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

Rafael Fernández Ortiz

October 25, 2022

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

We are constantly evolving, society is changing and with it the most basic needs of human beings are no longer a priority. Nowadays, something as primitive as acquiring food or a piece of clothing is something we no longer worry about. Due to industries and logistics processes, we have normalized, for example, being able to buy a chicken fillet easily. We have crossed that survival barrier to something common and easily accessible.

New needs arrive, new ways of understanding the world and of thinking about possible things that before could not even be imagined: flying, sending messages to someone in another country in a matter of seconds, creating medicines that improve our health, having a library 2 cm thick, etc. All this progress has originated, among many other things, thanks to technology.

Technology presents the great evolution in the paradigm of our society. Not only because it is a beast that grows by itself and feeds itself, but also because the very pace of its evolution is above that of the human being. What 50 years ago was a giant machine to perform calculations, today is a prosthesis on which we depend more and more: the smartphone. But such a level of subtlety makes up a complex web of logical sentences: software.

Software is the brain present in every technological device. It is what makes it work and on which, in a certain sense, we depend. So one might ask: If it is something so delicate and essential of those elements on which we end up depending or relying on, how do we know that it works properly? Specifically, how do we know that it performs the expected behavior?

1.1 Testing

Software testing consists of executing a program on a known pre-selected set (test suite) of inputs (test cases) and inspecting whether the outputs match respect the expected results. This process validates semantic properties of a program's behavior.

In other words, a **test** consists of a set of executions of 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. A test has an "incomplete" result if a series of tests produces no errors but the series is not complete under the coverage metric.

Although this is a straightforward definition, an even more interesting point of view is the following.

Testing tries to find counter-examples and choosing the test cases to this effect is often a difficult task. The tester who usually know most of those cases which produces 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 testing methodology that addresses this need. 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 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 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. [**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**]

1.2 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