Randomised testings of optimising code substitution

Jérôme Amiguet

École Polytechnique Fédérale de Lausanne

February 15, 2017

Overview

Rewrites mechanism

One new technology being developed for Dotty is a Rewrites mechanism that allows the definition of optimising code substitution rules. Granting the library authors the ability to provide code optimisations to the library users.

Defining a substitution rule

The following code defines a simple substitution rule

```
@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
    )
}
```

Rule applied

```scala
/*...*/
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)
/*...*/
```

Rules criteria

The rules have to hold the following properties

- Correctness
- Efficiency
- Keeping Side Effects

Randomised test framework (1)

This project aims to provide a generation of randomised tests for the defined substitution rules. To achieve that it uses scala.meta and ScalaCheck.

Randomised test framework (2)

Ideally, the user should not change anything to the file containing the rules when he wants to test them. By using the same annotation, the framework add different inner classes to the object containing the rules to verify the properties.

Randomised test framework (3)

The rules are applied to different usage of the library that provides them. Generating random inputs for the different tests allow to reveal unexpected counter examples.

Correctness and efficiency

Those two properties are straightforward, as we have to check that the evaluation of both side are equals, and that the evaluation of the right side *(to)* takes less time than the left side *(from)*.

Side Effects

The substitution rule should keep all side effects and keep the same order for the side effects.
The tests look at the side effects from the functions, and assume that they are on the same medium.

Side Effects — Test restrictions

Obviously, there could be other arguments to the rules that could produce side effects, but the goal was to keep the framework simple. Side effects can be expected from a function.
The worst case for the side effects is that they interfere with each other by using the same resource.

Side Effects — Representation

The side effects from a function call is represented by the name of the function and its arguments as a string.
All the representation are stored inside a ListBuffer.
We can check if the order is the same, and if the arguments are the same.

Side Effects — User defined representation

As we are limiting the side effects to the functions, we offer another alternative to test side effects with another set of tests. This set requires the user to provide the side effects representation and a method to verify between the two evaluations.

Using the framework (1)

When defining simple and basics rules, the framework:

- ✓ Makes it easy to test the properties
- ✓ Provides counter examples for corner cases
- ✓ Helps to improve the rules

Using the framework (2)

When defining rules using user defined types, we have to provide the object containing the rules with the **associated generators**. It Could be automated, if we can **infer** the constructors of the different types.
There is no restriction to the generated elements, if the rules are applied for a **domain specific set** then, *presumably*, the user has to mask the generators of ScalaCheck with their own specific generators.

Questions ?