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

Rafael Fernández Ortiz

February 28, 2023

Abstract

Building *generators* to create test cases that Property-Based Testing use to test software behavior is a hard, quite costly, and error-prone task. Even though most Property-Based Testing frameworks cover simple scenarios, there is some kind of issues with preconditioned generated test cases.

In the context of Property-Based Testing for strong and statically well-typed languages, such as Haskell or Scala, we propose an approach to relieve the programmer from the task of writing generators.

Our approach consists in provide an efficient and automatic generation of input test values that satisfy a given specification. In particular, we consider the case when the input values are algebraic data types satisfying complex constraints. The generation process is performed by writing specification expressions driven by the language' syntax and via symbolic execution using constraint logic programming.

Contents

Chapter 1

Introduction

Software is a set of instructions that tell a computer what to do. It can be used for various 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.

Software risks can come from various 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 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 it functions as intended and to identify and fix 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 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. Unit testing aims 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 software components 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 bugs 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.

Test Driven Development or Correctness by Constructions?

Also, during the life-cycle of software development has used several known techniques in order to get free-bugs software in an efficient and proper way. The most common paradigm, which is also the most natural way, is **Test Driven Development** (a.k.a TDD).

TDD is a software development technique in which tests are developed before the code, in short and incremental cycles. This technique proposes for the developer to create a new flawed test, and then to implement a little piece of code, in order to satisfy the current test set. Then, the code is

refactored if necessary, to provide a better structure and architecture for the current solution.

The challenge addressed in this work is to use TDD in applications with non-deterministic behavior as stated before. Although it is not possible to know exactly what the output will be, it is usually possible to check whether the generated output is valid or not.

The following factors make it difficult to develop randomized software using TDD:

- Results for each execution may be different for the same inputs, which makes it difficult to validate the return value.
- Obtaining a valid return for a test case execution does not mean that valid return will be delivered on the next executions.
- The random decisions and their paths number make it not viable to create Mock Objects that return fixed results for these decisions.
- It is difficult to execute a previous failed test with the same random decisions undertaken in its former execution.

In conclusion, **you have to iterate several times in case of testing fails**.

On the other hand, some techniques like **Correctness by Constructions**, try to formalize some specifications and build code based on them.

The idea is to start with a succinct specification of the problem, which is progressively evolved into code in small, tractable refinement steps. Experience has shown that the resulting algorithms are invariably simpler and more efficient than solutions that have been hacked into correctness. Furthermore, such solutions are guaranteed to be correct (i.e. they are guaranteed to comply with their specifications) in the same sense that the proof of a mathematical theorem is guaranteed to be correct. Here you don't have to iterate too as other ones, **but the formalized process is, in general, a complex task**.

For many reasons, in order to get a good enough solution for testing, Property-Based Testing was coming up.

1.2 Property-Based Testing

Property-based testing (a.k.a PBT) 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 an automated way to test the software in a wide range of scenarios using randomly generated test cases. 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 randomly generated inputs in PBT 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.

PBT 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.

The specification of one or more properties is the driver of the testing process, which assures that the given program meets the stated property, leaving aside the task of generating valid inputs.

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*.

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. PBT validates that the final product is free of specific flaws. Because PBT concentrates on generic flaws, it is ideal for focusing on analysis late in the development cycle after program functionality has been established.

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.

PBT 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.

When is useful to use Property-Based testing?

Property-Based testing can be used for 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.

However, stateful properties which would be the most interesting properties to test, are the challenging tasks at most times, and PBT, as we will see, cannot deal well with that.

1.3 Problem: Test cases that satisfy a given specification

Property-Based testing provides a helpful solution to generate random test cases for testing. And it is good enough for most pieces of code that want to test. However, we will put focus on those functions that assume inputs with preconditions.

Example 1.3.1 (Ordered Lists)**.** Let S be a set, `xs` a list of elements of S, and consider the ordering relationship $\preceq$ over elements of S. We can say that xs is an **ordered list** if it holds one of the following invariants:

`INV1` xs is empty.

`INV2` $\forall xss$ sublist of xs with $xss \neq \emptyset$, $\exists a \in xss$ such that $\forall x \in xss, a \preceq x$ holds.

Let's suppose we want to test the behavior of our `insertOrdered` function which its expected behavior should be the following:

`PROP1` *Given an ordered list,* `insertOrdered` *inserts an element and its result is an ordered list.*

Listing 1.1: Insert an element in an ordered list

```
insertOrdered :: Ord a => a -> [a] -> [a]
insertOrdered a [] = [a]
insertOrdered a xs'@(x:xs)
  | a <= x     = a : xs'
  | otherwise = x : insertOrdered a xs
```

Listing 1.2: Insert an element in an ordered list (Scala version)

```
def insertOrdered[A <: Ordered[A]]: A => List[A] => List[A] =
  (a: A) => {
    case Nil => List(a)
    case as@ ::(x, xs) => if (a <= x) a :: as else x :: insertOrdered(a)(xs)
  }
```

A Property-Based Testing framework should generate several enough random **ordered lists** to check if `PROP1` holds. This is not as easy as you could think. Let's do our own mental exercise step by step. Let's suppose we have a good PBT framework:

1. First of all, the framework has to generate randomly a set of lists.
2. Then, it has to check which one of them is an **ordered list**, i.e, it has to check if holds either `INV1` or `INV2`.
3. Finally checks if the property `PROP1` holds, which means the result has to be an **ordered list**, i.e. it has to check if holds either `INV1` or `INV2` too.

This process is more complex, but for this moment we can consider this friendly description. We will deeply get ahead in the following chapters.

Therefore, imagine that your PBT framework generates 100 randomly generated lists in every iteration. Probably, one or, if you have lucky, two lists of them are ordered lists.

On the one hand, it is so difficult to achieve so many scenarios (at least in a shorter time). Also, the framework probably brings you just the same empty list (because it is the easiest generated test case that holds one of the invariants) as the input value for checking the property which would make the results unreliable.

And on the other hand, the property `PROP1` is relatively simple but, what happens if we consider a more complex property? For example, we can consider a red-black tree.

Example 1.3.2 (Red-Black Tree)**.** A **Red-Black Tree** is a binary search tree where each node has two labels: a color `C`, which is either `red` (`R`) or `black` (`B`), and an integer `N`. For the purpose of test generation, node values are abstracted away in the definition of the data structure:

```
   C  ::=  R | B
   N  ::=  ... | -1 | 0 | 1 | ...
Tree  ::=  nil | C N Tree Tree
```

A Red-Black Tree must also satisfy the following three invariants:

`INV1` Every path from the root to a leaf has the same number of black nodes

`INV2` No red node has a red child and

`INV3` For every node n, all the nodes in the left (respectively, right) subtree of n, if any, have keys that are smaller (respectively, bigger) than the key labeling n.

Since red-black trees enjoy a weak form of balancing, operations such as inserting, deleting, and finding values are more efficient, in the worst case, than in ordinary binary search trees.

Let's suppose we want to test the behavior of our `insertOrderedRBTree` function which its expected behavior should be the following:

`PROP2` *Given a red-black tree,* `insertOrderedRBTree` *inserts an element and its result is a new tree which is a red-black tree.*

Listing 1.3: Insert an element in an Red-Black Tree

```
data Color = R | B deriving Show

data Tree a = Nil | T Color a (Tree a) (Tree a) deriving Show

makeBlack :: Tree a -> Tree a
makeBlack (T _ y a b) = T B y a b
makeBlack t = t

balance :: Tree a -> Tree a
balance T B z (T R y (T R x a b) c) d = T R y (T B x a b) (T B z c d)
balance T B z (T R x a (T R y b c)) d = T R y (T B x a b) (T B z c d)
balance T B x a (T R z (T R y b c) d) = T R y (T B x a b) (T B z c d)
balance T B x a (T R y b (T R z c d)) = T R y (T B x a b) (T B z c d)
balance t = t

insert :: (Ord a) => a -> Tree a -> Tree a
insert x s = makeBlack $ insertAux s
  where insertAux Nil  = T R x Nil Nil
        insertAux (T c y a b)
          | x < y  = balance T c y (insertAux a) b
          | x == y = T c y a b
          | x > y  = balance T c y a (insertAux b)
```

Listing 1.4: Insert an element in an Red-Black Tree (Scala version)

```
  sealed trait Color
  case object R extends Color
  case object B extends Color

  sealed trait Tree[A]
  case object Nil extends Tree[Nothing]
  case class T[A](color: Color, node: A, tl: Tree[A], tr: Tree[A]) extends Tree[A]


  def makeBlack[A]: Tree[A] => Tree[A] = {
    case ttree@T(_, _, _, _) => ttree.copy(color = B)
    case t => t
  }

  def balance[A]: Tree[A] => Tree[A] = {
    case T(B, z, T(R, y, T(R, x, a, b), c), d) => T(R, y, T(B, x, a, b), T(B, z, c, d))
    case T(B, z, T(R, x, a, T(R, y, b, c)), d) => T(R, y, T(B, x, a, b), T(B, z, c, d))
    case T(B, x, a, T(R, z, T(R, y, b, c), d)) => T(R, y, T(B, x, a, b), T(B, z, c, d))
    case T(B, x, a, T(R, y, b, T(R, z, c, d))) => T(R, y, T(B, x, a, b), T(B, z, c, d))
    case t => t
  }

  def insert[A <: Ordered[A]]: A => Tree[A] => Tree[A] =
    (x: A) => {
      def insertAux: Tree[A] => Tree[A] = {
        case Nil => T(R, x, Nil, Nil)
        case ttree@T(c, y, tl, tr) =>
          if (x < y) balance(T(c, y, insertAux(tl), tr))
          else if (x == y) ttree
          else balance(T(c, y, tl, insertAux(tr)))
      }

      makeBlack andThen insertAux
    }
```

Following the same reasoning that we did before, the reader can deduce how complex is the task.

In general, PBT is not prepared to generate inputs with preconditions and much fewer inputs with complex preconditions.

1.4 Our Approach

Building *generators* to create test cases that Property-Based Testing use to test software behavior is a hard, quite costly, and error-prone task. Even though most Property-Based Testing frameworks cover simple scenarios, there are many troubles with preconditioned generated test cases as we have just seen.

The approach we propose is (1) to build an efficient and automatic generator of input test values that satisfy a given specification and (2) a language syntax-driven bijection between the origin language's expressions and the constraint logic programming language ones. In particular, we will consider the case when the input values are Algebraic Data Types satisfying complex constraints. The generation process is performed via symbolic execution in the CLP language of the translated expressions of those ADTs and their specifications.

We will focus on the strong and static well-typed language Haskell and we will use Prolog as CLP language. Although it is well known that the mainly Property-Based Testing framework for Haskell is **QuickCheck**, and it has its own *generators*, we will provide steps to build a mechanism to generate those kinds of preconditioned input values. In particular, we will explore how to create a model that maps Haskell's ADT expressions and its specification to Prolog expression, generate symbolic Prolog expressions that hold the specifications and returns those expressions to Haskell.

1.5 Related Works

1.5.1 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 that 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**]

1.5.2 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**]

1.5.3 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 a property-based testing tool, designed to test programs written in the Erlang programming language. Its focus is on testing the behavior 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**]

1.5.4 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**]

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