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

Rafael Fernández Ortiz

May 17, 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 (Sorted 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.*

The following code is inspired by the Haskell ADT definition of red-black trees in [6].

Listing 1.3: Definition of Red-Black Tree ADT

```
data Color = R | B deriving Show

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

Listing 1.4: Insert an element in an Red-Black Tree

```
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 1.5: Definition of Red-Black Tree ADT (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]
```

Listing 1.6: Insert an element in an Red-Black Tree (Scala version)

```
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) **provide a language syntax-driven bijection between the origin language's expressions and the constraint logic programming language ones** which can translate and map expression from one language to other. 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.

Following the recent state-of-the-art approach, we use a *declarative* language such as Haskell following those works that are based on using CLP languages as generators. Being a declarative language allows us to separate the process of defining the algebra data type structure expression and their constraints from their semantics or instances. This separation helps improve the correctness of the developed test case generators, which implies that we can separate between *what* we want to generate in an expressive way.

But the main reason is the nature of being static-typed. It allows us to reason about types in a safe way and also we gain in static type checking in compile time, which adds us a plus about program verification. Therefore, Haskell allows to support us the fact that algebraic data types expressions are their own type (the 'ADT' type that corresponds), and then, intuitively and implicitly, we can assume and reason the ADT's invariants holds thanks to the fact that they are from that type (the 'ADT' type).

In contrast to the recent works that use Z3 or some SMT solver such as a test generator and output test validator. We use constraint logic programming language such as our test generator.

This approach fits better with our purpose thanks to how flexible, expressive, and descriptive the syntax of both languages Haskell and Prolog are. In fact, we show how natural is to define an algebraic data type in Haskell and find an expression 'equivalent' (in the sense of translation) in Prolog. This leads us to find a mechanism that translates both languages' expressions which also can fit so well with the Erlang language.

However, in contrast to the recent state-of-the-art research and regarding to the mentioned Erlang language, we use **static-typed and strong-typed** language.

The use of dynamic language could provocate non-safety in its type checking, and therefore, we cannot ensure what type is working when we deal with a given formal expression. For instance, in Erlang we can have both `adt` : τ_1 and `adt` : τ_2, which their formal expression is `adt`. When we generate test cases for that expression and come back to Erlang, we could provide wrong test input values.

There seems to be some controversy about the advantages of static vs dynamic typing. Type-checking provides another barrier against nonsensical programs. Moreover, whereas in a dynamically typed language, type mismatches would be discovered at runtime, in strongly typed statically checked languages type mismatches are discovered at compile time, eliminating lots of incorrect programs before they have a chance to run.

> Languages without type systems or even safe, dynamically checked languages, tend to offer features or encourage programming idioms that make type-checking difficult or infeasible.
> ...
> The assertion that types should be an integral part of a programming language is separate from the question of where the programmer must physically write down type annotations and where they can instead be inferred by the compiler. A well-designed statically typed language will never require huge amounts of type information to be explicitly and tediously maintained by the programmer. [7]

In Haskell, the definition of the ADT expression is itself the declaration of the type to which it belongs. That means, we cannot type two ADT with the same expression and try belonging on different types. There is a univocal relationship between the ADT's expression and the its type.

Finally, for the purpose of this work, we will illustrate all these concepts and ideas by applying them while writing a generator for Red-Black Trees. As we defined before:

Definition (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.

The definition of this structure, which holds a set of complex invariants, can be defined as an algebraic data type in Haskell as follow:

Listing 1.7: Definition of Red-Black Tree ADT

```
data Color = R | B deriving Show

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

We can deduce that the expression `Nil` and `T Color a (Tree a) (Tree a)` in Haskell can be mapped to the expressions `nil` and `t(C,X,L,R)` in Prolog, respectively; where `C` is the color (either `red` or `black`), and both `L`, `R` are red-black trees too, i.e., they can be expressed as `nil` or `t(C',X',L',R')`. **This brings us light on our hard way, don't you?**. Let's see an example before starting this work.

Definition (Sorted 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.

First of all, we have to identify the Haskell lists' syntax expression, that is:

A list expression in Haskell could be either `[]` or `x:xs` syntax. Therefore, the list ADT could be expressed in Prolog as follows `[]` or `X:Xs`, respectively.

Then, in order to provide the invariants `INV1` and `INV2` and get a definition of a sorted list in Prolog, we should translate in someway these two (written in Haskell or, more precisely written in Liquid Haskell) or type directly in scratch the restrictions in Prolog to get the following expressions or similar:

`INV1` `sorted_list([])` which holds that an empty list is a sorted list.

`INV2` `sorted_list(_:[])` which holds that a one-element list is a sorted list.

`INV2` `sorted_list(X1:X2:Xs) :- X1 #=< X2, sorted_list(X2:Xs)` which means that a list is a sorted list if it holds that every two elements hold an ordering relationship and the tail of the list is a sorted list too.

In resume, we should map the Haskell definition of a sorter list to some Prolog definition like this:

Listing 1.8: Sorted List in Prolog

```
:- use_module(library(clpfd)).

%% Base case for an empty list
sorted_list([]).

%% Recursive case for a non-empty list
sorted_list(_:[]).
sorted_list(X1:X2:Xs) :- X1 #=< X2, sorted_list(X2:Xs).
```

With this implementation, we can get the following test cases using, for instance, the Prolog-ClI prompt and the expression `sorted_list`. However, readers can think that modeling list adt and the invariants for sorted list could be a little bit easy. Let's see a final challenge that shows interest in this approach.

Definition (Binary Search Tree)**.** A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties.

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than or equal to its parent node's key.

A Binary Search Tree must also satisfy the following invariants:

`INV1` An empty tree is a binary search tree.

`INV2.1` The left sub-tree is a binary search tree.

`INV2.2` The right sub-tree is a binary search tree.

Here we can see that invariants not only are more complex than the sorted list example but also it holds recursively onto their inner structures. Therefore, getting a test case for this example is a hard task. But we can do as before. First of all, we have to identify the Haskell bst' syntax expression, that is:
A bst expression in Haskell could be either `Nil` or `T a BSTree a BSTree a` syntax. Therefore, the bst ADT could be expressed in Prolog as follows `nil` or `t(N,L,R)`.

Then, in order to provide the invariants `INV1`, `INV2.1` and `INV2.2` and get a definition of a bst in Prolog, we do as similar as before to get the following expressions or similar:

`INV1` `bst(nil)` which holds that an empty tree is a binary search tree.

`INV2.1` **and** `INV2.2` `bst(t(_,nil,nil))` defines a non-empty tree with no left or right subtrees, which holds that a one-element tree is a binary search tree.

`INV2.1` `bst(t(N,L,nil)) :- L = t(LN,_,_), LN #< N, bst(L)` defines a non-empty tree with a left subtree and no right subtree.

`INV2.2` `bst(t(N,nil,R)) :- R = t(RN,_,_), RN #> N, bst(R)` defines a non-empty tree with a right subtree and no left subtree.

`INV2.1` **and** `INV2.2`

```
bst(t(N,L,R)):-L = t(LN,_,_), LN #< N, bst(L)
               R = t(RN,_,_), RN #> N, bst(R)
```

defines a non-empty tree with both left and right subtrees.

In resume, we should map the following Haskell definition of a binary search tree:

Listing 1.9: Binary Search Tree in Haskell

```haskell
data BSTree a = Nil | T (BSTree a) (BSTree a) deriving Show
```

to some Prolog definitions like this:

Listing 1.10: Sorted List in Prolog

```
:- use_module(library(clpfd)).

% Base case for an empty tree
bst(nil).

% Recursive case for a non-empty tree
bst(t(_,nil,nil)).
bst(t(N,L,nil)) :- L = t(LN,_,_), LN #< N, bst(L).
bst(t(N,nil,R)) :- R = t(RN,_,_), RN #> N, bst(R).
bst(t(N,L,R)) :- L = t(LN,_,_), LN #< N, bst(L), R = t(RN,_,_), RN #> N, bst(R).
```

1.5 Related Works

1.5.1 QuickCheck: Automatic testing of Haskell programs

QuickCheck [8] 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

Efforts to provide best-fitted generators to complex constrained test cases are becoming a very current research topic. It is faced from some perspectives.

Regarding to Fioravanti et al. research works [1], [2] and [3], we can observe a long projection to provide a similar approach. Indeed, in [1] and [2], they based on the idea to relieves the developer from the task of writing data generators of valid inputs using CLP languages. They show how using CLP languages for that propose and prove **how many benefits and how efficient** is to use these kinds of logic programming languages as test generators.

Finally, in [3], they bring us some kinds of best-performances using program transformation instead of the classical and native left-to-right strategy implemented by standard CLP systems which can very inefficient. These works, in essence, propose a framework based on Constraint Logic Programming for the systematic development of generators of large sets of structurally complex test data structures.

They adopt a declarative approach too following the so-called *constrain-and-generate* approach. However, they focus on the **dynamically typed** functional programming language Erlang and refer to how arduous is the task of writing a generator that satisfies all constraints defined by a filter for the PropEr framework.

Ricardo Peña et al. research works in [4] propose a system that can automatically generate an exhaustive set of black box test cases, up to a given size, for programs under test requiring complex preconditions. The key of this research is how they translate a formal precondition into a set of constraints belonging to the decidable logic of SMT solvers. They use an SMT solver as a test case generator to directly generate test cases satisfying a complex precondition. Also, they describe both black-box and white-box approaches to fit searching strategies for those test cases.

Chapter 2

Preliminaries

In this section, we present the basic notions of the Haskell and Prolog languages. Both will be the main actors of this work and before starting we need to bring some context about both languages, identifying which one will be our scope and what will need from each one.

2.1 Haskell, the functional guy

Haskell is a purely functional programming language, lazy, statically typed, concurrent, type inference, and why not, elegant and concise language.

Purely functional programming. Every function in Haskell is a function in the mathematical sense (i.e., "pure"). Even side-effecting IO operations are but a description of what to do, produced by pure code. There are no statements or instructions, only expressions which cannot mutate variables (local or global) nor access state like time or random numbers. The following function takes an integer and returns an integer. By the type it cannot do any side-effects whatsoever, it cannot mutate any of its arguments.

Listing 2.1: Purely Functional Programming

```
square :: Int -> Int
square x = x * x
```

The following string concatenation is a type error:

Listing 2.2: Purely Functional Programming

```
"Name: " ++ getLine -- error
```

Because `getLine` has type `IO String` and not `String`, like "Name: " is. So by the type system you cannot mix and match purity with impurity.

Lazy. Functions don't evaluate their arguments. This means that programs can compose together very well, with the ability to write control constructs (such as if/else) just by writing normal functions. The purity of Haskell code makes it easy to fuse chains of functions together, allowing for performance benefits.

Statically Typed. Every expression in Haskell has a type which is determined at compile time. All the types composed together by function application have to match up. If they don't, the program will be rejected by the compiler. Types become not only a form of guarantee, but a language for expressing the construction of programs. All Haskell values have a type:

Listing 2.3: Statically Typed

```
char = 'a'     :: Char
int = 123      :: Int
fun = isDigit :: Char -> Bool
```

You have to pass the right type of values to functions, or the compiler will reject the program:

Listing 2.4: Statically Typed

```
isDigit 1 -- error
```

Concurrent. Haskell lends itself well to concurrent programming due to its explicit handling of effects. Its flagship compiler, GHC, comes with a high-performance parallel garbage collector and light-weight concurrency library containing a number of useful concurrency primitives and abstractions. Easily launch threads and communicate with the standard library:

Listing 2.5: Concurrent

```
main = do
  done <- newEmptyMVar
  forkIO (do putStrLn "I'm one thread!"
             putMVar done "Done!")
  second <- forkIO (do threadDelay 100000
                       putStrLn "I'm another thread!")
  killThread second
  msg <- takeMVar done
  putStrLn msg
```

Use an asynchronous API for threads:

Listing 2.6: Concurrent

```
do a1 <- async (getURL url1)
  a2 <- async (getURL url2)
  page1 <- wait a1
  page2 <- wait a2
  ...
```

Type Inference. You don't have to explicitly write out every type in a Haskell program. Types will be inferred by unifying every type bidirectionally. However, you can write out types if you choose, or ask the compiler to write them for you for handy documentation. This example has a type signature for every binding:

Listing 2.7: Type Inference

```
main :: IO ()
main = do line :: String <- getLine
          print (parseDigit line)
  where parseDigit :: String -> Maybe Int
        parseDigit ((c :: Char) : _) =
          if isDigit c
             then Just (ord c - ord '0')
             else Nothing
```

But you can just write:

Listing 2.8: Type Inference

```
main = do line <- getLine
          print (parseDigit line)
  where parseDigit (c : _) =
          if isDigit c
             then Just (ord c - ord '0')
             else Nothing
```

Elegant and concise. Because it uses a lot of high-level concepts, Haskell programs are usually shorter than their imperative equivalents. And shorter programs are easier to maintain than longer ones and have less bugs.

2.1.1 Grammar specification

For our purpose, it is necessary to identify which is the specification of Haskell's grammar. More precisely, which is the specification of the definition of an algebraic data type in Haskell. Then, we need to identify which elements participate in the specification, and, in the next chapter, we will see how we need to translate those ones elements to Prolog elements within a complete formal Prolog expression. Therefore:

Let's suppose we want to formally define an algebraic data type. Regarding [9] and [10], we can consider a subset of the grammar specification (expressed in BNF-Like syntax) for defining an ADT expression in Haskell. In our case, we won't consider either `context` or `deriving` words because we want just an approach to simple algebraic data type expressions. Therefore, for our purpose and for simplicity, we have considered the following subset.

special ::=	`(` | `)` | `,` | `;` | `[` | `]` | `` ` `` | `{` | `}`	
reservedop ::=	`..` | `:` | `::` | `=` | `\` | `|` | `<-` | `->`	
	| `@` | ∼ | `=>`	
ascDigit ::=	`0` | `1` | ... | `9`	
ascSmall ::=	`a` | `b` | ... | `z`	
ascLarge ::=	`A` | `B` | ... | `Z`	
ascSymbol ::=	`!` | `#` | `$` | `%` | `&` | `⋆` | `+` | `.` | `/` | `<`	
	| `=` | `>` | `?` | `@` | `\` | `^` | `|` | `-` | ∼	
uniDigit ::=	Any Unicode decimal digit	
uniSmall ::=	Any Unicode lowercase letter	
uniLarge ::=	Any uppercase or titlecase Unicode letter	
uniSymbol ::=	Any Unicode symbol or punctuation	
digit ::=	*ascDigit* | *uniDigit*	
small ::=	*ascSmall* | *uniSmall* | `_`	
large ::=	*ascLarge* | *uniLarge*	
symbol ::=	*ascSymbol* | $uniSymbol_{<special \mid \_ \mid : \mid " \mid '>}$	
reserveid ::=	`case` | `class` | `data` | `default` | `deriving` | `do` | `else`	
	| `if` | `import` | `in` | `infix` | `infixl` | `infixr` | `instance`	
	| `let` | `module` | `newtype` | `of` | `then` | `type` | `where` | `_`	
varid ::=	$(\ small\ \{\ small \mid large \mid digit \mid \text{'}\ \})_{<reserveid>}$	
conid ::=	*large* { *small* | *large* | *digit* | `'` }	(constructors)
consym ::=	$(\ \texttt{:}\ \{\ symbol \mid \texttt{:}\ \}\ )_{<reservedop>}$	
tyvar ::=	*varid*	(type variables)
tycon ::=	*conid*	(type constructors)
simpletype ::=	$tycon\ \ tyvar_1\ \ \ldots\ \ tyvar_k$	($k \geq 0$)
modid ::=	*conid*	(modules)
qtycon ::=	[*modid* `.`] *tycon*	
gtycon ::=	*gtycon*	
	| `()`	(unit type)
	| `[]`	(list constructor)
	| `->`	(function constructor)
	| `(` `,` { `,` } `)`	(tupling constructor)

btype ::=	[*btype*] *atype*	(type application)
type ::=	*btype* [-> *type*]	(function type)
atype ::=	*gtycon*	
	| *tyvar*	
	| $(type_1, ..., type_k)$	(tuple type, $k \geq 2$)
	| [*type*]	(list type)
	| (*type*)	(parenthesized constructor)
con ::=	*conid* | (*consym*)	(constructor)
constr ::=	*con* [!] $atype_1 \ldots$ [!] $atype_k$	(arity *con* = k, $k \geq 0$)
constrs ::=	$constr_1$ | ... | $constr_n$	($n \geq 0$)
topdecl ::=	`data` *simpletype* `=` *constrs*	

2.2 Prolog, the logical guy

Prolog is a logic programming language associated with artificial intelligence and computational linguistics.

Prolog has its roots in first-order logic, a formal logic, and unlike many other programming languages, Prolog is intended primarily as a declarative programming language: the program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a query over these relations.

A logic program is a set of axioms, or rules, defining relations between objects. A computation of a logic program is a deduction of consequences of the program. A program defines a set of consequences, which is its meaning The art of logic programming is constructing concise and elegant programs that have the desired meaning.

Prolog is a practical and efficient implementation of many aspects of "intelligent" program execution, such as non-determinism, parallelism, and pattern-directed procedure call. It provides a uniform data structure called the *term*, from which all data, as well as Prolog programs, are constructed. A Prolog program consists of a set of clauses, where each clause is either a fact about the given information or a rule about how the solution may relate to or be inferred from the given facts. Thus Prolog can be seen as a first step towards the ultimate goal of programming in logic.

In another way, in Prolog, a program logic is expressed in terms of relations, and a computation is initiated by running a query over these relations. Relations and queries are constructed using Prolog's single data type, the *term*. Relations are defined by *clauses*. Given a query, the Prolog engine attempts to find a resolution refutation of the negated query. If the negated query can be refuted, i.e., an instantiation for all free variables is found that makes the union of clauses and the singleton set consisting of the negated query false, it follows that the original query, with the found instantiation applied, is a logical consequence of the program. This makes Prolog (and other logic programming languages) particularly useful for database, symbolic mathematics, and language parsing applications.

2.2.1 Getting Started

Let's show the essential elements of the Prolog in real programs, but without becoming diverted by details, formal rules, and exception, before getting started with its specification.

Computer programming in Prolog consists of:

- Specifying some *facts* about object and their relationships
- Defining some *rules* about objects and their relationships
- Asking *questions* about objects and their relationships

For example, Let's suppose we told to Prolog system our rule about the *father relationship*. Let's suppose that Anakin' son is Luke, Julio's son is Anakin and there is a fourth guy, Manuel, who is just Luke's friend. Therefore, we can observe that just Anakin is related to Luke and Julio is related to Anakin by the *father relationship*. These logical rules can be written in Prolog as follow:

Listing 2.9: Father Relationship

```
father(anakin, luke).
father(julio, anakin).
```

And if we type in Prolog-CLI `father(anakin, luke)` it returns `true`, also with `father(julio, anakin)`. However, if we try to type `father(julio, manuel)` or whatever with Manuel, (or any other combination that does not represent the relationship written before) it will return `false`.

Facts

The simplest kind of statement is called a *fact*. Facts are a means of stating that a relation holds between objects. In the example above, the expression `father(anakin, luke)` is a fact which says that `anakin` is the *father* of `luke`. Names of the individuals are known as *atom*s.

The *plus* relation can be realized via a set of facts that defines the addition table. An initial segment of the table is:

Listing 2.10: Plus Relationship

```
plus(0, 0, 0).
plus(0, 1, 1).
plus(1, 0, 1).
plus(1, 1, 2).
plus(0, 2, 2).
plus(2, 0, 2).
```

This is not the best approach to defining *plus* relation but it is good enough to catch the idea.

Queries

The second form of a statement in Prolog is a *query*. Queries are a means of retrieving information from a logic program. A query asks whether a certain relation holds between objects. For example, the query `father(anakin, luke)?` asks whether the *father* relationship holds between `anakin` and `luke`. Given the facts written before, the answer to this query is *yes*.

The Logical Variable, Substitution, and Instances

A logical variable stands for an unspecified individual and is used accordingly. Consider its use in queries. Suppose we want to know of whom `luke` is the father. One way is to ask a series of queries, `father(julio, luke)?`, `father(manuel, luke)?`, `father(anakin, luke)?`, until an answer *yes* is given.

A variable allows a better way of expressing the query as `father(X, luke)?` to which the answer is `X=anakin`. Used in this way, *variables are a means of summarizing many queries*. A query containing a variable asks whether there is a value for the variable that makes the query a logical consequence of the program.

Definition (Variable)**.** *Variables* in logic programs behave differently from variables in conventional programming languages. They stand for an unspecified but single entity rather than for a store location in memory.

When Prolog uses a variable, the variable can be either *instantiated* or *not instantiated*

Definition (Instantiation)**.** A variable is *instantiated* when there is an object that the variable stands for. A variable is not instantiated when what the variable stands for is not yet known.

Having introduced variables, we can define a *term*.

Definition (Term)**.** A *term* is the single data structure in logic programs. The definition is inductive. Constants and variables are terms. Also, compound terms or structures are terms.

Prolog can distinguish variables from terms because any term beginning with a capital letter is taken to be a variable.

Definition (Compound Term)**.** A *compound term* comprises a **functor** (called the principal functor of the term) and a sequence of one or more arguments, which are terms. A *functor* is characterized by its name, which is an atom, and its arity or number of arguments.

Sintatically, compound terms have the form $f(t_1, t_2, \ldots, t_n)$, where the functor has name f and is of arity n and the t_i with $1 \leq i \leq n$, are the arguments.

Examples of compound terms include `s(0)`, `hot(milk)`, `name(john,doe)`, `list(a,list(b,nil))`, `foo(X)`, and `tree(tree(nil,3,nil) ,5,R)`.

Queries, goals, and more general terms where variables do not occur are called *ground*. Where variables do occur, they are called *nonground*. For example, `foo(a, b)` is *ground*, whereas `bar(X)` is *nonground*.

Definition (Substitution)**.** A *substitution* is a finite set (possibly empty) of pairs of the form $X_i = t_i$, where X_i is a variable and t_i is a *term*, and $X_i \neq X_j$, for every $i \neq j$, and X_i does not occur in t_j, for any i and j.

An example of a substitution consisting of a single pair is `X=anakin`. Substitution can be applied to terms. The result of applying a substitution σ to a term W, denoted by $W\sigma$, is the term obtained by replacing every occurrence of X by t in W, for every pair $X = t$ in σ.

The result of applying `X=anakin` to the term `father(X, luke)` is the term `anakin, luke)`.

Definition (Instance)**.** X is an *instance* of W if there is a substitution σ such that $X = W\sigma$

The goal `father(anakin, luke)` is an instance of `father(X, luke)` by definition. Similarly, `plus(1,0,1)` is an (one of them) instance of `plus(1,Y,X)`.

Those definitions are enough to encourage to talk about the grammar of Prolog.

Rules

Conjunctive queries are defining relationships in their own right. We can add to our facts the following table:

Listing 2.11: Force-Side Relationship

```
sith(anakin).
jedi(luke).
jedi(julio).
```

The query `father(X,Y),sith(X)?` is asking for a *father* which is also a *sith*.
The query `father(anakin,X),jedi(X)?` is asking if there exists an `anakin`'s son who is also a `jedi`.

This brings us to the third and most important statement in logic programming, a *rule*, which enables us to define new relationships in terms of existing relationships.

Definition (Rules)**.** *Rules* are statements of the form

$$A \leftarrow B_1, B_2, \ldots B_n$$

where $n \geq 0$.

The goal A is the *head* of the rule, and the conjunction of goals $B_1, B_2, \ldots B_n$ is the body of the rule.

Rules, fats, and queries are also called *Horn clauses* or *clauses* for short. Note that a fact is just a special case of a rule when $n = 0$. Facts are also called *unit clauses*. We also have a special name for clauses with one goal in the body, namely, when $n = 1$. Such a clause is called an *iterative clause*. As for facts variables appearing in rules are universally quantified, and their scope is the whole rule.

A rule expressing the *enemy* relationship could be

$$\texttt{enemy(X,Y)} \leftarrow \texttt{jedi(X),sith(Y).}$$

Similarly one can define a rule expressing the *master* relationship

$$\texttt{master(X,Y)} \leftarrow \texttt{sith(X),sith(Y).}$$

Rules can be viewed in two ways. First, they are a means of expressing new or complex queries in terms of simple queries. A query `enemy(luke,Y)?` to the program that contains the preceding rule for `enemy` is translated to the query `jedi(luke),sith(Y)?` according to the rule. A new query about the `enemy` relationship has been built from simple queries involving `jedi` and `sith` relationships. Interpreting rules in this way is their *procedural* reading.

The second view of rules comes from interpreting the rule as a logical axiom. The backward arrow $\leftarrow$ is used to denote logical implication. The `enemy` rule reads: *X is an* `enemy` *of Y if X is a* `jedi` *and Y is a* `sith`.

Although formally all variables in a clause are universally quantified, we will sometimes refer to variables that occur in the body of the clause, but not in its head, as if they are existentially quantified inside the body.

For example, the `enemy` rule can be read: "For all X and Y, X is an enemy of Y if there exists a couple of X and Y, such that X is a Jedi and Y is a Sith". The formal justification for this verbal transformation will not be given, and we treat it just as a convenience.

Definition. The law of *universal modus ponens* says that from the rule

$$R = (A \leftarrow B_1, B_2, \ldots B_n)$$

and the facts B'_1, B'_2, ... B'_n, A' can be deduced if

$$A' \leftarrow B'_1, B'_2, \ldots B'_n$$

is an instace of R.

Definition. A *logical program* is a finite set of rules.

Definition. An existentially quantified goal G is a logical consequence of a program P if there is a clause in P with a ground instance $A \leftarrow B_1, B_2, \cdot B_n$ with $n \geq 0$ such that $B_1, B_2, \cdot B_n$ are logical consequences of P, and A is an instance of G.

Recursive Rules

The rules described so far define new relationships in terms of existing ones. An interesting extension is recursive definition of relationships that define relationships in terms of themselves. One way of viewing recursive rules is a generalization of a set of nonrecursive rules.

Consider the problem of testing connectivity in a directed graph. A directed can be represented as a logic program by a collection of facts. A fact `edge(Node1, Node2)` is present in the program if there is an edge from `Node1` to `Node2` in the graph. Let's suppose the following directed graph program:

Listing 2.12: A direct graph

```
edge(a,b).   edge(b,d).   edge(d,e).
edge(a,c).   edge(c,d).   edge(f,g).
```

Two nodes are connected if there is a series of edges that can be traversed to get from the first node to the second. That is, the relation `connected(Node1, Node2)` is `true` if `Node1` and `Node2` are connected, is the transitive closure of the `edge` relation. We could represent the connected relationship as follow:

Listing 2.13: Connected relationship

```
connected_one(N1, N2) <- edge(N1, N), edge(N, N2).
```

However, this just shows the relationship of *one-path-connexion*, i.e. the rule `connected_one(a,c)` returns `true`, but `connected_one(a,d)` returns `false`. We should define again a new one to reach the node `d` in our *connected* relationship.

Listing 2.14: Connected relationship

```
connected_one(N1, N2) <- edge(N1, N), edge(N, N2).
connected_two(N1, N2) <- edge(N1, N), connected_one(N, N2).
```

Again, we have now the *one-path-connexion* and *two-path-connexion* relationships, i.e. the rule `connected_one(a,c)` returns `true`, `connected_two(a,d)` returns `true` but we don't have any way to reach the node `e`.

A clear pattern can be seen, which can be expressed in a rule defining the relationship `connected(Node1, Node2)` recursively.

Listing 2.15: Generalization of the connected relationship

```
connected(N,N).
connected(N1, N2) <- edge(N1, N), conected(N, N2).
```

2.2.2 Grammar specification

This section will be short. Prolog is hugely flexible with respect to its grammar and the expressions that one can define. This is really helpful for doing syntax transformation between Haskell to Prolog, and vice versa.

In the next chapter, we will see step by step how we should define the primitive type generators, then we will talk about how each part of the ADT specification should be translated into Prolog expressions, and finally, we will define a generator for the ADTs.

Chapter 3

Syntax-Translation Mechanism

Bibliography

[1] E. De Angelis, F. Fioravanti, A. Palacios, A. Pettorossi and M. Proietti. (2019). Property-Based Test Case Generators for Free. 10.1007/978-3-030-31157-5_12.

[2] V. Senni and F. Fioravanti. (2012). Generation of Test Data Structures Using Constraint Logic Programming. 7305. 115-131. 10.1007/978-3-642-30473-6_10.

[3] F. Fioravanti, M. Proietti and V. Senni. (2015). Efficient generation of test data structures using constraint logic programming and program transformation. Journal of Logic and Computation, vol. 25, no. 6, pp. 1263-1283, doi: 10.1093/logcom/ext071.

[4] R. Peña, J. Sánchez-Hernández, M. Garrido and J. Sagredo. (2019) SMT-based Test-Case Generation with Complex Preconditions. Actas de las XIX Jornadas de Programación y Lenguajes (PROLE 219).

[5] O. Kiselyov and C. Shan. (2008). Interpreting types as abstract values. Formosan Summer School on Logic, Language, and Computation.

[6] C. Okasaki. (1999). Red-black trees in a functional setting. Journal of Functional Programming, 9(4), 471-477. doi:10.1017/S0956796899003494

[7] Benjamin C. Pierce. (2002). Introduction. Types and Programming Languages, MIT Press, 2002, pp.1-13.

[8] K. Claessen and J. Hughes. (2000). QuickCheck: a lightweight tool for random testing of Haskell programs. SIGPLAN Not. 35, 9, 268–279. https://doi.org/10.1145/357766.351266

[9] Haskell. Haskell 98. Syntax. https://www.haskell.org/onlinereport/syntax-iso.html

[10] Haskell. Haskell 98. User-Defined Datatypes. https://www.haskell.org/onlinereport/decls.html