ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Randomised testing of optimising code substitutions

Master's Thesis

Jérôme Amiguet

LAMP

Professor
Martin Odersky

Supervisor
Dmitry Petrashko

February 20, 2017

Abstract

Extending compiler with domain specific optimisation by defining rewrite rules is a very powerful tool that is easy to misuse. Rules can be wrong in multiple subtle ways: either them not being an optimisation, or returning a different output on some input or by not keeping side effects. This thesis proposes an automatic testing engine for rewrite rules. The framework was used to verify rewriting rules on Scala collections.

Contents

1 Introduction

The purpose of this project is to provide a framework that uses randomised tests to verify the correctness of code substitution rules, defined by using the `@rewrites` mechanism. We will show the benefits to have the ability to define code substitution rules. After all, we want to have a code that is kept simple, without hindering its efficiency.

We will provide the definition of a set of simple rules using the rewriting mechanism. We will give a short explanation on the rewriting system. We will show how to use the mechanism to apply the rules.

In order to be practical, rules should satisfy some requirements. We will define the requirement as *correctness, efficiency* and *keeping side effects*. Which will lead us to the necessity of verifying all the properties.

We will describe how we can test specific properties with the project. We will illustrate the strength of randomised tests to provide counter example. Which leads to the improvement and rectification of substituting rules.

Finally, we will discuss the tools that are used to build the project, in particular how we can work with scala.meta and the new macro paradigm to retrieve the information of the substitution to use them when defining the randomised tests. We use the ScalaCheck framework to define the randomised tests.

2 Motivation

Dotty[5] is a next generation compiler to test new technologies for Scala. One technology being developed is a *Rewrites* mechanism. It is used to define rules for code substitution at compile time. The principle is to define a pattern that might occur inside the code, and to provide the compiler with a replacement for that pattern. For instance, when we have the code of Listing 1, to improve the efficiency we want to replace the call of `xs.map(f1).map(f2)`, that

```scala
val xs: Seq[Int] = /*...*/
val f1(i: Int): Int = /*...*/
val f2(i: Int): Int = /*...*/
xs.map(f1).map(f2)
```

Listing 1: Motivation to rewrite the code

will go through each elements of the sequence twice, by the call `xs.map(x => f2(f1(x)))`, with an anonymous function that will apply both functions one after the other.

There are reasons for us to write the code as in the Listing 1. Separating a multi stage computation in reusable stages allows to improve the code readability and code comprehension. Splitting each transformation into simple steps make it easier to follow the program execution than with a convoluted function.

While in the isolated example of Listing 1, it this is is easy to find a more efficient code. When we are using a library, it might not be so obvious how to produce the most efficient code. The creators of the library may have a better knowledge on it and they can provide substitution rules with their expertise. A library with rewriting rules can be used in a simple, readable and understandable manner. It let the programmer rely on the substitution rules to improve the efficiency of his code.

3 Rewriting mechanism

As of now, the way to design and use rules is done through a branch of the Dotty compiler, maintained by Dmitry Petrashko, called Dotty-Linker[9, 10]. You have to use the macro annotation @rewrites and use the Rewrite object to define substitution rules. Listing 2 illustrates how to define a simple rule. The first argument to the object Rewrite, denoted here by *left side (from)*, is the original code and the second argument, denoted by *right side (to)* is the substituting code. When the compiler matches a syntax tree with the expression from the

```
import dotty.linker._

@rewrites
object Rules {
  def twoMaps(xs: Seq[Int], f1: (Int => Int), f2: (Int => Int)) =
    Rewrite(
      from = xs.map(f1).map(f2), // left side
      to   = xs.map(x => f2(f1(x))) // right side
    )
}
```

Listing 2: Two map rule

left side, it *binds* the parameters of the rule to the actual values. When the compiler observe the code of Listing 3, it will match that part of the syntax tree with the left side and it will binds myInts with xs, and the functions plusOne, double with f1 and f2 respectively. The

```
/*...*/
def plusOne(i: Int) = i + 1
def double(i: Int) = 2*i
val myInts = Seq(1, 2, 3, 4)

// rewritten as myInts.map(x => double(plusOne(x)))
myInts.map(plusOne).map(double)
/*...*/
```

Listing 3: Illustration of rewrites mechanism

syntax tree for the right side is an incomplete syntax tree with *holes* that have to be plugged. In our example they are xs, f1, and f2. Now that we have bound the actual values, they are plugged inside the second tree. The completed tree is substituting the original syntax tree inside the code.

Those kinds of rewrites are enabled by passing the -rewrites flag to the compiler from the Dotty-Linker branch.

4 Rules correctness criteria

The mechanism aims to improve the efficiency of the code. Beyond that, we have to make sure that the substituting code does not change the results of the execution. We aim to test the three properties: *correctness, efficiency* and *keeping side effects*.

4.1 Resulting value

For any rule, the results of each side is the same. The rules have to keep the semantic of the program. Let us look at the rule from the Listing 4. Intuitively both sides should yield the same results. That would be true if we were able to represent the infinity of integers. But as the integers are fixed bit-length, what could happen is that the sum of the two numbers is not

```
def twoDropRight(xs: Seq[Int], n: Int, m: Int) =
  Rewrite(
    xs.dropRight(n).dropRight(m),
    xs.dropRight(n + m)
  )
```

Listing 4: Two `dropRight` rule

within the bounds of the integers, and then the program throws an exception. Now we do not have always the same results. To fix that, we need to modify the rule and add guards for those corner cases, so if we want to have a rule that substitutes two `dropRight`, then we have the rule in the Listing 5.

```
def twoDropRight(xs: Seq[Int], n: Int, m: Int) =
  Rewrite(
    xs.dropRight(n).dropRight(m),
    {
      val n0 = Math.max(0, n)
      val m0 = Math.max(0, m)
      if ((n0/2 + m0/2) >= Int.MaxValue/2)
        Nil
      else
        xs.dropRight(n0 + m0)
    }
  )
```

Listing 5: Two `dropRight` rule with guards

4.2 Efficiency

The substitution should be more efficient than the original code. As we have seen with the Listing 5, we have to define some guards on corner cases to prevent bad behaviours. Whenever guards are added to a rule, we are adding extra steps. We want to check if these steps are too impactful on the program. Overall, the new code complexity has to be bounded by the complexity of the original code.

4.3 Side effects

Order of side effects

Some rules can have holes bound to expression that have side effects. The substituting code should yield the side effects in the same order as the original code. The order of side effects, in most cases, matter.

If we look at the rule form Listing 2, as long as either `f1` or `f2` do not have side effects, then the substitution is fine. If both of them have side effects, then the rule will change the order of side effects, as shown in the Listing 6.

Keeping side effects

Likewise, if any parameter has side effects we want the substituting code to produce the same side effects. If we look at the code of Listing 7, we get the same results but we can loose side effects. We are mapping the elements of `xs` only for the `n` first elements. The function `f` is not

```
val xs = Seq(1,2,3,4)
def f1(i: Int) = { print(i); 2*i }
def f2(i: Int) = { print(i); i + 2 }

xs.map(f1).map(f2) // print 12342468
xs.map(x => f2(f1(x))) // print 12243648
```

Listing 6: Different side effects order for the two maps rule

```
def mapAndTake(xs: Seq[Int], f: (Int => Int), n: Int) =
  Rewrite(
    xs.map(f).take(n),
    xs.iterator.map(f).take(n).toVector
  )
```

Listing 7: `map` and `take` rule

applied for the remaining elements of the sequence, which is different from the original code where all elements are mapped by the function. If `f` has side effects, we are discarding some of them. Listing 8 illustrates the loss of side effects. The discarded side effects, in most case,

```
def f(i: Int) = { print(i); i + 1 }
val xs = Seq(1,2,3,4)

xs.map(f).take(2) // print 1234
xs.iterator.map(f).take(2).toVector // print 12
```

Listing 8: Losing side effects

matter for the program. If we want to keep the side effects, then the rule should be written as in Listing 9.

```
def mapAndTake(xs: Seq[Int], f:(Int => Int), n: Int) =
  Rewrite(
    xs.map(f).take(n),
    {
      val it = xs.iterator
      val retVal = it.map(f).take(n).toVector
      it.map(f).toVector
      retVal
    }
  )
```

Listing 9: `map` and `take` rule keeping all side effects

5 Rules Tester

We have defined a set of properties that the rules must have. This project can help improving or finding counter examples for some rules. The macro add four inner classes to the object containing the rules:

Correctness check the correctness of each rules;

Efficiency check the efficiency of each rules;

FunctionSideEffects check side effects for each combination of function with side effects, for each rules;

GeneralSideEffects check side effects with self defined method.

All the specific tests are in separated classes as some may not be relevant for all set of rules. There can be some *a priori* on the arguments used by the rules, for instance all the functions are always pure making the side effects tests irrelevant. The collection of tests are within an inner class to give more flexibility. Other test can be added when defining the test object.

The tests of `GeneralSideEffects` requires the method of the trait defined in the Listing 10 to be implemented. As we do not know how the side effects are tested, we need the user of the framework to provide these informations.

```scala
trait TestFunctions {
  // Called at the start of the test
  def cleanUp(): Unit
  // Meant to observe the side effects of "from" (left)
  def commitLeft(): Unit
  // Meant to observe the side effects of "to" (right)
  def commitRight(): Unit
  // Verify the state of side effects from both sides
  def checkEffects(): org.scalacheck.Prop
}
```

Listing 10: Helpers when using self defined side effects

Let the object `Rules` contains the different rewriting rules. Activating the different tests can be done by defining the object as in Listing 11. As they are subclass of `Properties` from

```scala
/*...*/
object TestCorrectness extends Rules.Correctness

object TestEfficiency extends Rules.Efficiency

object TestEffects extends Rules.FunctionSideEffects
/*...*/
```

Listing 11: Activating the different collection of tests

ScalaCheck, they have a method that will check every properties when running the tests with `sbt`[13].

The GitHub repository `https://github.com/nolondil/Rewriting-Rules-Tests` host examples of rewriting rules and how to verify them.

5.1 Correctness

If we were to look at the initial rules for two `dropRight` in Listing 4, the tests would be falsified when the sum of the two numbers is greater than the maximum value, or lower than the minimum value. Sometimes it is falsified with huge negative number, as shown in Listing 12. We are able to discern from where the problem is coming. When running the tests after adding the guards to the substitution rule, all tests pass. When there is a negative and

```
[info] ! RulesCorrectness.wrongTwoDropRightsCorrectness: Falsified after 28
↪  passed tests.
[info] > Labels of failing property:
[info] Expected Vector() but got Vector(0)
[info] > ARG_0: Vector(0)
[info] > ARG_0_ORIGINAL: Vector(-2052695715)
[info] > ARG_1: 2147483647
[info] > ARG_2: 1
[info] > ARG_2_ORIGINAL: 1287019512

[info] > Labels of failing property:
[info] Expected Vector() but got Vector(0)
[info] > ARG_0: Vector(0)
[info] > ARG_0_ORIGINAL: Vector(2147483647)
[info] > ARG_1: 1
[info] > ARG_2: -2147483648
```

Listing 12: Counter example for the wrong rule for two `dropRight`

positive number, if the first `dropRight` applied use the negative number then no element is dropped. The second `dropRight` will remove elements. But the sum of the two is negative, meaning that we are not dropping any element form the list with the substitution.

5.2 Efficiency

We want to know if the substitution is efficient. For a rule, the test will give the best case improvement, the average, the median and the standard deviation. If we were looking at the efficiency of the rule for a `map` and `take` from the Listing 7, we would get the results form Listing 13. We get a small improvement of the original code, understandably as we are not

```
[info] + RulesEfficiency.mapAndTakeEfficiency: OK, passed 200 tests.
[info] > Collected test data:
[info] 100% (Best: 22.264, worst: 1.084, avg: 2.314, median: 1.912, Stddev:
↪  2.201)
```

Listing 13: Efficiency of the rule with a `map` and `take` discarding side effects

mapping over all elements form the list. But, if we were to use the version in Listing 9, we get worse results as shown in Listing 14. When we want to check if a collection `xs` is empty we can

```
[info] + RulesEfficiency.mapAndTakeEfficiency: OK, passed 200 tests.
[info] > Collected test data:
[info] 100% (Best: 6.471, worst: 0.780, avg: 2.109, median: 1.995, Stddev:
↪  0.686)
```

Listing 14: Efficiency of the rule with a `map` and `take` keeping side effects

write `xs.length == 0`. But there is a method on collection that can check if the collection is empty so we can define the rule of Listing 15. As expected we are getting better results when using the substituting code to check if the collection `xs` is empty, as seen in Listing 16

```
def isEmpty(xs: Seq[Int]) =
  Rewrite(
    xs.length == 0,
    xs.isEmpty
  )
```

Listing 15: Rule to use the `isEmpty` method instead of checking the length

```
[info] + RulesEfficiency.isEmptyEfficiency: OK, proved property.
[info] > Collected test data:
[info] 100% (Best: 11.235, worst: 1.317, avg: 1.964, median: 1.697, Stddev:
↪  1.329)
```

Listing 16: Efficiency of rewriting `xs.length == 0` by `xs.isEmpty`

5.3 Side effects

When we tests the different combination of side effects for the two `map` rule of Listing 2, we get the results of Listing 17. As we did expect, having one or the other functions with side

```
[info] + RulesFunctionSideEffects.twoMaps{f1}Impures: OK, passed 200 tests.
[info] + RulesFunctionSideEffects.twoMaps{f2}Impures: OK, passed 200 tests.
[info] ! RulesFunctionSideEffects.twoMaps{f1+f2}Impures: Falsified after 63
↪  passed tests.
[info] Expected List("f1(0)", "f1(0)", "f2(0)", "f2(0)")
        but got List("f1(0)", "f2(0)", "f1(0)", "f2(0)")
[info] > ARG_0: Vector(0, 0)
[info] > ARG_0_ORIGINAL: Vector(-78667205, 0)
[info] > ARG_1: <function1>
[info] > ARG_2: <function1>
```

Listing 17: Tests with side effects in the function for the two `map` rule

effects is not a problem, but we run into a problem if both have side effects. It is important to notice that it was falsified with a sequence with two elements, the order of side effects would not have been changed if the sequence was empty or containing only one element.

When we test the rule with a `map` and `take`, given in the Listing 7, we get a counter example, Listing 18 for the side effects. And when we use the corrected rule of Listing 9, all side effects tests are passing.

Sometimes, it can help uncover some library specificities. if we look at the rule from Listing 19, we would expect to have no problem when only `f` has side effect, because the mapping comes after the `takeWhile`. Yet, when we run the tests it fails when only `f` has side effects, as seen in Listing 20. what is happening is that the call to the method `toSeq` will not go through each elements and delay the call on both the predicate and the function to the next time the collection is traversed. The tests fails when only the predicate has side effects, for the same reasons. We have to force the whole sequence to be build, hence we use the method `toVector` instead.

6 Implementation

As this project aims to be helpful for the user of the rewrite mechanism, it is build so the user can plug the file with the rule to the test generator without changing anything to it. As we are

```
# Version discarding side effects:
[info] ! RulesFunctionSideEffects.mapAndTake{f}Impures: Falsified after 21
↪  passed tests.
[info] > Labels of failing property:
[info] Expected List("f(0)") but got List()
[info] > ARG_0: Vector(0)
[info] > ARG_1: <function1>
[info] > ARG_2: -1
# Version keeping side effects:
[info] + RulesFunctionSideEffects.mapAndTake{f}Impures: OK, passed 200
↪  tests.
```

Listing 18: Results from the side effects tests from the rule of `map` and `take`

```
def takeWhileAndMap(xs: Seq[Int], ptw: (Int => Boolean), f: (Int => Int)) =
  Rewrite(
    xs.takeWhile(ptw).map(f),
    xs.iterator.takeWhile(ptw).map(f).toSeq
  )
```

Listing 19: Rules for `takeWhile` and `map`

```
# using toSeq
[info] ! RulesFunctionSideEffects.takeWhileAndMap{ptw}Impures: Falsified
↪  after 64 passed tests.
[info] > Labels of failing property:
[info] Expected List("ptw(0)", "ptw(0)") but got List("ptw(0)")
[info] > ARG_0: Vector(0, 0)
[info] > ARG_0_ORIGINAL: Vector(1215732092, -1868828241)
[info] > ARG_1: <function1>
[info] > ARG_2: <function1>
[info] ! RulesFunctionSideEffects.takeWhileAndMap{f}Impures: Falsified after
↪  64 passed tests.
[info] > Labels of failing property:
[info] Expected List("f(0)", "f(0)") but got List("f(0)")
[info] > ARG_0: Vector(0, 0)
[info] > ARG_0_ORIGINAL: Vector(-77257244, -492179513)
[info] > ARG_1: <function1>
[info] > ARG_2: <function1>
# using toVector
[info] + RulesFunctionSideEffects.takeWhileAndMap{ptw}Impures: OK, passed
↪  200 tests.
[info] + RulesFunctionSideEffects.takeWhileAndMap{f}Impures: OK, passed 200
↪  tests.
```

Listing 20: Difference for the side effects tests results when using `toSeq` and `toVector`

working with written code, the test generator has to be able to understand which elements are used for the rules and how they are applied.

To be able to understand and use the code in the rewrite rules, the project use the scala.meta[1] toolkit. Then, the project use ScalaCheck[7] to have random generated tests.

6.1 Scala.meta

Scala.meta is a toolkit to do metaprogramming in Scala. It allows to deconstruct code into its syntax tree and work with its elements. Alongside the `StaticAnnotation` and the new language construct `inline` and `meta` described in [3], we can use the same macro annotation from the Dotty-Linker to process the definitions of the rules. The improvement in [3] allows us to write the code of Listing 21 to process the body of a rule object as in Listing 2.

```
import scala.meta
import scala.meta._
import scala.annotation.StaticAnnotation

class rewrites extends StaticAnnotation {
  inline def apply(defn: Any): Any = meta {
    /*···*/
  }
```

Listing 21: Skeleton of the tests generator

Scala.meta comes with great quasiquotes that can construct or deconstruct syntax tree. The main pattern for quasiquotes available in scala.meta are define in their the scala.meta documentation[2]. Two or three dots represent a sequence or a sequence of sequences, after the type of the variable has to match the type expected by the quasiquote.

For this project, the main quasiquote is the one that will deconstruct the object with all the definition it contains. In the Listing 22, we use the quasiquote to deconstruct the object into a value `name` repesenting the name of the object and value `stats` that represent the sequence of all definition within the body of the object. Then we go through every definition to retrieve

```
  val q"object $name { ..$stats }" = defn
```

Listing 22: Quasiquote to deconstruct the object containing the rules

all the rules. There is a quasiquote to deconstruct a method definition. As we know that the body will contain the object `Rewrite`, we can further specify the deconstructing quasiquote. The Listing 23 show how we can match the method definition with a substitution rule.

```
    q"""..$mods def $name[..$tparams](...$paramss): $tpe =
      Rewrite(${left: Term}, ${right: Term})"""
```

Listing 23: Quasiquote to match on a rule definition

There is a great tutorial[6] available to learn scala meta and play with it.

6.2 ScalaCheck

ScalaCheck is a library that allows property based testing, it is the Scala version of Haskell QuickCheck[4]. The principal aim is to test with random input user defined properties. It has different tools to build the property. For instance, if we want to test the rule from the Listing 2 we would have to write what is represented in the Listing 24.

```
property("twoMapsCorrectness") = forAll {
  (xs: Seq[Int], f1: (Int => Int), f2: (Int => Int)) =>
    xs.map(f1).map(f2) == xs.map(x => f2(f1))
}
```

Listing 24: Property defined to check the results of the substitution of two maps by one

What ScalaCheck does when it test a property like that, is to use random generators for all parameters, in the example it generates random elements for `xs`, `f1` and `f2`, and evaluate the boolean until either it returns false are a maximum number of tries. When the boolean returns false, then the test fails and the property is invalidated by this counter example. Obviously, it could fail to produce the counter example, but if the number of test is large enough than it is safe to assume that the property is correct.

ScalaCheck provides generators for the common type, we do not have to define generators for all the elements used. But ScalaCheck offers the possibility the define other generators, especially if it is a custom class then we have to provide ScalaCheck with a random generators. There is two parts, first a generator of type `Gen[T]`, and another element relying on the generator of type `Arbitrary[T]`. Let use have `case class A(val b: Int, val c: Int)`, then to use this class in a property, we would have to define the generator as follows

```
val genA : Gen[A] = for {
  b <- arbitrary[Int]
  c <- arbitrary[Int]
} yield A(b, c)
```

Then we have to define the `Arbitrary[A]` that will provide the property test with random elements from the generator. We can use the `Arbitrary` object to get that element and then we have

```
implicit lazy val arbA: Arbitrary[A] = Arbitrary(genA)
```

If the rule use customs classes, then the rule object will have to provide the generators and the arbitrary element, to work with this project. We will also have to create the generators for every impure function that we will define for the tests.

ScalaCheck also provide a class, `Properties(String)`, that can encompass a set of properties. It comes with a method that will check all properties within its body. With `sbt`, every object that is a subclass of `Properties` will see every property check if it is define in the `test` directory.

6.3 Tests

As discussed in Section 2, there are three mains properties to test, the correctness of the substitution, its average speed and if side effects matter.

Correctness

For the correctness, it is straightforward, we have the definition of the left side and ride sight in the value `left` and `right`. The parameters used in both of those expression are inside the value `params`, we just have to check that their evaluation yield the same results. Thus the test generated is given in the Listing 25. The operator `=?` from ScalaCheck will give the *expected* result and the *observed* result when the test fails.

Efficiency

When we evaluate the efficiency of the substitution rule, we want to try over many different inputs and measure the time of the computation. When know that in the rules definitions

```
q"""
  property(${name.toString + "Correctness"}) = forAll {
    (..$params) => {
      $left =? $right
    }
}
"""
```

Listing 25: Correctness test for a substitution rule

we have parameters that plug holes, we have to bind the parameters with the sample values. The list `params` of parameters give us the name and corresponding type that we have to bind. To retrieve random sample, we use ScalaCheck. Given a parameter `name: Tpe`, we need to write `val name = arbitrary[Tpe].sample.get`. The Listing 26 create all the declarations from the parameters. Once we have bound each arguments with a random value, we can

```
params.map {
  case Term.Param(_, name, tpe, _) => {
    // Casting into a Var.Term
    val valName = Pat.Var.Term(Term.Name(name.toString))
    // Casting the Type.Arg into a Type
    val valTpe = decltpe.get.toString.parse[Type].get
    q"val $valName = arbitrary[$valTpe].sample.get"
  }
}
```

Listing 26: Mapping the list of parameters into a list of sample

now use the body of the left and right sides of the rule to measure the time. We compute the quotient between the time needed for the *left side* and *right side*. We gather the time for different samples. A statistic is returned with the method `collect` from ScalaCheck. The test is given in the Listing 27.

Keeping side effects

The last point is more challenging because we have to produce the side effects. It is also split in two parts, it easy to catch when there is a function, but more difficult to deal with all method applied to user defined classes. In that regards, all combination of function impurity are generated and the corresponding test is created, but for more specialised impureness the user will have to provide the generator and means to check that impurity.

Function with side effects There is four main objectives in this part. First, we have to be able to tell which parameter is a function. Second, we have to create all possibilities whether the function remains pure or becomes impure. Third, we have to substitute the type of this parameter with a type that has side effects. Fourth, we have to produce the test and make sure that it is not interfered with other tests or executions.

Finding the functions What is great with scala.meta is that there is a specific type, `Type.Function` associated with a function. At the stage where we are working, the compiler replace the syntactic sugar of a `(Int => Int)` by a `Function1[Int, Int]`. Although, scala.meta deconstructs the former in a `Type.Function`, the latter is deconstructed in a `Type.Apply`. Therefore, to be able to catch a parameter that is a function, we have to match its

```
val efficiency = q"""
  property(${name.toString + "Efficiency"}) = {
    val nTries = 100
    val improvements = (0 to nTries).map(i => {
      ..${generateSpeedTest(params, left, right)}
    })
    val average = improvements.sum / improvements.length
    val sorted = improvements.sorted
    val median = sorted(nTries/2)
    val sd = Math.sqrt(
      (sorted.map(x => (x - average)*(x - average)).sum)/(1 +
↪   nTries).toDouble
    )
    val worst = sorted.head
    val best = sorted.last
    collect(
      "Best: " + "%.3f".format(best),
      " worst: " +"%.3f".format(worst),
      " avg: " + "%.3f".format(average),
      " median: " + "%.3f".format(median),
      " Stddev: " + "%.3f".format(sd))(true)
  }
"""
```

Listing 27: Efficiency test for a substitution rule

declared type with either a `Type.Function` or a `Type.Apply`. In the first possibility, no further work has to been done, in the second we have to be more careful as `Type.Apply` represents any type with type parameters. A function has a type name like `_root_.scala.FunctionN`, and we just have it to match with a corresponding regex. The first N type parameters are the type of the arguments of the function, the last type parameter is the type returned by the function, and we can easily rebuild a `Type.Function` from that.

Function with side effects combinations Now that we can spot when we have a function, we have to produce all combination of pure function and impure function. We always have two choices, either the function becomes impure or stay pure. That means if out of all the parameters we have n functions, we will have 2^n combination. We can discard the one were all functions remains pure so that leaves $2^n - 1$ possibilities. A naive way to build them is to construct the binary tree that results from the choice for each parameters, and use the leafs as the representation of all choices made.

Representing side effects Before tackling the creation of the subtype, we have to decide on how to represent the side effects. They are represented by appending a `String` to a `ListBuffer[String]`, all the impure functions within a test will write to the same buffer and the order of writing are compared between each representation. The new type has to be a subtype of the function type, moreover we have to redefined the `apply` method for this subtype. So this leaves three informations to represent in this subtype, the actual function used, the buffer where the side effects will be represented and what the impure function should write inside that buffer. In essence, if we are to build an impure function for `(Int => Int)`, we would like to have something similar to the code given in the Listing 28. If the buffer is a value of the class then when we have to provide the buffer to the generator. Additionally, we are sharing the buffer between the different arguments with side effects. It seems logical to delay the affectation of the buffer when we know all the arguments with side effects. For

```scala
case class ImpureFunction(
  val f: (Int => Int),
  val buffer: ListBuffer[String],
  val effect: String
  ) extends (Int => Int) {
  override def apply(i: Int): Int = {
    buffer.append(effect)
    f(i)
  }
}
```

Listing 28: First attempt for a sublcass of a function with side effects

the representation of the side effects, we could choose a random string as representation and provide this random value to the generator. We might run into collisions, and it is not really explicit when looking at a falsified test. Choosing the name of the argument as the representation of the side effects makes more sense. We can add the list of actual argument to the string representation, so it can differentiate the calls. As we can represent many different argument with the same subtype, the affectation of the effect is delayed. So the actual class that is being generated is illustrated in the Listing 29.

```scala
case class ImpureFunction(
  val f: (Int => Int),
  var buffer: ListBuffer[String] = ListBuffer.empty[String],
  var effect: String = ""
  ) extends (Int => Int) {
  override def apply(i: Int): Int = {
    buffer.append(effect)
    f(i)
  }
}
```

Listing 29: Example of the generated subtype of a function

Subtyping a function The difficult part now is to be able to subtype, accordingly, any kind of `Type.Function`. We have to give it a fresh type name, which is doable with the method `Type.fresh`, then we have to build the quasiquote that will create the syntax tree representing this new class. For that, we need to build the constructor of the class, the constructor call of the parent class and the body. First, let us extract all informations needed from the function type

```scala
// given Type.Function
val tpe : Type.Function = /*...*/
// Ctor.Call derived from the Type.Function
val funCtorCall = ("(" + tpe.toString + ")").parse[Ctor.Call].get
// Derived parameters from the Type.Function
// and the list of parameters for an actual call
val (applyParams, args) = (((tpe.params) zip (1 to tpe.params.length)) map {
  case (t, i) => {
    val name = Term.Name("x" + i)
    val retArg : Term.Arg = arg"$name"
    (param"$name : $t", retArg)
  }}).unzip
```

As we can see, we do not really have to know the arity of the function neither the types of that function. We have to build every piece related to the new class, that means giving it a new name, defining the constructor, we have the actual function Then, as we are going to use them with ScalaCheck, we have to create the generators. First, let us define the generator as

```
val genTArgs = List(tpe)
val newTpeArgs = List(tpeName)
val genTpe = t"Gen[..$newTpeArgs]"
val enumFun = enumerator"fun <- $genFun"
val createInstance = ctor"${Ctor.Ref.Name(tpeName.toString)}(fun)"
val genName = Pat.fresh(tpeName.toString + "Gen")
val valGen = q"""
val $genName : $genTpe =
for {..${List(enumFun)}}
  yield $createInstance
"""
```

With that, we can write the last part for the `Arbitrary[T]` with the code

```
val arbName = Pat.fresh(tpeName.toString + "Arb")
val arbTpe = t"Arbitrary[..$newTpeArgs]"
val argGen = arg"${genName.name}"
val createArb = ctor"Arbitrary($argGen)"
val lazyArb = q"implicit lazy val $arbName : $arbTpe = $createArb"
```

The last thing that we have to be aware of is if we already have subtyped a function type. It is likely that within a set of rules the same function type appears more than once. In order to prevent extra subtyping, we want to subtype a function the time we observe it. Afterward, we want to serve the created associated type when observing the same type of function. The project has a `Map[String, Type.Name]` that map the string representation of a `Type.Function` with the associated created subtype name. So we can serve the subtype name when finding the same `Type.Function`.

Function side effects test Now that we have our new subtypes, we have to create the test. As it has been defined, we have variable members for each of those subtypes, they will have to be set for each tests. That means, when we replace the function type by its new subtype, we would have to associate the parameters with the two values, but as the combination is being build from the first parameter to the last it cannot be done at this time. The only moment where we can bind the buffer to the concrete elements of the classes is when we know each parameters that will have to share the buffer. As a work around, the affectation of those values to the concrete elements is predefined in a function that takes a `Term.Name` as argument. Combining everything, we get the code of Listing 30. This test is represented as a function as we have not created the buffer yet, it will be done once every parameters have been accounted for. The buffer is cleared at the start of the test to remove anything that has been left from a previous test, after that all mark left by the functions during the evaluation of the left side and right side are compared. If they coincide then we can assume that side effects would not matter. Now if the side effects differs from each call, depending on what they are called with, this template will not catch the differences. There is certainly many scenarii where the order of side effects change the future side effects, but it is difficult to come up with a way to automatically generate them.

User defined side effects For that matter, there is also the possibility to handle more precise scenarii that would yield side effects. The test is the given in the Listing 31 The test has

```
// Name of the parameters with impurity
val names: List[String] = /*···*/
// Parameters with the binding definition if needed
val defs: Seq[Term.name => (Term.Param, Seq[Stat])] = /*···*/
// creation of the buffer
val bufferName = Pat.fresh("buffer")
val bufferDecl = q"var $bufferName = ListBuffer.empty[String]"
// Getting the list of parameters for the test and the bindings
val (args, seqDefs) = defs.map(_(bufferName.name)).unzip
val allDefs = seqDefs.flatten
val impureTest = q"""
  property(${name.toString + "{" + names.mkString("+")  + "}Impures"}) =
    forAll {
      (..$args) => {
        { ..$allDefs }
        ${bufferName.name}.clear
        val left = $left
        val leftInfos = ${bufferName.name}.toList
        ${bufferName.name}.clear
        val right = $right
        val rightInfos = ${bufferName.name}.toList
        leftInfos =? rightInfos
      }
  }
"""
```

Listing 30: Randomised test for a combination of function with side effects

```
q"""
property(${name.toString + "GeneralSideEffects"}) =
  forAll {
    (..$params) =>
      this.cleanUp()
      val left = $left
      this.commitLeft()
      val right = $right
      this.commitRight()
      this.checkEffects()
    }
"""
```

Listing 31: User defined side effects test

to implement the trait given in the Listing 10 that will allow the user to handle the way the side effects are represented and how they are to be checked. As we can see, it uses the parameters from the rule, therefore the rule has to be defined with the side effects elements, and as a consequence it has to provide the corresponding generators. Another thing to account for when designing self made side effects tests is concurrency issues. When ScalaCheck tests all properties in parallel, so they cannot share any element to represent side effects as they would affect each other properties. To avoid that problem, if any element is shared, is to check each property one after the other, which can be done with some tweaks and ScalaTest.

7 Future work

There is some improvement that can be done to this project, the efficiency tests are a little bit sketchy because it will depends on the set of parameters that will be generated for the tests. If they are uniformly distributed then we will have a good idea on the performance behaviour of the substitution rule, but if they are not well distributed then we might hit more often than not a best case scenario or a worst case scenario which would skew the results. In my perception, this test is there mostly to give an idea on whether it is an efficient replacement or not. And we can improve further by making sure that there is no preempt from another test during the evaluation of each sides of the rules. If there is a preemption in between to time evaluation, then we are also counting the time that the test we sleeping, which skew a little bit the results.

Another improvement would be to define the different generator for all the user defined classes. As of now, when the rules use a type that has not a corresponding generator in ScalaCheck, the generator has to be visible inside the object containing the rules. Such feat is possible if we are able to infer any constructor, and its arguments, for a user define class. We would have to use ScalaCheck to provide samples for all of the argument and then define the generator, similarly to the function with side effects.

Speaking of side effects, the way they are represented give a good hindsight if the rules messes with them. If the arguments of the function have a clean string representation then it will work well. When any argument has not a defined string representation and use the inherited default representation, we can run into some trouble. Let us examine the rules of Listing 32, We have an identity rule that takes the same argument, but their string repre-

```scala
@rewrites
object BadExamples {
  class MyInt(val i: Int) {
    def toLong: Long = i.toLong
  }

  val genMyInt = for (a <-arbitrary[Int]) yield new MyInt(a)
  implicit lazy val arbMyInt = Arbitrary(genMyInt)
  implicit lazy val cogenMyInt : Cogen[MyInt] = Cogen(_.toLong)

  def missingToString(f: (MyInt => Int)) =
    Rewrite(
    f(new MyInt(5)),
    f(new MyInt(5))
  )
}
```

Listing 32: Example of class without good string representation

sentation will differ because they are not the same instance of the class, as we can see in the Listing 33. There is a lot of room on how to represent the side effects. We could also argue on the depth where we allow side effects, we stay at the arguments of the rule for simplicity.

8 Conclusion

Rewrites is an annotation to define substitution rules. We have created a testing framework that discovers those annotation and tests the correctness, efficiency, and whether the rules keep side effects.

```
[info] ! BadExamplesFunctionSideEffects.missingToString{f}Impures: Falsified
↪  after 0 passed tests.
[info] > Labels of failing property:
[info] Expected List("f(BadExamples$MyInt@1155c216)")
        but got List("f(BadExamples$MyInt@723ca5d9)")
[info] > ARG_0: <function1>
```

Listing 33: Same value but not the same instance

The principle of code rewriting with user defined optimisation rules is really interesting. Having this tool to automatically generate a set of tests is a great help in producing better rules. The idea of being able to write a simple well structured code without caring too much about efficiency is something likeable. But it has its limitation to its usefulness, we have to define the expected code and provide a better alternative. If the code is badly written, in term of complexity, then it will be difficult to come up with a rule for every bad written code. It is better suited for the common usage of a library.

Bibliography

[1] Eugene Burmako. *scala.meta*. 2016. URL: `http://scalameta.org`.

[2] Eugene Burmako et al. *scala.meta quasiquotes documentation*. 2016. URL: `https://github.com/scalameta/scalameta/blob/master/notes/quasiquotes.md`.

[3] Eugene Burmako et al. *SIP-28 and SIP-29 - Inline meta*. 2016. URL: `http://docs.scala-lang.org/sips/pending/inline-meta.html`.

[4] Koen Claessen. *QuickCheck: Automatic testing of Haskell programs*. 2016. URL: `https://hackage.haskell.org/package/QuickCheck`.

[5] *Dotty A next generation compiler for Scala*. URL: `http://dotty.epfl.ch`.

[6] Òlafur Pàll Geirsson. *A Whirlwind Tour of scala.meta*. 2016. URL: `http://scalameta.org/tutorial/`.

[7] Rickard Nilsson. *ScalaCheck: Property-based testing for Scala*. 2015. URL: `https://www.scalacheck.org`.

[8] Martin Odersky. *The Scala Language Specification v 2.9*. 2014.

[9] Dmitry Petrashko. *Dotty Linker*. 2016. URL: `https://github.com/dotty-linker/dotty`.

[10] Dmitry Petrashko et al. *Call Graphs for Languages with Parametric Polymorphism*. 2016. URL: `https://infoscience.epfl.ch/record/217276`.

[11] *ScalaCheck User Guide*. 2016. URL: `https://github.com/rickynils/scalacheck/blob/master/doc/UserGuide.md`.

[12] Denys Shabalin. *Quasiquotes*. URL: `http://docs.scala-lang.org/overviews/quasiquotes/intro.html`.

[13] *The interactive build tool*. 2016. URL: `http://www.scala-sbt.org`.