic flaws, it is ideal for focusing on analysis late in the development cycle after program functionality has been established. Specifications state what a system should or should not do. The advantage of using specifications is the formalism they establish for verifying proper (or improper) program behavior.

In a property-based framework, test cases are automatically generated and run from assertions about the logical properties of the program. Feedback is given to the user about their evaluation.

Property-based testing naturally is based on the logic programming paradigm. Assertions are first-order formulas and thus easily encoded as program predicates. Therefore, a property-based approach to testing is intuitive for the logic programmer.

What kind of problems can I use property-based testing for?

Property-Based testing can be used from anything as simple as unit tests up to very broad system tests. For unit tests, there are stateless properties that mostly validate functions through their inputs and outputs. For system and integration tests, you can instead use stateful properties, which let you define ways to generate sequences of calls and interactions with the system, the same way a human tester doing exploratory testing would.

Stateless properties are easiest to use when you can think of rules or principles your code should always respect, and there is some amount of complexity to the implementation. It might be a bit of a humblebrag, but stateless properties are least interesting when the code you want to test is trivial.

Stateful properties work well in general. The two harder things with them is handling the initial setup and teardown of more complex stateful systems, and dealing with hard-to-predict behavior of code, such as handling timeouts or non-deterministic results. They're possible to deal with, but rather trickier. Anything else can be fair play.

Related Works

1.1.2 QuickCheck: Automatic testing of Haskell programs

QuickCheck is a library for random testing of program properties. The programmer provides a specification of the program, in the form of properties which functions should satisfy, and QuickCheck then tests that the properties hold in a large number of randomly generated cases. Specifications are expressed in Haskell, using combinators provided by QuickCheck. QuickCheck provides combinators

to define properties, observe the distribution of test data, and define test data generators.

QuickCheck is one of the original property-based testing frameworks. It was developed in 1999 as a tool for Haskell, and it has since been ported to other languages such as Erlang, Scala, and Clojure. QuickCheck uses random input generation and shrinking to automatically generate test cases for a given property. [**Repository**]

1.1.3 ScalaCheck: Automatic testing of Scala and Java programs

ScalaCheck is a library written in Scala and used for automated property-based testing of Scala or Java programs. ScalaCheck was originally inspired by the Haskell library QuickCheck, but has also ventured into its own. ScalaCheck is used by several prominent Scala projects, for example the **Scala compiler** and the **Akka** concurrency framework. [**Repository**]

1.1.4 PropEr: Property-based testing tool for Erlang

PropEr is a QuickCheck-inspired open-source property-based testing tool for Erlang, developed by Manolis Papadakis, Eirini Arvaniti, and Kostis Sagonas.

PropEr is such a property-based testing tool, designed to test programs written in the Erlang programming language. Its focus is on testing the behaviour of pure functions. On top of that, it is equipped with two library modules that can be used for testing stateful code. The input domain of functions is specified through the use of a type system, modeled closely after the type system of the language itself. Properties are written using Erlang expressions, with the help of a few predefined macros. [**Repository**]

1.1.5 Hypothesis: Property-based testing tool for Python

Hypothesis is a modern property-based testing library for Python. It's similar to the previously mentioned frameworks but with some differences, it also provides features like stateful testing and advanced strategies for input generation. [**Repository**]

1.2 State of the Art: Test Cases with Pre-Condition

1.2.1 Sorted List

1.2.2 Red-Black Trees

1.3 Proposal

Structure of this document