Proceedings of the 10th International Conference on Applied Informatics
Eger, Hungary, January 30–February 1, 2017. pp. 255–261
doi: 10.14794/ICAI.10.2017.255

When Desugaring Makes Your Code Sour
Reducing the Number of False Positives and Negatives with Quasi-quotes

Artúr Poór

Eötvös Loránd University, Budapest
`poor_a@inf.elte.hu`

Abstract

Programming language designers tend to continuously seek ways to make their languages convenient to use. Such a way is to introduce syntactic sugars, which are alternative but more terse notations for existing language constructs, and defines formal rewriting rules for those syntactic sugars which desugar code by reducing into equivalent code snippets of the base language. In case of Scala, this may come at the cost of inaccurate compiler checks, particularly check for defined but unused variables. When used together, the combination of independent rewrite rules may introduce false negative compiler warnings. Conversely, codes produced by other rewriting rules would cause so many false positive warnings that the compiler omit checks, making those language constructs prone to programmer errors.

In this paper we introduce an approach to tackle the issue of Scala [3] programs. In a nutshell the aim is to perform more accurate analysis using the original code with syntactic sugars as basis.

Keywords: static analysis, syntactic sugars, software correctness, compiler warnings, Scala

MSC: 68

1. Introduction

Some programming errors are harder to find than others from the perspective of a compiler. Some errors are simply impossible to catch during compilation, as in the case of a dereference of a null pointer, or require more sophisticated type systems, for instance to detect out-of-bound indexing. Still, we expect the compiler to catch as many errors as possible to reduce the time needed for testing and the number of test cases.

As we will see in this paper, in some cases the compilation process hinders error detection. We will focus on detecting defined but unused variables, which are reliable indicators of programming errors.

Many programming language features one or more syntactic sugars. They are included because they make expressing specific ideas more concise and make for a reduced number of programming errors. Syntactic sugars are usually not part of the core language per se. Instead, they are higher level constructs defined by rewriting rules, which are textual rewritings of syntactic sugars to code fragments involving more fundamental language constructs. These rewritings are performed by the compiler in early phases of compilation.

In Scala, a hybrid object-oriented and functional programming language influenced by Java, there are several examples of syntactic sugars. Such an example is the *for comprehension*. Suppose one needs a handful of powers of two in a list. One might write the following in Scala to accomplish the task:

```
for (x <- List(1, 2, 3, 4))
yield intPow(2, x)
```

The evaluation is as follows. The `x <- List(1, 2, 3, 4)` part is called *generator*. The variable `x` iterates over the list of numbers, taking each number as value. For each number in the list, the output expression `intPow(2, x)` is evaluated and is put in the result list. One might easily guess that the result is `List(2, 4, 8, 16)`. The function `intPow` is defined so that the result list is populated with integers instead of floating-point numbers.

As in functional languages, not only variables can be written on the left side of the arrow but arbitrary patterns can occur. (After all, in most functional languages a variable is one type of patterns.) This makes iterating over complex data structures easy. Let us represent points on a plane as pairs of integers. One might define point reflection of some points in the following way. For the sake of this argument, the second component of the result is deliberately incorrect.

```
for ((x, y) <- List((1, 2), (-2, 4));
     (a, b) = (1, 1))
yield (a + a - x, b + b)
```

Using the pattern `(x, y)`, one could bind variables to the components of a point while iterating over a list of points, so that the components could be accessed directly. The point `(a, b)` is a constant and does not change during evaluation. If one tries and compiles this code (of course it needs to be enclosed in a class first) with the Scala compiler version 2.12.1 and earlier, with the `-Ywarn-unused` argument supplied, then she gets the following output:

```
Point.scala:4: warning: local val in value $anonfun
is never used
    (a, b) = (1, 1)
     ^
Point.scala:4: warning: local val in value $anonfun
```

```
is never used
    (a, b) = (1, 1)
        ^
two warnings found
```

It is, without doubt, very surprising. How could the compiler deduce that `a` and `b` are unused? And, perhaps more importantly, how could the compiler not notice that `y` is actually not used? This embodies both false positive and negative warnings.

The situation slightly improved starting with Scala compiler version 2.12.2 in a sense that the programmer does not receive any warnings at all. In fact, should she use none of the variables `x`, `y`, `a` and `b`, she will not get warnings from the compiler. This embodies false negative warnings only.

Is there a bug in the compiler? Unfortunately, there is nothing wrong with it. The underlying cause is more complicated than that, as we will see in this paper.

There is one more example of false negative warnings in Scala 2.12.1 and earlier. The observable phenomenon is the same for anonymous, or lambda, functions. The underlying cause is completely different, though. For instance, the programmer does not see any warnings from the compiler when she defines the point reflection across the x axis as the function below.

```
def reflect = { case (x, y) => (-x, x) }
```

This is due to the lack of implementation of checking variables in function arguments.

In this paper we make the following contributions:

- We identify the root cause of false negative warnings for defined but unused variables (Section 2), and show what makes it hard to solve.
- We propose a solution to these shortcomings. We build upon quasi-quotes, a powerful mechanism for constructing and inspecting abstract syntax trees, already included in the Scala compiler (Section 3). We also employ standard static analysis [2], in order to detect truly unused variables in Scala programs.

2. Background

In the first example, where for comprehensions were used for point reflection, two distinct syntactic sugar rewritings play role. One is the for comprehension itself, the other is pattern definition. We will explain them in turn in this section.

As it was already mentioned, for comprehensions in Scala are not part of the core language. Instead, they are defined through rewriting rules. In the most simple case, a for comprehension with one variable transforms elements of a list into a new list in such a way that the result has same length as the original and the same code is applied to each element. Thus, the comprehension

```
for (x <- List(1, 2, 3, 4))
yield intPow(2, x)
```

could be written using `map` function of the `List` class. `map` is a higher-order function which takes a function as argument and produces a new list by applying the function on each element. So the aforementioned for comprehension could be written as

```
List(1, 2, 3, 4).map{ x => intPow(2, x) }
```

Indeed, this is how the compiler rewrites for comprehensions with only one variable during desugaring, as part of the compilation. This also holds for comprehensions with only one generator, as we will see shortly.

Before we introduce constant variables in for comprehensions, let us now turn our attention to pattern definitions, which also play crucial role. In Scala, a pattern definition such as `val (a, b) = (1, 2)` is used to simultaneously define variables `a` and `b`. This is rewritten during compile time to the following:

```
val t$1 = (1, 2) match {
  case (a, b) => (a, b)
}
val a = t$1._1
val b = t$1._2
```

That is, a pattern definition is expanded to a pattern matching expression and a sequence of value definitions. The new variable `t$1` is introduced by the compiler so that it does not clash with any other identifiers in scope. It holds values for the newly introduced variables. The attributes `_n` are projections of a tuple.

The matter becomes even more interesting when pattern definitions are used together with for comprehensions. Generally speaking, when a generator is followed by a pattern definition, they are merged into one generator. Let us consider the point reflection example from page 2 again.

```
for ((x, y) <- List((1, 2), (-2, 4));
     (a, b) = (1, 1))
yield (a + a - x, b + b)
```

The desugaring of this expression consists of three steps. First, the generator and the pattern definition `(a, b)` are merged together into one generator. Then the pattern definition is expanded, and the for comprehension is rewritten using `map`. After the first step, the code is similar to the following:

```
for (((x, y), (a, b)) <-
       for ( p1 @ (x, y) <- List((1, 2), (-2, 4)) )
       yield { val p2 @ (a, b) = (1, 2); (p1, p2) })
yield (a + a - x, b + b)
```

The inner for comprehension pairs up each elements from the list with the constant `(1, 1)` pair, creating a list of pairs where both components of the elements are pairs themselves. Here, the pattern `p1 @ (x, y)` causes that the pair `(x, y)` is also named as p1 for readability. An attentive reader might guess that a and b in the inner for comprehension are unused. Nonetheless, the compiler strictly obeys the rewriting rules and defines them, regardless of context.

After the last step, the completely desugared code is similar to the following (we have removed the annotations from the code so to clarify the point and make the code easier to read):

```
List((1, 2), (-2, 4))
  .map { p1 =>
           p1 match {
             case (x, y) => {
               val t$1 = (1, 1) match {
                 case (a, b) => (a, b)
               }
               val p2 = t$1
               val a = t$1._1
               val b = t$1._2
               (p1, p2)
           }
       }
  }
  .map { case ((x, y), (a, b)) =>
           (a + a - x, b + b)
  }
```

Now it is very clear that the compiler introduced two unused variables, namely a and b. As it is already mentioned, using Scala 2.12.1 or earlier this results in unjustified warnings from the programmer's point of view since the programmer actually used these variables. On the other hand, the unused y does not cause any warnings, because the compiler does not check variables bound in patterns. Starting with Scala 2.12.2, this issue has been partially solved. The compiler does check variables bound patterns and does not check variables in for comprehensions at all. As a result, there are no more false positive warnings but only false negatives in for comprehensions.

Could we be more precise than the compiler? Perhaps if we have access to the original, programmer-written version of the source code, then we do not have to take these synthesised variables into account. Unfortunately, the desugaring takes place during parsing and the original version is therefore lost.

3. Quasi-quotes to the Rescue

A powerful metaprogramming tool exists in several programming languages, such as Haskell [5], MetaOCaml [1] and Scala [6, 4], called *quasi-quoting*. In Haskell, a compiler extension named Template Haskell [5] allows to construct abstract syntax trees from code fragments. That is, the user does not need to construct trees manually, instead she encloses code fragments between special brackets. As an example, `[| 1 + 2 |]` represents the abstract syntax tree of the expression `1 + 2` in Haskell. The advantage is that the familiar concrete, or surface, syntax may be used to create syntax trees.

The mechanism is the same in Scala. The difference is that programmer specify code fragment in an ordinary string literal. For example, `q"1 + 2"`, when evaluated, represents the abstract syntax tree of `1 + 2`. Here, the leading `q` is called quasi-quoter in terminology. Note that quoted code is not type checked, meaning that `q"true + 1"` results a syntax tree despite the fact one cannot add `true` to `1`.

`q` is not the only quasi-quoter in Scala. There is `tq` for syntax trees of types, `cq` for case branches, `pq` for patterns, and `fq` to quote for comprehensions. Throughout this paper, we restrict ourselves to `q` and `fq` disregard the others.

It is important to note that the template to be quoted must be fully known at compile time. Therefore it must be a string literal. The reason is that quasi-quoters are implemented as macros and evaluated during compilation to avoid runtime overhead. The template may contain unspecified parts, or *holes.* Holes are filled at the syntax tree level when the tree is constructed. As example, in `q"o.m($arg)"` the variable `arg` must be in scope and must hold a syntax tree. The `$arg` means that value of `arg` is *spliced in* when the tree is constructed.

Contrary to Template Haskell's brackets, which may only be used for tree construction and ordinary pattern matching is necessary to inspect the code representation, quasi-quote of Scala is also useful for pattern matching on syntax trees. That is, the following code always prints `invocation of contains`:

```
val tree : Tree = q"Array(1, 2, 3).contains(2)"
tree match {
  case q"$o.$m($arg)" => println("invocation of " + m)
  case _              => println("something else")
}
```

This feature makes quasi-quotes all the more powerful metaprogramming tool. Combined with the ability of Scala reflection API to obtain abstract syntax trees of Scala programs, one could build static analysis tools more easily because the Scala reflection API provides us with a parser and type checker.

Indeed, we could employ quasi-quotes for static analysis, and searching for unused variables to be more specific. This is because using pattern matching we could inspect syntax tree of a for comprehension:

```
val q"for (..$enums) yield $output" =
  q"for (x <- List(1, 2, 3)) yield intPow(2, x)"
```

Here, `enums` is a list of syntax trees, as the leading `..$` indicates. It holds syntax trees of generators, guards and value definitions in the comprehension. The rest of the program is free to inspect them in turn using the `fq` quasi-quoter. The variable `output` hold the syntax tree of the output expression, that of `intPow(2, x)`.

The message of this section is this. In order to accurately identify unused variables, it is better to analyse code which did not go through desugaring rather than on code that did. As we saw earlier, combination of distinct rewrite rules has unforseen consequences. As it was demonstrated in this section, quasi-quoting allows us to inspect the code as if desugaring did not happen. It also allows us to forget about auxiliary variables introduced by the compiler.

4. Conclusion

Syntactic sugars provides us a way to express our ideas more concisely and with less errors. However, there are cases when they make checks such as reporting unused variables very difficult, if not impossible, in certain circumstances. Our approach is noble in the sense that it works, independent of rewrite rules. It may also be applied successfully in the future, when perhaps a new rewrite rule poses another such difficulty. Then it may take a few compiler releases to fix the issue, whereas quasi-quoting can be readily used.

References

[1] K. Czarnecki, J. T. O'Donnell, J. Striegnitz, and W. Taha. *DSL Implementation in MetaOCaml, Template Haskell, and C++*, pages 51–72. Springer Berlin Heidelberg, 2004.

[2] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 2nd edition, 2005.

[3] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala – A comprehensive step-by-step guide*. Artima Press, 3rd edition, April 2016.

[4] D. Shabalin. Quasiquotes Introduction. `http://docs.scala-lang.org/overviews/quasiquotes/intro.html`. Accessed: 1 June 2017.

[5] T. Sheard and S. P. Jones. Template Meta-programming for Haskell. *SIGPLAN Notices*, 37(12):60–75, December 2002.

[6] D. Wampler and A. Payne. *Programming Scala – Scalability = Functional Programming + Objects*. O'Reilly Media, 2nd edition, December 2014.