Style Checking With Scala.Meta

Mathieu Demarne
mathieu.demarne@epfl.ch

June 2015

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

1 Abstract

In many companies, code reviews are big parts of the day-to-day life of an engineer. Some can argue that the time spent on someone else's code is time that some cannot spent on her or his own work. Code reviews usually allow to find two kinds of issues. The first kind consists of high-level design errors compared to the specifications of the program while the second type consists of style problems.
In this report, we argue that the search of the later kind of error can be automated, as the set of style issues is usually well defined for a programming language. We propose modifications to a linter tool called Obey, which allows to analyze Abstract Syntax Trees of Scala code, raise warnings and suggest changes that can be automatically persisted.
Over the course of this project we worked jointly with Codacy, a company proposing automated code reviews, to have an industrial application which would allow users to define their own set of rules. This report also presents the results of this real life testing.

Keywords: Style Checking, Automated Code Review, Code Inference, Metaprogramming

2 Introduction

Reviewing code is a required step for every engineer in the industry. It can even be a legal requirement to deploy software. Unfortunately, reviewing code is hard. The reviewers first have to focus on the programmer's task, need to have in mind all the specifications of the pieces of code to review, immerse themselves into the programming style of the other, before being able to start reviewing. Moreover, reviewing is rarely quantified and corresponds to time that the reviewers could instead spend on their own work. As an outcome, reviewing code is sometimes neglected, done too fast or without the full attention that it should receive, leading to undiscovered bugs which at best will be found in beta testing and force to restart the deployment procedure.
In the past few years, the platforms allowing better control over a project development as well as regression testing have grown in number and importance. For instance, Phabricator [1] allows a better grip on merge requests, Jira/Crucible [2] makes reviewing code an integral part of the development process rather than a work done when time permits, and finally many platforms such as Travis, Shippable, Gitlab CI or even Jenkins offer regression testing based on revision control [3].

The solution Codacy [4] provides is at the cross of those two areas. By fetching the latest modifications done in a repository, the Codacy platform analyzes code statically, produces warnings and gives a final grade to the code style. It is also flexible, allowing the programmer or project owner to select from a large set of warning definitions. Most recently, a prototype developed by the company also proposes to write custom rules. Codacy currently supports a variety of languages such as Scala, JavaScript, PHP, Python and CSS.
As a part of our effort with Obey, our linter tool developed on top of Scala.Meta [5], we have worked jointly with Codacy to power code reviews of Scala projects.

We have also pushed the boundaries of the Obey project to propose transformation rules that persist changes while keeping layout information and comments, a feature that might interest ingineers in the future, both for style rules as well as to automate various migration processes.
Defining migration rules would allow to seamlessly move the code base of a project from an old version of a framework or library to a new one. For instance, the Play Framework [6] allows to develop high velocity web applications using Scala and Java and is under constant development. As an outcome, versions are not always backward compatible, and a few changes need to be done for each migration. This could be automated using Obey.

This report presents Obey in section 3, how we deal with persisting AST modifications while preserving the original source code format in section 4 and the work we have done jointly with Codacy in section 5. It also shows and comments use cases of the Obey prototype in section 6.

3 Obey: A Linter Tool

Repository: github.com/mdemarne/Obey

Lint [7] is a program developed at Bell Labs in the eighties to identify suspicious and non-portable constructs in the C programming language. The term *lint-like* became popular to designate programs running static analysis on source code and generating warnings.

3.1 Origins And Motivation

Generating warnings inside compilers allow little customization and makes the modification of the applied rules a complicated task for programmers unfamiliar with the internal steps of the compilation process.
Due to the rich syntax of Scala, adding custom warnings is becoming increasingly relevant. A few tools have already been studied and developed in the past for such purpose, such as Linter [8] or Abide [9]. However, they do not propose to fix the source code on behalf of the programmer, a process which could be automated for small changes and style errors. This report proposes the second version of a linter tool which can persist changes into source code based on rules defined by the programmer: Obey. Obey has originally been developed by A. Ghosn [10] and uses the internal representation of the Scala.Meta project to reflect on syntactic as well as semantic information provided by the typer phase of the Scala Compiler. Obey is partially inspired from Abide and proposes a minimalistic and user-friendly interface. It is also easy to integrate with SBT projects.

3.2 Design

Obey defines a lightweight model composed of rules returning warning messages. The rules themselves use the TQL library (Traversal Query Language) [11] to traverse the tree. This allow to define clean and simple basic blocks. Obey runs as a Scala compiler plugin, using the semantic information provided by the typer phase to reflect on trees in a given context. Finally, the compiler plugin can be automatically added to a project using an SBT AutoPlugin [12], which is presented in section 3.2.3.
The global design of Obey has gone through a few changes since its first version. This section presents its main concepts, before focusing on the features and modifications that have been added.

3.2.1 Interface

Obey allows to define rules by extending three basic Scala `traits`. All `traits` use the same syntax, but are used to differentiate the way the rules are applied at runtime:

`WarnRule` defines rules which will be used to solely print warnings, but any modification done by the rule will not be persisted.

`StatRule` defines rules which will be used to compute statistics. Statistics are displayed after all the rules have been applied and are aggregated together. It is an extension of `WarnRule`, and follows the same behavior, except for the way results are shown.

`FixRule` defines rules which will be used to modify the Abstract Syntax Tree. The results can then be persisted using token inference. See section 4 for more details.

Each `trait` extends the following definition:

```
sealed trait Rule {
  def description: String
  def apply: Matcher[List[Message]]
}
```

The `description` explains the behavior of the rule, while the `apply` method corresponds to the TQL code that will be used by the traverser during application. Note that `apply` returns a `Matcher`, which is a construct specific to TQL, and contains a list of messages as result.
Messages are basic blocks generated when a rule has been successfully applied. They are treated differently depending of the type of the rule generating them, and are defined as follow:

```
case class Message (
  message: String,
  originTree: scala.meta.Tree
)
```

A message returned after the application of a `WarnRule` or a `FixRule` will be reported immediately after the analysis of a compilation unit (which contains the Abstract Syntax Tree corresponding to one source file), while the results of a `StatRule` will be counted once the whole compilation process has ended and displayed in a grid showing the number of times a message with the same text description has been found. For an example on the output of such rule, see section 6.
Finally, tags can be associated to rules in order to allow the programmer to select only a specific subset to be applied. Tags are defined as annotations:

```
case class Tag (
  tag: String,
  others: String*
) extends StaticAnnotation
```

The following definition shows a simple rule that could be applied by Obey:

```
@Tag("Scala", "Style")
object EnforceTry extends WarnRule {

  def description = "Use util.Try rather than try/catch"
  def message(t: Term) = Message(description, t)

  def apply = collect {
    case t: Term.TryWithCases => message(t)
    case t: Term.TryWithTerm => message(t)
  }.topDown
}
```

For more details on the rules currently available, see section 3.4. For more details on the TQL syntax, please refer to the report *Traversal Query Language For Scala.Meta* by E. Beguet.

3.2.2 Compiler Plugin

The Obey compiler plugin runs after the typer phase of the Scala compiler, and is therefore able to use semantic information while reflecting on Scala.Meta trees. The compiler plugin stores the different flavors that an application of Obey should follow and output the warnings using the compiler reporter. Using SBT, it is possible to simply add the compiler plugin with the following command:

```
addCompilerPlugin("com.github.mdemarne"
  % "obey-compiler-plugin_2.11.6" % "0.1.0-SNAPSHOT")
```

The compiler plugin also filters the rules to apply based on their specific tag using an *Optional Filtering Language* (OFL), which was developed by A. Ghosn and is defined as follow:

```
tag := [\w\*]+.r
tags := { ~ tag ~ ([;,].r ~ tag).* ~ }
OFL := ("[+-]".r ~ tags).*
```

Below is an example of OFL which will apply all rules with the *Scala* tag, but not the rules with the *Dotty* tag:

```
+ {Scala} - {Dotty}
```

Note that this OFL is case-insensitive. Moreover, the negative set has a higher importance than the positive one.

The compiler plugin can receive the following options, passed to `scalac` at the command line:

`obeyRulesDir:<paths>` specifies paths to folders containing compiled rules. It is possible to specify multiple paths by separating them by `;`.

`obeyRulesJar:<paths>` specifies paths to jars containing compiled rules. It is possible to specify multiple paths by separating them by `;` as well. Rules specified by both commands will be loaded by the compiler plugin and applied together, regardless of their origin.

`warnings:<OFL>` specifies the OFL syntax that will be used to filter warnings and statistic rules.

`fixes:<OFL>` specifies the OFL syntax that will be used to filter fixing rules.

`dryrun` will tel Obey not to persist changes done on ASTs. This allows to let the programmer know which modifications will be applied to her or his code in a normal run, and has mainly an informative role.

`listRules` will stop the compilation process and not apply Obey. Instead, it will output all the rules that are selected.

The following example shows a valid combination of commands that will run a dry run of fixing rules.

```
-Xplugin:obey:dryrun
-Xplugin:obey:fixes:+{Scala}-{Dotty}
-Xplugin:obey:obeyRulesJar:~/.ivy2/local/com.github. ...
```

3.2.3 SBT Plugin

The SBT plugin abstracts the logic of the compiler plugin away from the programmer. It can be added to any SBT definition by specifying:

```
addSbtPlugin("com.github.mdemarne" %% "sbt-obey"
  % "0.1.0-SNAPSHOT")
```

The plugin defines a set of settings that can be overidden in the build definition and echoes the options that can be passed to the compiler plugin:

`obeyRules` specifies the OFL sentence for both fixing rules and warning rules. If specified, it will override the sentences given by `obeyFixRules` and `obeyWarnRules` (see below).

`obeyFixRules` specifies the OFL sentence for fixing rules only.

`obeyWarnRules` specifies the OFL sentence for warning rules only. Note that this also includes statistic rules.

`obeyRulesDir` specifies the directories of compiled rules.

`obeyRulesJar` specifies the jars containing compiled rules.

Below is an example of a valid build definition for a simple project:

```
lazy val root = (project in file(".")).
settings (
  scalaVersion := "2.11.6",
  ObeyPlugin.obeyRules := "+{Scala}-{Completeness,Dotty}",
) enablePlugins(ObeyPlugin)
```

Moreover, the SBT plugin allows to run Obey in different flavours, using dedicated commands:

`obey-list` will list all the available rules.

`obey-check` will apply all warning rules and fixing rules and show the corresponding warnings.

`obey-fix` will apply all fixing rules, show the corresponding warnings and persist changes to source files.

`obey-fix-dryrun` will apply all fixing rules and show the corresponding warnings. It will however not persist changes.

3.3 Implemented Rules

Obey as a plugin comes without a set of predefined rules. However, it is possible to fetch a default set of rules that can be automatically added to any Scala project as a dependency:

```
libraryDependencies +=
  "com.github.mdemarne" % "obey-rules" % "0.1.0-SNAPSHOT"
```

Those rules are ordered in packages and use default tags. There exists currently four kind of packages: `health` contains eleven style rules, `Statistics`, one general statistic rule, `dotty` contains a few rules for Dotty only, and finally `transformations` contains one experimental rule that can help migrate actors defines as Scala Actors into Akka Actors [13]. This experimental application is discussed in section 6.

3.4 Modifications Compared To The Previous Version

This section presents what has been changed since the first version of Obey:

- The rule behavior is clearly stated by `WarnRule`, `StatRule` and `FixRule`.
- The set of commands using Obey has been changed to reflect the clear separation of rule behaviors.

- There is no more default rule automatically added to Obey. This gives to the programmer the entire freedom to define her or his own set of rules. However, a basic set of rule is available.
- It is possible to add rules defined in a local repository as well as in jars.
- Statistic rules have been added.
- When running an Obey command such as `obey-check`, the compilaton process is stopped after the Obey phase. As an outcome, this might lead to unmatched dependencies if non-Scala source files had to be compiled (this for instance is the case for Java source files) as this is triggered by SBT independently. We filter out those files to avoid having errors coming from `javac` – the java sources files are simply not taken into account for Obey commands.
- Finally, by inferring tokens, Obey is able to persist modifications from Abstract Syntax Trees while keeping the original layout and comments of a source file. See section 4.

For more details on the Obey implementation, please refer to the Github repository of the project or A. Ghosn technical report, *Obey: Code Health for Scala.Meta*, 2015.

4 Persisting AST Modifications

Obey allows to create rules that can modify the Abstract Syntax Tree representation of a program. Unfortunately, using prettyprinters that do not take into account the initial sources to persist those changes is not enough in most cases, as the original code usually contains comments and formats that can play a crucial role in the maintainability of a program.
In this report, we present a different approach based on token inference that allows to keep the original layout.

4.1 General Approach

Scala.Meta uses specific tree definitions that define ASTs uniquely in term of strings and AST nodes. By doing so, they are sensibly simpler to manipulate than the trees of `scalac` and naturally suited for reflection.
Moreover, when parsed using the Scala.Meta parser, those trees contain a direct mapping from AST nodes to the tokens generated by the Scala.Meta tokenizer, which allow to retain more information about the layout of the original source code and ease the persistence of the changes, as explained below. On the other hand, trees converted after the typer phase of `scalac` do not contain such information.
When an AST is modified, the original token stream becomes outdated as well as the source code originally linked to it. In order to persist modified trees after a run of Obey, we need either to modify those tokens and reprint them, or operate at the string level itself – in other words, we either infer the new tokens or prettyprint the changes.
The second approach is used for instance by Scala-Refactoring [14] and Scalariform [15]. Scala-refactoring is a library providing automated refactoring for Scala code and can be integrated in an IDE such as Eclipse. In order to preserve the layout information, Scala-Refactoring uses two prettyprinters, each having a specific task. The normal prettyprinter prints code using standard techniques, while the second one reuse the layout that was present in the original code. This might be the case for instance if a method has been extracted from one object to be placed into another. When reprinting the source code, Scala-Refactoring then switches from one prettyprinter to the other based on the nature of the piece of code that needs to be output. Conversely, Scalariform allows to format source code automatically and works with its own parsers and produces strings.

Using Obey, there is no guarantee regarding the size and the locality of the changes that an AST can undergo, on the contrary of refactoring tools such as Scala-refactoring. Moreover, tokens contain more information regarding the articulations of the various parts of the source code, and thus allow a better grip on the sections that need to be reprinted. Using quasiquotes in Obey rules also allows to generate fragments of trees that are mapped to their own token sequences, which can easily be merged with existing streams.
Following those remarks, we have developed an implementation of a token inferencer able to produce tokens for changes while keeping as much as possible the layout and comments of the original source. Token quasiquotes developed by M. Duhem [16] also allow to easily generate those sequences, letting us design an implementation that is somewhat close to a prettyprinter in its structure and layout.

4.2 Adding Layout Information In Converted Trees

Forked repository of Scalahost: github.com/mdemarne/scalahost

Scala.Meta adds a phase to the compiler that converts ASTs produced by the typechecker into Scala.Meta trees. Those trees contain useful semantic information that can

be used in reflection as well as in tools such as Obey. However, scalac does not keep track of the original token stream themselves, although it maintains a mapping from nodes to positions in source files. As an outcome, we have developed a simple procedure that parses the original source file and merges its output with the converted tree, allowing to have both semantic and syntactic information.
Unfortunately those two trees might differ as they are not produced by the same parsers and as the typer phase is able to add more information. For instance, parsing source code directly using Scala.Meta produces the follwoing representation of tuples:

```
Type.Tuple(List(Type.Name("A"), Type.Name("B")))
```

On the other hand, the converted tree might be desugared:

```
Type.Apply(
  Type.Select(Term.Name("scala"), Type.Name("Tuple2")),
  List(Type.Name("A"), Type.Name("B"))
)
```

As part of this project, we have developed a minimalist function merging converted and parse trees covering a few of those corner cases. As work is currently on progress to directly add semantic information on top of parse trees, this implementation is only temporary and is somewhat rudimentary.

4.3 Inferring Tokens

Forked repository of Scalameta: github.com/mdemarne/scalameta

When a sub-tree is modified and placed back into the AST, its parent nodes are copied and the tokens they contain become obsolete. As an outcome, the `copy` constructor lazily call the token inference when the associated stream is accessed. This allow to easily propagate changes at a low cost from leaves to root.
Let's consider the following example:

```
def square(x: Double /* a square root */) = x * x
```

The AST for such a program looks like:

```
Defn.Def(
  Nil, Term.Name("square"), Nil,
  List(List(
    Term.Param(Nil, Term.Name("x"), Some(Type.Name("Double")),
      None)
  )),
  None,
  Term.ApplyInfix(Term.Name("x"), Term.Name("*"), Nil,
    List(Term.Name("x")))
)
```

We can now apply a simple transformation that will change `x * x` into `Math.pow(x,2)`. As an outcome,

```
Term.ApplyInfix(Term.Name("x"), Term.Name("*"), Nil,
  List(Term.Name("x")))
```

is transformed into:

```
Term.Apply(Term.Select(Term.Name("Math"), Term.Name("pow")),
  List(Term.Name("x"), Lit.Int(2)))
```

When tokens are fetched, the inferencer will thus reuse the tokens linked to all other sub-parts of the original `Defn.Def`, infer the tokens for the new `Term.Apply` and for the parent definition itself. The inferred token stream for the `Apply` above will look like:

```
Math (0..4), . (4..5), pow (5..8), ( (8..9), x (9..10), , (10..11),
  2 (11..12), ) (12..13)
```

As an outcome, the string corresponding to the token stream will still contain the comment present in the original code, as it is associated with the `Term.Param`, which was left unmodified:

```
def square(x: Double /* a square root */) = Math.pow(x, 2)
```

Since we want to lazily propagate changes, we never infer tokens for the tree as a whole. The inference is thus indirect: when inferring tokens for a specific node, the inferencer will lookup the token streams of the node's children, which will be computed if required – calling the inferencer once more. By doing so, the parent node does not need to know if its children are left untouched or are modified. One drawback of this approach is that only tokens are passed through the indirect recursion, while a prettyprinter could pass more information regarding the context in which a node needs to be printed. This has proven to be useful to put back parentheses in case of mixed infix operators. As an outcome, the inferencer needs to lookup the parent of a node in order to check whenever a term needs to be parenthesized, introducing a few more boilerplate code.
Our implementation of the inferencer follows the SLS specifications [17] with a few minor simplifications regarding parentheses inference, due to the extra lookup explained above.

4.4 Preserving High-Level Comments

The technique presented above allows to infer tokens with fine granularity. One drawback however is that comments embedded into token streams associated with modified nodes will not be preserved. Even if this is negligible for small comments inside the code itself – especially since those comments might become outdated based on the modification brought to the tree – Scala source code usually contain a lot of high-level notes present at the root of the file (e.g. scaladoc describing the global behavior of a class). Such comments are usually crucial for the programmer as they describe the general purpose and behavior of the source file itself. In the internals of Scala.Meta, the root of a source file is always a `Source` node containing the sequence of all top-level statements as well as the high-level comments. Following this implementation, modifying any part of the tree will force the inference of the top-level `Source` node and thus the high-level comments will not be preserved using the technique described above. We have therefore developed a special case using the original `Source` Node to preserve high-level comments under tree transformation.
Our implementation takes into parameter the original `Source` tree and extracts its corresponding tokens. It then gets all top-level statements inside this original `Source` and maps them to their corresponding statements in the new one, and replace them progressively in the token stream representing the whole file. If more statements are present in the modified `Source`, they will be added at the end of the file. On the other hand, if less statements are present, the ones that were remove will simply be replaced by an empty token stream in the global sequence.
This approach has the benefit of keeping the general, top-level comments, but does some simplifications:

- The approach assumes that rules do not operate often directly on top-level statements and does not shuffle them. This is due to the fact that the mapping between the statements present in the original `Source` and the new one is done by a trivial zip,
(i.e. `originSource.stats zip modifiedSource.stats`).

- Special indentation before the beginning of a statement might produce surpring outputs when fully inferred. This is due to the fact that this indentation is not contained in the statement itself. For instance, the sequence of tokens corresponding to the class in the following source file:

```
        class A
//_____^ <-- Note the spacing
```

  will be:

```
Tokens(class (8..13),   (13..14), A (14..15))
```

  thus ignoring the original spacing at the beginning of the line.

4.5 Preserving Indentation

Preserving indentation is simple when a statement is moved at the same depth in the tree, as it is then sufficient to take the original token stream as is. On the other hand, this is not trivial when moving statements to a place where the layout should be different (for instance, by refactoring the code and moving a statement inside an object into a class inside the object itself).
In such cases, the indentation needs to be re-inferred, even if the original AST node was left untouched by Obey. To execute such manipulation, the inferencer is based on the following simplifications:

- The standard indentation is used, regardless of the one in the original code. A future work could be to guess the indentation scheme used in the original source code and use it to infer tokens with the proper formatting.

- It assumes that spaces are used for indentation in the original source file, as specified by the Scala Style Guide [18]. If tabulations are used, the indentation scheme will be partially mixed.

- When the token sequence corresponding to a node is on multiple lines, the last one possesses only indentation that is not directly linked to its content. To illustrate this assumption, let's take a simple example:

```
class MyMath {
  def square(x: Int) = {
    x * x
  }
}
```

  In such a class, the indentation that is linked to the `Term.Block` containing the body of the `square` function will lead to the following representation:

```
{
    x * x
  }
```

As shown, the brackets are not aligned. This is due to the fact that the first line of the `Term.Block` starts at the opening bracket, while the last line ends at the closing one. As an outcome, no indentation is present on the first line, while all others still contain the indentation from the source file. In order to represent the content of this `Term.Block` properly aligned, we can remove all the indentation present on the last line to all other lines above. In our case, it corresponds to one shift to the left. We will then get:

```
{
  x * x
}
```

Using those two assumptions, the inferencer is able to properly re-indent the sequence of tokens by shifting the lines to the right when required. Moreover, this approach will keep the original layout inside unmodified nodes.

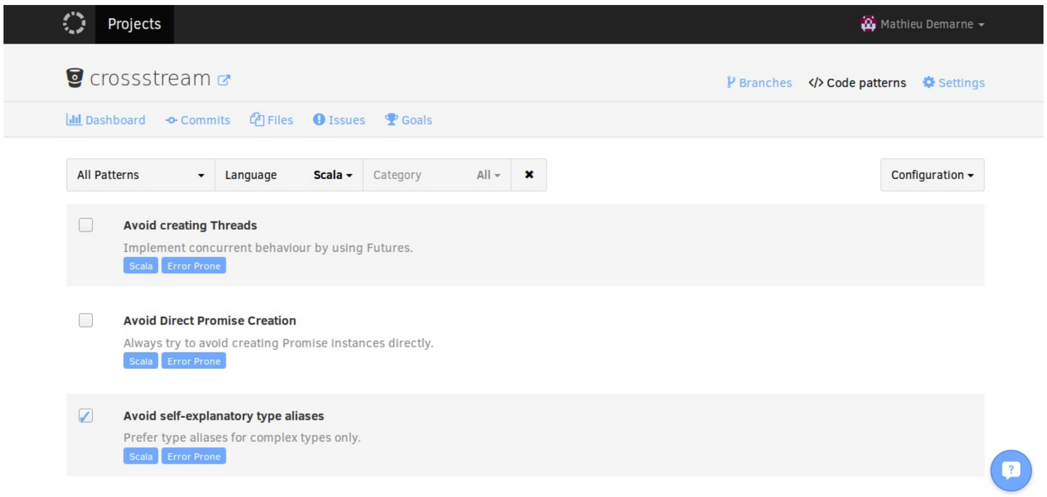
5 Style Checking With Codacy

Codacy home page: codacy.com

Codacy is a company proposing automated continuous static analysis for various programming languages. It was founded in 2012 by Joao Caxaria and Jaime Jorge and has receive founding from venture capital firms both from the United Kingdom and Spain. The company is based in London.
We have worked jointly with Johann Egger from Codacy in order to have an industrial use case of the way patterns are defined in Obey for style rules. The solution presented in this section is the result of our direct collaboration.

5.1 Original Engine

When a project is configured to use the Codacy SaaS platform, each change done into its repository triggers an analysis. The Codacy platform fetches the modifications and generates warnings based on the entire code base.
While analyzing JavaScript code is relatively straightforward as it is not strongly typed, semantic information on Scala code can be a non-negligible asset when applying style rules. A simple example consists in analyzing all `.get` calls: using semantic information, it is possible to determine when an internal `Select` is actually done on a Scala `Option`, thus allowing to raise a warning.
Unfortunately getting semantic information has a price in resources, as the source code has to go through a typechecker. For performance reasons, triggering the typechecker for each analysis is not suitable at scale.
The original engine used for the analysis parses Scala source code using the Scala Reflect toolbox [19] before converting them into Json. The analysis is then ran using rules defined in JavaScript, which directly operate on top of the Json representation of the Scala code. Users are able to select among a set of predefine rules that will be used to analyze their projects:



[ ] Avoid creating Threads
Implement concurrent behaviour by using Futures.
Scala | Error Prone

[ ] Avoid Direct Promise Creation
Always try to avoid creating Promise instances directly.
Scala | Error Prone

[✓] Avoid self-explanatory type aliases
Prefer type aliases for complex types only.
Scala | Error Prone

Illustration 1: Codacy dashboard

As explained, this approach needs to translated Scala parse trees into Json prior to the analysis, adding a small overhead in term of performance.

5.2 Proposed Approach

Running Obey rules directly on top of Scala trees removes the overhead of this translation into Json. Moreover, working directly on ASTs allows a precise analysis even if they do not contain semantic information.
The adaptation of the Obey model has been done in two phases. First of all, the parser from Scala Reflect has been replace by the one proposed natively by Scala.Meta. Since those trees are not converted to Json anymore, they are sent to the analyzer directly as `scala.meta.trees`. Secondly, the analyzer itself filters out the rules to apply based on the analysis configuration received from the server, apply them, formats their output properly and creates a response object containing all warnings or potential parsing errors.
The adaptation from the Obey model has proven to be mostly straightforward. The major modifications are:

- Tags do not exist anymore. Instead, each rule receives a unique identifier used by the engine during filtering.
- The engine itself does not use a standard reporter to report errors. Instead, warnings are mapped to line numbers, allowing to extract the faulty slices of code from the original source files and display them nicely to the user.
- The analyzer is an isolated engine. It receives analysis requests using the HTTP POST protocol and returns the results over HTTP as well. The Scala.Meta engine had therefore to be integrated into this architecture.

A total of nine rules inspired both from the default package of Obey and of the patterns present on the Codacy platform have been implemented for prototype testing:

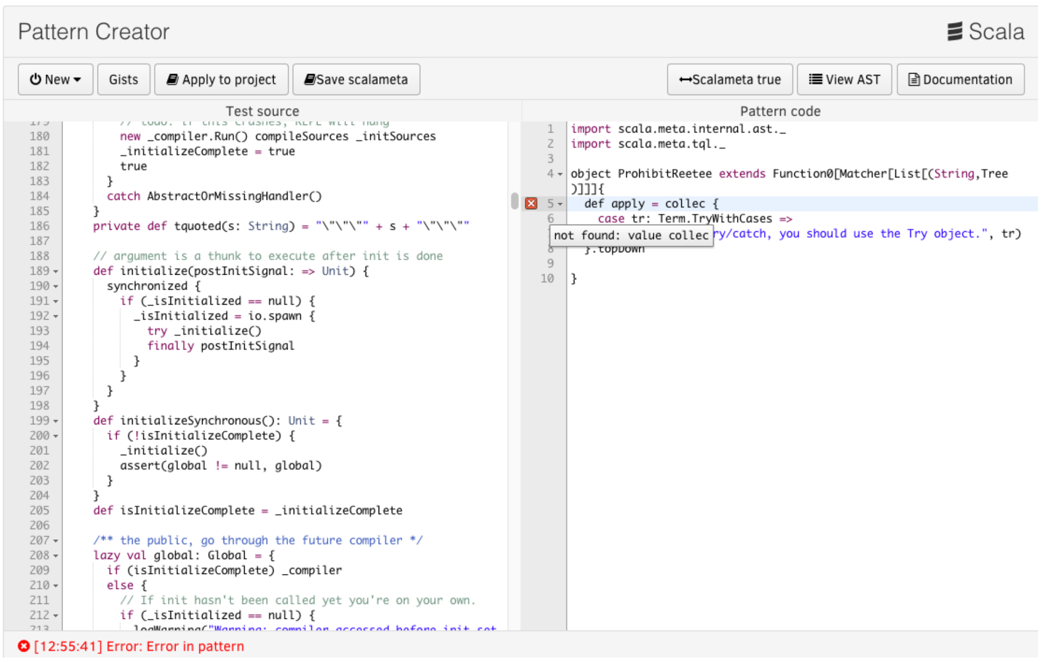
1. We raise a warning when `case class`es with no parameter and no body could instead be case objects.
2. Partial implementation (e.g. `val x = ???`) are detected.
3. Conventional `try { ... } catch {...}` can be replaced by the use of the `util.Try` object, as it ensures that the user is properly considering error handling and strengthen type checking.
4. The `null` literal is prohibited. It is instead suggested to use `Option`s.
5. `return` is implemented as exception throwing and catching in bytecode. Moreover, Scala ensures that the last statement in a method is automatically returned. We raise a warning if such a keyword is found.
6. `while` loops are deprecated when using strict functional programming, and finding some raises warnings as well.
7. We suggest that `var`s that are never re-assigned could be `val` instead.
8. Calling `.get` on an option will throw an exception. We therefore detect such calls. The best option in such a case is to use functional constructs such as `map` and `flatMap`.

Unfortunately parsing trees using the Scala.Meta parser does not allow us to have semantic information, as we would have to analyse the project as a whole, which is not suitable in term of performance. Due this lack of information, the two last rules are implemented using small tricks. Rule 7 is implemented by considering variable names regardless of their scope and might therefore fail to return warnings in some cases. Rule 8 on the other hand checks that the call to `.get` does not take any parameter:

```
def apply = collect {
    case Term.Apply(Term.Select(_, t @ Term.Name("get")), args)
      if args.size == 0 =>
        message(t) // generates a warning
    case s @ Term.Select(_, t @ Term.Name("get"))
      if s.parent.map(!_.isInstanceOf[Term.Apply])
         .getOrElse(false) =>
        message(t) // generates a warning
  }.topDown
```

5.3 User-Defined Rules

Codacy also proposes a prototype of an interface allowing users to define their own warning patterns. This interface has been adapted to allow to define rules directly in Scala.
One drawback using rules defined in Scala directly is that they need to be compiled prior to their application. To solve this issue, we have implemented a specific rule builder which parses and compiles source code on-the-fly using virtual directories (see `scala.reflect.io.VirtualDirectory`) and a dedicated compiler instance (using Global from the Scala Compiler: `scala.tools.nsc.Global`). As an outcome, the builder only needs to get the source code of the rule to build as a `string` and `scala-library` and `scalameta` as library dependencies. The return type of the builder is a `Try[Rule]`. If an error occurred a `Failure` containing the compilation error is returned instead.
As an outcome, the user can define and test its rules and save them directly in the interface. Thanks to the builder, user-defined rules can then be seamlessly used throughout the platform.
Below are a few screenshots of the prototype interface analyzing a piece of code from Spark [20]. These illustrations are courtesy of Johann Egger and the Codacy team.



```
        new _compiler.Run() compileSources _initSources
        _initializeComplete = true
        true
      }
      catch AbstractOrMissingHandler()
    }
    private def tquoted(s: String) = "\"\"\"" + s + "\"\"\""

    // argument is a thunk to execute after init is done
    def initialize(postInitSignal: => Unit) {
      synchronized {
        if (_isInitialized == null) {
          _isInitialized = io.spawn {
            try _initialize()
            finally postInitSignal
          }
        }
      }
    }
    def initializeSynchronous(): Unit = {
      if (!isInitializeComplete) {
        _initialize()
        assert(global != null, global)
      }
    }
    def isInitializeComplete = _initializeComplete

    /** the public, go through the future compiler */
    lazy val global: Global = {
      if (isInitializeComplete) _compiler
      else {
        // If init hasn't been called yet you're on your own.
        if (_isInitialized == null) {
```

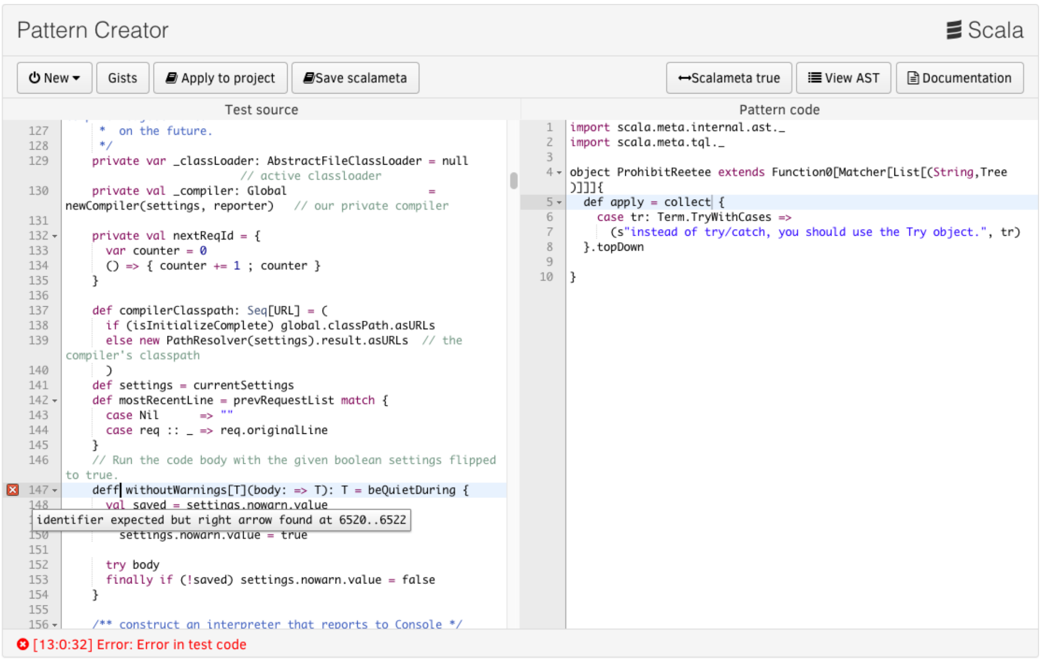
```
import scala.meta.internal.ast._
import scala.meta.tql._

object ProhibitReetee extends Function0[Matcher[List[(String,Tree
)]]]{
  def apply = collec {
    case tr: Term.TryWithCases =>
not found: value collec ry/catch, you should use the Try object.", tr)
  }.topDown

}
```

[12:55:41] Error: Error in pattern

Illustration 2: Error detected by the rule builder



```
    *  on the future.
    */
   private var _classLoader: AbstractFileClassLoader = null
                              // active classloader
   private val _compiler: Global                          =
newCompiler(settings, reporter)   // our private compiler

   private val nextReqId = {
     var counter = 0
     () => { counter += 1 ; counter }
   }

   def compilerClasspath: Seq[URL] = (
     if (isInitializeComplete) global.classPath.asURLs
     else new PathResolver(settings).result.asURLs  // the
compiler's classpath
     )
   def settings = currentSettings
   def mostRecentLine = prevRequestList match {
     case Nil      => ""
     case req :: _ => req.originalLine
   }
   // Run the code body with the given boolean settings flipped
to true.
   deff withoutWarnings[T](body: => T): T = beQuietDuring {
     val saved = settings.nowarn.value
identifier expected but right arrow found at 6520..6522
       settings.nowarn.value = true

     try body
     finally if (!saved) settings.nowarn.value = false
   }

   /** construct an interpreter that reports to Console */
```

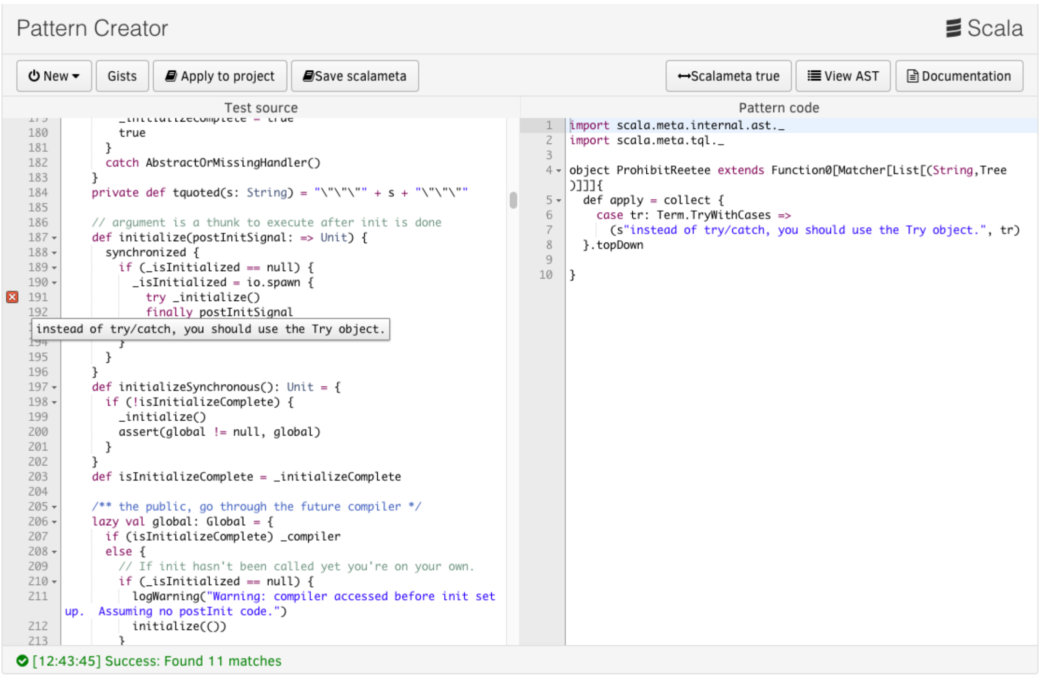
```
import scala.meta.internal.ast._
import scala.meta.tql._

object ProhibitReetee extends Function0[Matcher[List[(String,Tree
)]]]{
  def apply = collect {
    case tr: Term.TryWithCases =>
      (s"instead of try/catch, you should use the Try object.", tr)
  }.topDown

}
```

[13:0:32] Error: Error in test code

Illustration 3: Error detected by the parser during analysis



```
       _initializeComplete = true
       true
     }
     catch AbstractOrMissingHandler()
   }
   private def tquoted(s: String) = "\"\"\"" + s + "\"\"\""

   // argument is a thunk to execute after init is done
   def initialize(postInitSignal: => Unit) {
     synchronized {
       if (_isInitialized == null) {
         _isInitialized = io.spawn {
           try _initialize()
           finally postInitSignal
instead of try/catch, you should use the Try object.
         }
       }
   }
   def initializeSynchronous(): Unit = {
     if (!isInitializeComplete) {
       _initialize()
       assert(global != null, global)
     }
   }
   def isInitializeComplete = _initializeComplete

   /** the public, go through the future compiler */
   lazy val global: Global = {
     if (isInitializeComplete) _compiler
     else {
       // If init hasn't been called yet you're on your own.
       if (_isInitialized == null) {
         logWarning("Warning: compiler accessed before init set
up.  Assuming no postInit code.")
         initialize(())
       }
```

```
import scala.meta.internal.ast._
import scala.meta.tql._

object ProhibitReetee extends Function0[Matcher[List[(String,Tree
)]]]{
  def apply = collect {
    case tr: Term.TryWithCases =>
      (s"instead of try/catch, you should use the Try object.", tr)
  }.topDown

}
```

[12:43:45] Success: Found 11 matches

Illustration 4: The rule is applied and a warning is raised

Note that all errors are either directly coming from the Scala.Meta parser or the Scala compiler.

5.4 Outcomes

The collaboration with Codacy shows that using the Obey model allows to define valuable style rules. In the future, the development of TASTY [21] might allow to use semantic

information in style rules by analyzing serialized trees provided either by the user or computed on-the-fly.
Nevertheless, applying rules in Scala rather than JavaScript reduces time overhead, while using a typed language with powerful pattern matching fits more in the requirements of such a program.
As an outcome, a next development step on the Codacy side will be to replace the current engine with an evolution of the prototype using Scala.Meta and transfer all rules currently implemented in JavaScript to TQL syntax.

6 Experimentations With Obey

Working jointly with Codacy already shows the value that can be brought by using the Obey model for style rules. As an outcome, we focused our experimentation on other areas. First of all, we wanted to show that our plugin implementation could run on large projects. Finally, we wanted to draft a use case of Obey that will allow to migrate code across versions of an underlying framework or library.

6.1 Collecting Statistics

We have ran Obey on the core code base of Scalaz [22], which is an extension of the Scala library for functional programming. We have used our statistic rules to count the number of various Scala constructs used throughout the system. As an outcome, we have found very few `vars` – which is better when considering pure functional programming, as scalaz does. The number of statements such as `return` and `do` or `while` is also small. The following output shows the number of times a specific construct was detected:

```
[info] Compiling 249 Scala sources to ...
[info] Global Statistics:
[info] Type                 Count
[info] ==========================
[info] def                  7812
[info] val                  861
[info] trait                817
[info] class                479
[info] object               346
[info] partialFunction      194
[info] var                  23
[info] list                 18
[info] try/catch            17
[info] do-while             15
[info] potential Option.get 6
[info] set                  6
[info] return               1
```

Running Obey on a large project was successful, although the compilation time was greater by a factor of two. However, this project did not focus on performance, and as an outcome the various steps of the rule application process could be optimized to reduce this overhead.

6.2 Potential Use Case For Migration

github.com/mdemarne/Obey-migration-example

Scala Actors are deprecated and have been replaced by Akka Actors since the version 2.11.0 of the Scala Programming Language [23]. As an outcome, actors implemented using the old toolkit need to be migrated. Using Obey, we have developed a small migration plan for simple Actors. This allows to show the potential of Obey with migration rules and the implementation of the token inference. Note that this rule however does not use semantic information, hence some of its particular equality comparisons between names:

```
/* Changes "act" to "receive" */
val transformDef = (focus {
  case Defn.Class(_, _, _, _, Template(_, parents, _, _)) =>
```

```
      parents.exists(p => p match {
        case pp: Name => pp.value == "Actor"
        case _ => false
      })
  } andThen (transform {
    case t @ Defn.Def(_, nm, _, _, _, body1) if nm.value == "act" =>
      val findPartialFunction = (collect {
        case Term.Apply(
            Term.Name("receive"),
            (body2: Term.PartialFunction) :: Nil
          ) => body2
      }).topDownBreak
      findPartialFunction(body1).result.headOption match {
        case Some(newBody) =>
          t.copy(
            name = Term.Name("receive"),
            body = newBody,
            decltpe = None
          ) andCollect ...
        case None => t andCollect ...
      }
  }).topDownBreak).topDown

  /* Changes the import clause */
  val changeImport = (transform {
    case s: Import if s.show[Code].contains("scala.actors._") =>
        q"import akka.actor._"
    ...
  }).topDown

  def apply = transformDef + changeImport
```

This rule first focuses on all classes extending `Actor`, finds the `act` method it should contain and the `PartialFunction` that is triggered when a message arrives. It then renames the `act` method into `receive`, and move the `PartialFunction` to the root of its body. One drawback of this approach is that other parts of the `act` function are not preserved, but the rule could be extended to support more complicated constructs. The snippets below show a part of such object:

```
  import scala.actors._
  /* Simple actor repeating what you send to it */
  class Echo(times: Int) extends Actor {
    def act {
      while (true) {
        receive {
          /* Repeat Strings and Ints */
          case s: String => repeatString(s)
          case i: Int => repeatInt(i)
          case x => println(s"Dunno what that is: $x.")
        }
      }
    }
    ...
  }
```

This is transformed as follow when the migration rule is applied:

```
import akka.actor._
/* Simple actor repeating what you send to it */
class Echo(times: Int) extends Actor {
  def receive = {
    /* Repeat Strings and Ints */
    case s: String => repeatString(s)
    case i: Int => repeatInt(i)
    case x => println(s"Dunno what that is: $x.")
  }
  ...
}
```

This code is compliant with the Akka syntax for Actors and compiles. Note that the rule presented above will work for simple Actor definitions only, even if it could be extended, as explained.
However, it shows the potential of using Obey to define migration rules that could be automatically implemented, as well as the preservation of the original layout and comments under transformation, thanks to token inference.

7 Future Work

7.1 Obey

Only some of the enhancements proposed by A. Ghosn were added to this second version of Obey, as we focused mainly on token inference and the integration with Codacy. In order to have an exhaustive list of potential improvements, we add here the enhancements proposed in the report *Obey: Code Health for Scala.Meta* as well.
The potential enhancements are as follow:

- Allow a better rule distribution. Note that so far, we are able to distribute rule using the `obey-rules` default package, but this one need to be added as a project dependency, which mean that the rules will be available in the classpath of the project. This could be avoided.
- Using a tag hierarchy to classify rules. If the set of available rules grows, a more complex implementation of the tag system might be required.
- Specify the rule application order. There is so far no guarantee, as there is no way to ensure that rules are loaded in the same order. Moreover, the current rule design does not allow to distinguish them.
- Implement additional rules. Documentation on the Github page of the project has been added in this direction.
- Run parts of the Obey model outside of the compiler plugin. For instance, there is no need to trigger the compiler plugin with the `obey-list` command. Similarly, rules that do not require semantic information could be run as standalone, using the Scala.Meta parser.
- Add compatibility with other plugins. Some plugins add sources to compile using dedicated procedures. For instance, the Play Framework translates web routes to Java and compiles them using `javac`. As an outcome, stopping the compilation procedure after the Obey phase does not prevent the compilation of those routes, which returns an error since it is dependent of the output of the compilation of the Scala source files. Even if the current Obey SBT plugin removes the `.java` files from the sources when running commands such as `obey-check`, those do not include the special Java files generated by Play. So far, we have not found a good way to prevent it [24].

Moreover, a few enhancements could be done regarding token inference:

- Extract indentation from the original source file. So far, we assume an indentation of two spaces.
- Preserve more comments: the way high-level comments are preserved could be abstracted and generalized to inner nodes as well.

7.2 Codacy

Working with Codacy has been a great experience. However, rather than just returning warnings, the style rules defined using the Obey model could propose changes. Those changes could be shown to the user using token inference. A potential work that could be done for the benefit of both Obey and Codacy would therefore be to propose a better framework to show potential transformations to the user, leaving the choice to apply them or not. Note that the persistence process could be automated as well, considering only subparts of the modified tree.

8 Conclusion

In this report, we describe how we extended the definition of Obey by allowing to persist changes brought to Abstract Syntax Tree while preserving the original layout. Our experimentation has shown that Obey can scale on large code bases and preserve layout and comments under transformation.
We also present the work done in collaboration with Codacy in order to have a use case of Scala.Meta in the industry. The collaboration was successful, resulting in adoption of Scala.Meta in Codacy's core framework and enabling new functionality in Codacy's code review platform [25].

8.1 Acknowledgments

We would like to thank all the Codacy team for their precious time and collaboration. This collaboration as well as the work done on Obey and Scala.Meta would also not have been possible without the precious insights and the help of Eugene Burmako and Denys Shabalin, who designed the core components of Scala.Meta.

References

[1] Phabricator: *github.com/phacility/phabricator*

[2] Crucible: *atlassian.com/software/crucible/overview/code-quality-jira*

[3] Various Continuous Integration platforms: *travis-ci.org, shippable.com, github.com/gitlabhq/gitlab-ci, github.com/jenkinsci*

[4] Codacy home page: *codacy.com*

[5] Scala.Meta: *scalameta.org*

[6] Play Framework: *playframework.com*

[7] *Lint, a C Program Checker*, S.C. Johnson, 1978

[8] Linter: *github.com/HairyFotr/linter*

[9] Abide: *github.com/scala/scala-abide*

[10] *Obey: Code Health for Scala.Meta*, Adrien Ghosn and Eugene Burmako, 2015

[11] *Traversal Query Language For Scala.Meta*, Eric Beguet and Eugene Burmako, 2015

[12] SBT AutoPlugin: *scala-sbt.org/0.13.5/api/index.html#sbt.AutoPlugin*

[13] Akka: akka.io

[14] *Scala-Refactoring*, Mirko Stocker, 2010

[15] Scalariform: *mdr.github.io/scalariform*

[16] Token quasiquotes were originally defined in Martin Duhem Parsermacro project: *github.com/Duhemm/parsermacros*

[17] *Scala Language Specification*, Martin Odersky, 2014

[18] Scala Style Guide: *docs.scala-lang.org/style* – scala-lang.org

[19] Scala Reflect Toolbox: *scala-lang.org/api/2.11.0/scala-compiler/index.html#scala.tools.reflect.ToolBox*

[20] Spark: *github.com/apache/spark*

[21] *TASTY*: Serialized Typed Abstract Syntax Trees, see *Martin Odersky presentation at the San Francisco ScalaDays 2015: "Scala – Where It Came From, Where It Is Going"*

[22] Scalaz: *github.com/scalaz/scalaz*

[23] Actor Migration Guide: *docs.scala-lang.org/overviews/core/actors-migration-guide.html*

[24] See the question *Stopping compilation after a compiler phase while using the Play Framework* on stackoverflow.com

[25] A quick look at Scalameta: *blog.codacy.com/2015/06/04/a-quick-look-at-scalameta*, Johann Egger, 2015