Obey: Code Health for Scala.Meta

Technical Report

EPFL, Switzerland

Adrien Ghosn, Eugene Burmako

firstname.lastname@epfl.ch

Abstract

Obey is a user-friendly tool that helps programmers enforce code health requirements in their projects. Requirements are expressed as rules, written with the TQL library[1] combinators used to traverse scala.meta trees[2], that generate compiler warnings and can automatically correct the source code. Programmers are provided with a set of basic rules and can easily implement and use their own. Obey can be used to systematically enforce user-defined requirements, to automate migration between different library versions or simply to format source code. In this paper, we will present the user interface, the implementation details and the results we obtained during this semester project.

Keywords: code health, compiler, plugin, scala, abstract syntax tree, SBT, scala.meta, TQL library, meta-programming

Main Repository: https://github.com/aghosn/Obey

1. Motivation & Related Work

Compilation warnings should be easy to add, remove and modify without requiring any modification of the compiler. As a programming language evolves, new deprecated features and potentially dangerous code practices appear. Implementing all the warnings inside the compiler itself slows down the evolution process and increases its complexity. Furthermore, it requires a deep knowledge of the compiler's internal structure to modify or correct one such warning.

Historically, Lint[3] was a program developed at Bell Labs used to flag non-portable constructs and potential bugs in C language source code. The term Lint-like now refers to tools used to report suspicious code constructs in software in various languages. Most of them rely on static code analysis to identify dangerous patterns in the source code. They also enable to add, remove and modify warnings easily without requiring any modification of the compiler's code.

One example of advanced linter-tool for C++ is clang-tidy[4]. It provides an extensible framework to diagnose & fix programming errors and can be easily extended with new 'checks'. Clang-tidy also

enables a fine-grained selection of checks to be applied and provides the user with an optional auto-correction of the source code when fixes are available. This linter is very similar in terms of goals to what we strive to implement with Obey.

The “*Toward a Safer Scala*” presentation[5] describes several Lint-like tools for the Scala programming language. The Scala Abide[6] project inspired our own implementation and we decided to dedicate section 5 to the comparison between the two projects. Worth noting, the scalac option '-*Xlint*' performs checks for several corner cases and allows selective enabling since version 2.11.4. The WartRemover project[7] on the other hand, is a compiler plugin easy to integrate with sbt. The project is well documented and has a very simple interface & integration process that corresponds to what we would like to provide in Obey. WartRemover also allows the users to define their own WartTraverser, i.e., rules, and provides sbt command-line tools in order to quickly probe the source code.

In Obey, our goals are: 1) to expose a small & easy to implement interface, 2) to provide an extensible & flexible framework that enables the users to define and reuse their own rules with as little overhead and configuration as possible, 3) to make Obey easy to integrate into existing projects, 4) to provide tools & commands that enable quick and efficient probing of code.

2. Design & Interface

In this section, we will present Obey’s interface and high level design. We first expose the basic building blocks, i.e., the Rule, Message & Tag models. Then, we present the interface with the user, that is, the compiler plugin & the sbt-plugin.

2.1. The Rule Model

The rule model defines the interface available to the users in order to implement their own rules. It is divided into three elements: Rule, Message, Tag.

Rules are the basic blocks used to express the code health requirements. The Rule trait is defined in *scala.obey.model* and has the following interface:

```
trait Rule {

  def description: String

  def apply: Matcher[List[Message]]
}
```

Figure 1: Rule Interface

The *description* is used for display purposes. It should be a concise explanation of what the rule does. The *apply* method is the implementation of the rule itself. Its return type is *Matcher[List[Message]],* where the *Matcher* type is defined in the TQL library. Once applied to a tree, the result has the type and interface described in Figure 2 with parameter type *List[Message]*.

```
trait MatchResult[T] {
  def result: T
  def tree: scala.meta.Tree
}
```

Figure 2: MatchResult[T] Interface

The user can select to only generate warnings, contained in the *result* value, or can persist the fixes performed in the rule and correct the source code by replacing the original tree with the *tree* obtained in the *MatchResult*.

The *Message* type is defined in *scala.obey.model* and has the following signature:

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

Figure 3: Message Interface

Where the *message* describes the warning we generated, and the *tree* is the associated scala.meta tree for which we produced the warning. The *Message* type also has a *position* value automatically computed from the tree and used by the reporter to display the warnings.

Rules can produce warnings and/or rewrite the abstract syntax trees to comply with the code health requirements. Figure 4 presents an example of rule that reports and corrects all newly created lists casted as a sets, and proposes to fix them by defining a set instead.

```
@Tag("List", "Set") object ListToSet extends Rule {
  val description = "defining List.toSet is defining a Set"

  def message(t: Defn.Val)= Message(s"The assignment $t creates a useless List", t)

  def apply = {
    (transform {
      case t @ Term.Select(Term.Apply(Term.Name("List"), l),Term.Name("toSet")) =>
        Term.Apply(Term.Name("Set"), l) andCollect message(t)
    }).down
  }
}
```

Figure 4: Example of Rule

In this example, the *transform* keyword from the TQL library enables us to match the case we are looking for and return a new, corrected tree. The *andCollect* keyword enables us to also emit a warning message at the same time. The *.down* selects the traversal we use for this rule, that is, here we decided to traverse the entire tree in a top-down fashion in order to look for all matching constructs. Applying this rule to a tree will return a *MatchResult* as described in Figure 2. If we decide to use this rule to format the source code, the original tree will be replaced with the corrected one.

Rules are further tagged with *Static Annotation's* of *Tag* type, defined in *scala.obey.model* and with the interface defined in Figure 5.

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

Figure 5: Tag Interface

Tags are used to filter rules and solely rely on strings. Working with strings enables to keep

enough flexibility at the user's end compared to, for example, predefined values in an enumeration. One can define a dedicated tag for rules that pertain to a specific project, e.g., "*play*" for the Play framework. We can create as many levels of specification as we want using the tag system, e.g., "2.2", "2.3" etc.

2.2 The Compiler Plugin

Obey is a compiler plugin that runs after the typer phase, that is, the *ObeyPlugin* extends *scala.tools.nsc.plugins.PluginComponent* and defines a phase named "*obey*". To use the Obey plugin, one simply has to add the following line to the *build.scala* project settings:

```
addCompilerPlugin("com.github.aghosn" % "plugin_2.11.2" %
"0.1.0-SNAPSHOT")
```

Figure 6: SBT line to add the compiler plugin

The ObeyPlugin defines four options that help the user to select and manage the rules to be applied, all of them preceded by the *-Xplugin:obey:* as per usual for a compiler plugin.

The *ListRules* option is used to output the list of rules that are going to be used, according to the current project settings. The output distinguishes the rules that are going to be used to format the code and the ones that will simply output warnings. One can also list all available rules by using the option *ListRules: -all*.

The *addRules:<path>* is used to specify an arbitrary path to user-defined compiled rule classes. This option enables the user to share locally defined rules, found at *<path>*, between different projects without having to copy them in the different projects where they are used.

For the next options, we defined an *Option Filtering Language* (OFL) that enables the user to perform a fine-grained selection of the rules to be applied, based on tags and rule names. The OFL grammar is defined by the following regular expression parser:

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

Figure 7: OFL grammar

Where the '*' character can be used to match any alpha-numeric sequence inside a tag at the specified position. Furthermore, the user can also simply specify '++' or '- -' to respectively add all or remove all the rules. These two special operators are aliases for *+{*}* and *-{*}*.

For example, Figure 8 selects all the rules that contain one or several of the tags or are named "*List*", "*Set*" and/or "*VarInsteadOfVal*", while excluding the ones with the tag/name "*Dotty*". It is worth noting that tags are not case sensitive, hence the two versions on Figure 8 are strictly equivalent. Furthermore, the negative set has a higher importance that the positive one, that is, if one rule matches both the positive and negative set, it will not be included in the final list of filtered rules.

```
+{List, Set, VarInsteadOfVal} – {Dotty}
+{list, set, varinsteadofVal} -{dotty}
```

Figure 8: Example of OFL expression

The *warn:<OFL>* is used to specify filters for the warning rules, i.e., rules that will generate warnings but for which we will discard the resulting formatted tree, hence preventing any modification from being applied to the original source code. The *<OFL>* stands for a valid OFL expression as defined above. The default behavior is to add all the available rules to the warning list, that is, both the ones provided by the Obey framework and the ones added by the user.

The *fix:<OFL>* is used to specify filters for the formatting rules, i.e., rules that will be used to both generate warnings and format the original source code. The default behavior is to disable source code formating, that is, no modification is performed unless specifically asked by the user using this option.

Finally, the *all:<OFL>* applies filters to restrict the set of rules considered by both the *fix* and the *warn* options. By default, all rules are considered by the filtering options.

Figure 9 gives a valid set of options passed to the compiler.

```
-Xplugin:obey:all:+{Dotty}
-Xplugin:obey:fix:+{List*}-{Var}
-Xplugin:obey:warn:-{Val,Var*}
-Xplugin:obey:addRules:/home/user1/rules/target/scala-2.11/classes
```

Figure 9: Example of plugin Options

2.3. An SBT Plugin

The sbt-obeyplugin encapsulates the compiler plugin into a simpler, higher-level abstraction. The sbt-obeyplugin is an *sbt.AutoPlugin* and can be used by adding the following line to the *plugins.sbt* file:

```
addSbtPlugin("com.github.aghosn" %% "sbt-obeyplugin" % "0.1-SNAPSHOT")
```

Figure 10: sbt-plugin line in plugins.sbt

Afterwards, the plugin needs to be enabled in the project definition inside the *build.scala* as described in Figure 11.

```
lazy val myProject = Project(...) enablePlugins(obeyplugin)
```

Figure 11: build.scala Setting

The sbt plugin will then simply add the Obey compiler plugin as described in *2.2.* to the project settings.

The sbt-obeyplugin exports useful *sbt.settingKey's* that are easy to modify in the project settings. The *obeyFix settingKey* is an alias for the *obey:fix:* option described in *2.2.*. One simply has to assign it a valid OFL expression. The *obeyWarn* is an alias for the *obey:warn:* option and works in a

similar way. Finally, the *obeyRules* is an alias for the *obey:addRules:* option and enables to specify a path to user-defined compiled rules.

The sbt-obeyplugin also defines useful sbt console commands that enable to quickly run the compiler plugin and print information about the current settings used in the project. These commands stop the compilation process after the "*obey*" phase and cannot be used to perform a full compilation of the source code.

The *obey-fix* command allows the user to run the Obey compiler plugin to format the source files according to the selected rules. When no argument is specified, the command will use the filters defined in the project settings, i.e., the OFL expression passed to the *obey:fix:* option. Rules that are used according to the *obey:warn:* option will be ignored. The user can override the filters by passing a valid OFL expression argument to the command.

The *obey-check* command allows the user to run the Obey compiler plugin and gather all warnings generated by the rules filtered according to the *obey:warn:* option. By default, the settings specified in the project are used. The rules that match the *obey:fix:* option will be ignored. As for the precedent command, the user can override these settings by passing a valid OFL expression as argument.

The *obey-list* command enables to display the list of selected rules according to the defined settings. The command will distinguish the rules selected to format the code, and the ones that will simply generate warnings. The command also accepts the *-all* argument that will then print all the rules available, that is, all the rules provided by the Obey framework and all the ones added by the user.

```
> obey-list
[info] Tag Filters:
[info] all: -> +{} - {}
[info] fix: -> +{List.*} - {}
[info] warn: -> +{} - {}
[info] List of selected Rules:
[info] Warn Rules:
[info] VarInsteadOfVal(var assigned only once should be a val, Tags:(var,val))
[info] Fix Rules:
[info] ListToSet(defining List.toSet is defining a Set, Tags:(list,set))
[info] ListToSetBool(List to Set evaluated to Boolean, Tags:(list,set,type))
```

Figure 12: obey-list command

```
> obey-list -all
[info] List of Rules available:
[info] VarInsteadOfVal(var assigned only once should be val, Tags:(var,val))
[info] ListToSet(defining List.toSet is defining a Set, Tags:(list,set))
[info] ListToSetBool(List to Set evaluated to Boolean, Tags:(list,set,type))
```

Figure 13: obey-list with -all option

3. Implementation & Internal Structure

This section will present how the Obey plugin is implemented. There are two main components: the plugin & the sbt-plugin.

3.1. The Plugin

The Obey Scala compiler plugin is responsible for traversing the scala.meta ASTs generated from the source code and applying the selected rules on them. As a compiler plugin, it runs after the *typer* phase, and right after the Scalahost *convert* phase that generates the scala.meta ASTs. The compiler plugin must perform three important tasks: parse user options, filter the rules & apply them on the scala.meta trees.

3.1.1. Parsing User Options

A *Loader* class is responsible for loading user defined rules according to the *addRules* option. We rely on Scala reflection to load the rules. All loaded rules are then added to the set of rules available to the user and displayed with the *obey-list* command.

In order to parse the user filtering options, we implemented a dedicated small parser combinator, called *SetParser,* that accepts any valid OFL expression as described in *2.2*. The main advantage of defining a small specialized parser is flexibility. In fact, the grammar can be modified and extended without requiring any modification in the rest of the code. We can further reuse this parser in various places where it might be needed.

The implementation of a parser enables to easily control the expressiveness of the option language and to sanitize the user's option in order to protect the plugin. Any failure to parse the user input will stop the execution, hence preventing the source code from being altered in a way that is not intended. Furthermore, since filters are translated into *java.util.matching.regex.Pattern*, we wanted to limit the expressiveness of the option language by allowing only alpha-numeric characters and the wildcard one, i.e., '*', . This choice was meant to simplify the use of the plugin while preventing unintended matches.

The parser enables to select traditional regular expression features and limit their use. For the moment, we only implemented support for the wildcard symbol, i.e., '*' that is a strict equivalent to the Scala regular expression '.*'. Adding support for other regular expressions, such as '*\d*' or the OR semantic, e.g., '*[ab]*' meaning either a or b, could be done easily.

The *SetParser* generates a tuple of *Set[Tag]*, one for the positive set and another for the negative one. This tuple is then transformed into a *TagFilter*, with Figure 14 signature, in which all the tags previously generated are transformed into a Pattern. This last transformation allows us to easily look for matches between the tag filters and the ones used to annotate the rules.

```
case class TagFilter(pos: Set[Pattern], neg: Set[Pattern])
```

Figure 14: TagFilter Interface

Finally, we also implemented alternative parsers, based on different option language grammars, that could replace the *SetParser* one. For example, *OptParser* accepts any input that respects the following regular expression:

```
([+-] ~ *.? ~ ident ~ *.?)*
```

Figure 15: Alternative Grammar

Figure 16 presents the equivalent of the OFL expression described in Figure 8.

```
+List +Set +VarInsteadOfVal -Dotty
```

Figure 16: Example for the Alternative Grammar

This grammar is not order sensitive and is more restrictive than the one defined in *SetParser*. In fact, wildcards can only be used at the beginning and/or at the end of an identifier, while *SetParser* accepts wildcards anywhere. This grammar was judged confusing as the minus operator is commutative in this language and hence was replaced with the *SetParser* one.

3.1.2 Filtering the rules

The rules are filtered according to the user specified options. As described above, user options generate a *TagFilter* which contains a positive and a negative set. Using reflection, we extract tags and names from the available rules. The filtering is made according to the following boolean expression:

```
Let T be the set of tags and the name for a rule.
Let P be the set of positive tags used in the filters.
Let N be the set of negative tags used in the filters.

We add the rule if and only if:
(P is Empty OR P ∩ T  not Empty ) And ( N ∩ T is Empty )
```

Figure 17: Filtering Expression

This formula implies that if both the negative and positive sets are empty, the default behavior will be to use all the available rules.

The intersection operator corresponds to a match between the regular expression defined in the positive/negative set and one of the tags or the rule's name. Furthermore, we decided to give a higher importance to the negative set for safety reasons, that is, if a rule matches both the positive and the negative set, it will not be used. This prevents the source code from being modified if the filters were applied to select formatting rules.

3.1.3 Applying the rules

When applying the rules, we ensure that the set of rules selected for formating and the ones selected for reporting do not intersect. This enables us to reduce the work performed to the strict minimum and use the operators defined by the TQL library for efficient traversals of the tree, as explained in the next paragraphs.

All reporting rules are combined using the '+>' operator. This operator discards the tree in the result and passes the generated warnings to the next rule. Using this operator, we enable the fusion of all reporting rules into one single rule. The fusion results in a single traversal of the tree which is obviously a lot more efficient than one traversal per rule.

All formating rules are combined using the '+' operator, which feeds the result of one traversal into

the next one, i.e., it aggregates the warnings and feeds the tree produced by one rule to the next. As a result, the order in which we apply the rules might influence the source code generated by the formating rules. For the moment, considering the restricted set of rules we are using and the little experience we have with possible ordering heuristics that could be used, we do not provide any guarantee on the execution order. As a safety measure, we keep a copy of the original source code with a "*.old*" extension.

3.2. The SBT Plugin

The obey-sbtplugin is implemented as an *sbt.AutoPlugin*. We already presented the interface and the default settings in section *2.3*.

The commands *obey-fix* & *obey-check* are implemented as *sbt.Command* that accept an arbitrary number of arguments (including none). The commands will remove the settings for their counter part, e.g., *obey-check* will set *obeyFix* to '- -', which means that no rule will be used to format the code. If no argument is provided, the filters specified in the project settings will be used. If one or several arguments are provided, we remove white spaces and concatenate all of them into a single string that would then be passed to the *SetParser* as explained in 3.1.1, therefore temporarily overwriting the project settings. For example, the following inputs are equivalent:

```
obey-fix +{List,Var,Set}-{Val}
obey-fix + {List, Val, Set} - {Val}
```

Figure 18: Example for obey-fix

This enables the users to insert white spaces in the filtering language at their convenience, making it easier to write and read OFL expressions.

4. Experimental Evaluation

In this section, we will present the Dotty auto-migration, an interesting use of the Obey framework. Afterwards, we will present a small benchmark that enabled us to measure the overhead produced by our plugin.

4.1. Dotty Migration

As a useful test case for Obey, we implemented rules corresponding to the rewriting performed in the *srewrite* project[12]. Those rules enable to format Scala code into Dotty[8] compatible code. As stated in *srewrite*, other transformations might appear to be needed in the future. With Obey, new transformations could easily be added.

We created a small project, called *IObey*[9] to test our plugin. Using the sbt-plugin, we only need to run the command *obey-fix* with the tag filter *+{Dotty}* to format the code and be able to compile it with the Dotty compiler.

4.2. Performance and Compilation Overhead

We performed a simple benchmark in order to measure the overhead introduced by our plugin. The results displayed in this section are an average computed on 10 runs. More advanced tests should be performed to evaluate the exact overhead of using the compiler plugin. One should refer to the TQL paper to learn more about the traverser's performances.

The overhead introduced by the dependencies, i.e., resolving the plugin and sbt-plugin dependencies, are present in both runs, that is, with and without the compiler plugin being ran. We assume that these overheads are constant and can therefore be ignored when computing the ratio between the two executions. Since resolving the dependencies depends on many external factors, such as the network connection, we do not provide any estimate for it.

The compilation time is measured as the time needed to execute the *sbt compile* command for a small project containing 8 Scala files in its core project, and 3 user-defined rules. The core project has approximately 500 lines of code. The user-defined rules are simple matches on variable and method constructs. The number of matches for the various rules and the execution time are presented in Figure 19.

In average, we see that the compilation with our plugin increases the compilation time by 12.2%. The overhead includes the execution of the sbt-plugin, the compiler plugin, the parsing of user options and the run of the plugin itself, that is, the rules execution. The observed overhead is however subjected to the complexity of the rules used by the plugin and the percentage given above might increase drastically with very complex and long rules.

Compilation of the project without the compiler plugin:

real 0m30.999s

user 0m53.136s

sys 0m0.644s

Compilation of the project with 6 rules enabled and 13 matches, no fixing:

real 0m33.754s

user 0m59.344s

sys 0m0.692s

Compilation of the project with 6 rules enabled and 13 matches, fixing:

real 0m34.308s

user 1m1.352s

sys 0m0.756s

Figure 19: Benchmark Results

5. Comparison with Abide

In this section, we will present Abide, a Lint-like tool developed at TypeSafe, and compare it with Obey. Abide was our original inspiration for the Obey project which, as a result, has a lot of similarities in terms of models and interfaces.

Abide is a Scala compiler plugin that defines and applies rules in order to detect and report wrong constructs in Scala software projects. As a compiler plugin, Abide runs after the *typer* phase.

The Rule model defined in Abide is more complex than the one we used in Obey. The Rule interface is defined as a trait and contains the elements presented in Figure 20.

All of these elements are absent in Obey, where only two attributes need to be implemented: the *description*, for display purposes, and the *apply* method.

Abide extends the *Rule* trait with different implementations in order to set up a hierarchy of rules. The *TraversalRule* represents a rule that needs a single traversal of the AST. The *TraversalRule*

further extends the OptimizingTraversal, which enables to fuse different traversals together. There are different types of rules such as: *ExistentialRule*, *WarningRule*, *PathRule*, *ScopingRule*. Each of them corresponds to a specific type of work performed on the trees, e.g., the *WarningRule* looks for potentially bad constructs and reports them, while the *ExistentialRule* handles the case where one construct might invalidate several warnings. None of these categories enables to correct the source code.

In Obey, we decided to rely on external technologies to simplify the user interface. The scala.meta project is a meta-programming API designed to be simple, robust, portable and replace the *scala.reflect* usage. Even though the project is still in an early stage, it enabled us to greatly simplify the extraction of information that we needed from the trees. Trees are self-contained, which facilitates the formatting actions needed in the rules.

The TQL library enabled to abstract from the traversal of trees and its optimization. Abide had to define its own traverser and fusing mechanisms. In our case, the TQL library provides a black box that implements various traversals and operators to combine them in different ways. Furthermore, the library enables to modify the trees during the traversal by the use of the *transform* combinator.

```
trait Rule {
  /* A Context is a wrapper for a symbolTable. It enables to share
informations between different rules.*/
  val context: Context
  /* An analyzer can be build based on a compiler and a set of rules*/
  val analyzer: AnalyzerGenerator
  /* The state holds the warnings and different information generated by the
rules*/
  type State <: RuleState
  type Warning <: RuleWarning
  /* A name is used for display purposes*/
  val name: String
}
```

Figure 20: The Abide Rule interface

6. Future Work

Obey is a flexible and extensible framework. Our implementation uncovered several interesting use cases. In this section, we will present the potential extensions and modifications that could be implemented in order to improve our framework.

6.1. Implementing Additional Rules

The Obey set of provided rules is small for the moment. The goal of this semester project was to implement the framework itself and hence little time was dedicated to the implementation of the rule library. As a result, we plan to extend the set of rules available directly in the framework by, for example, translating all the Abide rules into their Obey equivalent. Implementing scalac compiler warnings as Obey rules might also be a suitable option.

Moreover, we would like to implement migration rules for specific frameworks. For example, migrating from Play 2.2 to Play 2.3[10] requires to perform several modifications to the source

code. Using the Obey framework, such migration could be automated. The auto correction of deprecated features in collection libraries would also be an interesting extension.

6.2. Rules distribution

The Obey plugin could benefit from a more flexible distribution of rules. For the moment, all the provided rules are implemented inside the model project. An elegant alternative would be to package rules separately and publish them. This solution decouples the rules implementation from the framework itself, hence allowing to add new rules without requiring to update the entire plugin. Furthermore, we could create content based packages, e.g., Scala compiler warnings rules or Dotty migration rules. The user would then specify only the dependencies that are interesting to him, which should reduce downloading and running overheads. Finally, this would also allow different users to publish their own packages and make them accessible to other programmers. For example, the migration between two versions of a framework such as Play could be automated by publishing a package of migration rules.

6.3. Tags hierarchy & Filtering

For the moment, rule filtering is based on tags and names, which are represented as a set of strings. Defining a tree hierarchy for tags would facilitate the use of the plugin, i.e., the user would be able to identify the tags he needs and would hence easily choose the correct filtering expression. Such a solution could be implemented as the naming system described by Lampson in[11] and would for example result in the following tree:

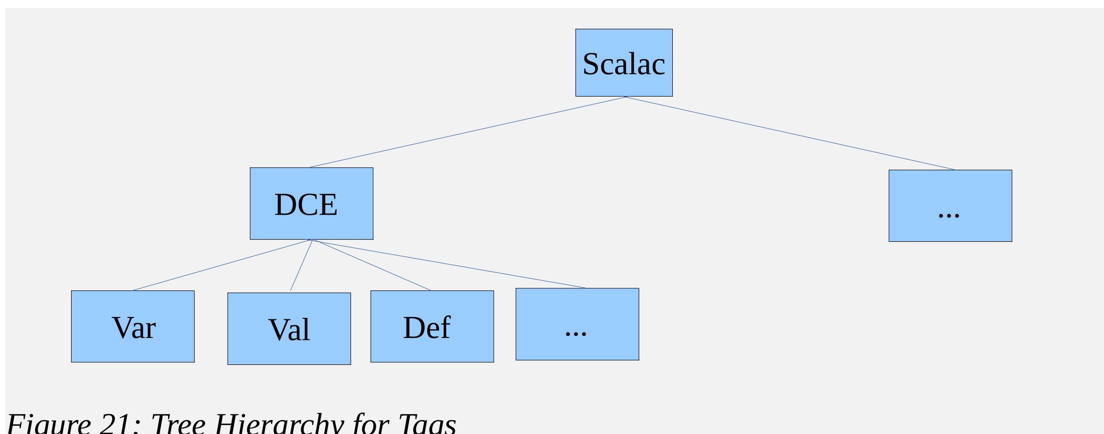

Figure 21: Tree Hierarchy for Tags

The filter option '+*{Scalac.DCE}*', for example, would return the entire subtree rooted at '*DCE*', and correspond to the following expression '+*{Var, Val, Def, ...}*'. A rule would still have a set of tags, but those would correspond to nodes in this hierarchy tree. If we combine this extension with the distribution one described above, different packages would be considered as different trees.

6.4 Rules Dependencies & Ordering

As explained previously, the result obtained from applying rules is order sensitive. Our solution for the moment does not ensure that the resulting code is the best possible solution. Requiring that all rules are commutative is not feasible, since we allow users to define their own rules. As a result, we have to rely on some heuristics to sort them.

One possibility would be to use the tag filtering described above, e.g., user defined rules should always be applied last and rules corresponding to leaves first.

Another solution would be to define new operators in the OFL language, e.g., '\+' and '\-', that

would be used to specify that rules should be applied according to the order in which the tags are listed.

```
\+{List, Var, ListToSet}
```

Figure 22: OFL Ordered expression

In Figure 22, all the rules that have the *List* tag will be applied first, then the ones with the *Var* tag, and finally the *ListToSet* one. If the rule's name matches one of the tags, this tag position is used to order the rule. Otherwise, we insert it according to the first matching tag position.

This solution seems easier to implement than the one described before, but requires a deeper understanding of what each rule does and the set of tags that it has in order to predict the execution order.

6.5. Automatic Translation of Abide rules

Abide and Obey have similar goals and structures. The choice we made to perform our traversals and modifications on scala.meta trees prevents us from directly importing Abide rules in Obey. As a future work, we envision to implement a transformer that would extract cases in Abide rules and translate them into match cases on scala.meta trees. While this might enable to translate rules from Abide and use them in Obey, we would still have to implement the formating part for each rule if we want to automatically correct all the reported warnings.

7. Conclusion

Our main objective was to create a code health framework for the Scala programming language that gathers all the good aspects of already existing linter tools for Scala, that is: flexibility, reliability, ease of use and extensibility. We also wanted to show that, by using external high-level technologies, we would be able to simplify its interface and create a powerful code-correcting linter tool. During this optional semester project, we reached all of our goals with Obey's implementation. As explained in this paper, we believe that Obey has an important potential that could help developers in various domains by automating migration and, more generally, automate code-updates as well as enforcing code health requirements.

REFERENCES

[1] TQL library project: https://github.com/begeric/TQL-scalameta

[2] scala.meta project: http://scalameta.org/

[3] Stephen Johnson, "*Lint, a C program checker*", Computer Science technical Report 65, Bell Laboratories, December 1977

[4] clang-tidy: http://clang.llvm.org/extra/clang-tidy.html

[5] Leif Wickland, "*Toward a Safer Scala – Detect errors earlier with static analysis*", PNWScala 2014

[6] The Abide project: https://github.com/scala/scala-abide

[7] The WartRemover project: https://github.com/puffnfresh/wartremover

[8] Dotty project: https://github.com/lampepfl/dotty

[9] IObey project: https://github.com/aghosn/IObey

[10] Play Framework: https://www.playframework.com/

[11] Butler W. Lampson, "Designing a Global Name Service", Proc. 4Th ACM Symposium on Principles of Distributed Computing,pp 1- 10, 1986

[12] Samuel Gruetter, *srewrite* project: https://github.com/samuelgruetter/srewrite