École Polytechnique Fédérale de Lausanne

Scala 3 syntax rewriting

by Mark Tropin

Master Research Project Report

Approved by the Examining Committee:

Prof. Martin Odersky
Project Advisor

Adrien Piquerez
Project Supervisor

EPFL IC SCALA
INR 328 (Bâtiment INR), Station 14
1015 Lausanne
Switzerland

January 29, 2024

Simplicity does not precede complexity, but follows it.
— A. Perlis

Dedicated, in respect and admiration, to the spirit that lives in the computer.

Acknowledgments

I would first like to thank the people who helped me discover the Scala Center: Antonella Veltro and Eileen Hazboun.

I am grateful to Darja Jovanovic, Anatolii Kmetiuk and Sébastien Doeraene for their swift communication and cooperation that helped me choose the topic of this research project.

Special thanks go to my supervisor Adrien Piquerez. Thank you for your guidance, helpful advice, patience and incredible attention to detail.

Additional thanks go to Bauch Saleh at the University of Geneva who helped with the preprocessing of the Scala Center logo seen on the first page.

Lausanne, January 29, 2024 Mark Tropin

Abstract

We present syntax rewriting rules that translate Scala 2 code into Scala 3.

Two major syntactic changes are introduced: new control structure syntax and significant indentation.

We describe the design and the implementation of these rules and evaluate their performance on a large Scala project.

Finally, we discuss strategies regarding how this project can be extended in further iterations.

Résumé

On prèsente deux règles de réécriture syntaxique qui traduisent un code Scala 2 en Scala 3.

Deux changements syntaxiques sont introduits : une nouvelle syntaxe pour les structures de contrôle et l'indentation significative.

On décrit la conception et l'implémentation de ces règles et on évalue leur performance sur un grand project Scala.

Enfin, on propose des strategies d'extension pour les prochaines versions de ce projet.

Contents

Chapter 1

Problem definition

Scala [1] is a modern multi-paradigm programming language designed to write elegant and type-safe code.

The two main versions of Scala that are currently in use are Scala 2.13 and Scala 3.3. Scala 3 aims to provide a lighter, more readable syntax via introducing two major syntactic changes:

- **Optional parentheses in control structures.** The traditional C-like syntax where the condition or generator of a loop is delimited with a pair of parentheses (see B.1) is now delimited with a pair of keywords (like if-then, for-do; see B.2).
- **Braceless syntax.** Using curly braces for a block expression or a template[1] definition is now optional: a block of statements will be treated as such as long as it is properly indented.

A large portion of Scala developers still use Scala 2.13 but would like to update their codebases to Scala 3. What are the options for transitioning to Scala 3?

- The Scala 3 compiler, also known as Dotty [2], features an optional parameter that lets its users try out the new syntax. However:
  - It does not provide any configuration. The user has no control over the application of the rules.
  - The code has to be validated by the Scala 3 compiler (it has to pass the type checks and successfully compile). This means that codebases that heavily rely on deprecated Scala 2 features will not be able to use this option without significant refactoring.
  - Since this is a supplementary feature of the Scala 3 compiler, there are many bugs that require fixing.

[1] A *template* is the definition (body) of a class, object, trait or package object.

- Scalafmt [3] is Scala's code formatting tool. While it is capable of rewriting Scala sources to a specified code style, it has two major drawbacks:
  - Scalafmt reformats the whole project without giving the user the option to modify specific files. The user is unable to test the rewriting on a few source files before deciding whether the rules should be applied to the whole codebase.
  - The Scalafmt rewriting rules are not *reversible*. The user cannot revert the changes introduced by the rules easily, which does not let the user to test the new syntax rules before applying them to their project.

1.1 Project criteria

The goal of this project is to offer a solution that implements these syntactic rules in a way that is:

- Reversible.
- Configurable.
- Transparent (the rewriting rules should clearly explain the change that is introduced).
- Minimal (the rewriting should preserve the coding style of the original).
- Lightweight and efficient.

Additionally, we hope that this project will eliminate the need of having rewrite rules in the compiler itself. This will establish a separation of concerns (syntax rewriting should be handled by code formatting tools) and make the compiler code more maintainable.

Chapter 2

Solution

2.1 Scalafix

In order to overcome the issues that were mentioned in the previous chapter, we have decided to implement these refactoring rules in Scalafix.

Scalafix [4] is a refactoring and linting tool for Scala projects. It allows its users to refactor a given codebase by configuring a predefined rule and applying it to the code.

One obvious benefit to the end user is the configuration: all rules come with default parameters that can be tweaked in order to match the requirements of the end user. This option is not available for the rewriting rules in the compiler, which transform the codebase in a *uniform* manner.

On the developer's side, the rules in Scalafix offer multiple levels of *granularity*: a given fragment of code can be viewed as a sequence of tokens or as a syntax tree that presents the nested structure of the code. This allows us to develop rules that only introduce changes where necessary, which is a key metric for evaluating the quality of any refactoring tool.

Thus, we are able to fix the syntactic structure of the code to conform to a new grammar without removing the idiosyncratic properties of the original (e.g., if the user used more indentation than necessary or added multiple nested parentheses where they were not required, these aspects will be preserved as much as possible).

Furthermore, the lightweight nature of rules in Scalafix allows the user to experiment with the new Scala 3 syntax to see if it fits their coding style. For instance, if the user decides to try out the new syntax for control structures and sees that they prefer the original syntax with parentheses, they only have to apply the inverse rule to convert their codebase to its previous state. No other changes to the program's syntax are introduced.

2.2 Syntax rewriting rules

We provide the user with four syntactic rules which allow them to explore the new Scala 3 syntax and refactor their codebases accordingly. These rules are named:

- `Scala3ControlSyntax`
- `Scala2ControlSyntax`
- `IndentationSyntax`
- `AddEndMarkers`

This chapter presents the configuration parameters and typical use cases for each rule.

2.3 Rule 1: Scala 3 Control Syntax

The first rule allows the user to benefit from the new "quiet" syntax for control structures used in Scala 3. The main syntactic difference between the Scala 2 and Scala 3 syntaxes can be summarized as follows:

- Enclosing parentheses in control structures (`if, while, try/catch, ...`) can be removed.
- Instead, the condition/loop generator has to be delimited with a pair of keywords. Therefore, `if (cond)` becomes `if cond then`.

These syntactic changes apply to `if` expressions, `while` loops, `for` loops and `for` expressions, as well as `try` blocks.

In the last case, there is no expression that previously required parentheses; in that case, the syntactic difference is in the `catch` block. The `catch` clause can appear at the same level of indentation as the `try` clause, given that there is only one case (previously, this required significant indentation or a brace-delimited block).

Our implementation provides one configuration parameter which makes this last case optional: for more readability and consistency, many developers prefer to indent their `catch` clauses or add braces to remove any ambiguity. Therefore, this syntactic transformation is disabled by default and can be enabled by the user if needed.

2.4 Rule 2: Scala 2 Control Syntax

Since many developers prefer the parenthesized C-like syntax of control structures seen in Scala 2, our implementation provides a new rule that serves as an inverse transformation of the previous one.

If the user does not find the new syntax suitable for their needs, they can apply the rule `Scala2ControlSyntax` to revert the code back to its previous state. The following transformations are applied:

- The conditions or generators of all control structures are enclosed in a pair of parentheses.
- The new additional keywords are removed (e.g., `while true do` becomes `while (true)`).

As for the previous rule, the "catch inlining" feature remains optional. It is disabled by default.

2.5 Rule 3: Significant Indentation

The third rule implements the *significant indentation* syntax found in Scala 3.

In Scala 2, all templates (body of a trait, class or object) and blocks of statements have to be delimited by a pair of curly braces. The indentation levels between lines of code are not significant, as can be seen in Listing 2.1.

In Scala 3, these braces can be removed, provided that the template/block is properly indented, i.e. the indentation level of each subsequent line is non-decreasing (see Listing 2.2).

```
if (true) {
    println("a")
  println("b")
   println("c")
}
```

Listing 2.1: An (incorrectly indented) block with braces.

```
if (true)
    println("a")
        println("b")
        println("c")
```

Listing 2.2: A (correctly indented) block without braces.

The rule provides a variety of parameters that can be adjusted to fit the needs of its users. The first parameter enables/disables *end markers*, which are a new feature added to Scala 3 to make large indented blocks more readable and well-defined (visually). The following two parameters specify the way end markers are applied: the user can enumerate the list of structures where they do not want to add end markers (e.g. end markers can be applied to all indented structures except for classes

and traits) as well as set a minimal *line length* of a structure that would require an end marker[1]. Finally, the user can define the default indentation that they would like to see for an incorrectly indented code block: this has to be provided to the rule as a string and is preset to 2 spaces, the default indentation level used in Metals.

2.6 Rule 4: Adding End Markers

Scala 3 has introduced *end markers*, which are used to indicate the end of a large definition spanning multiple lines. Its most prominent use case is for delimiting braceless blocks and templates which can be hard to read in a sizable code base. For an example, see Listing 2.3.

```
object EndMarker:
  enum Color:
    case Red, Green, Blue
  end Color
end EndMarker
```

Listing 2.3: An example of end marker usage.

The configuration parameters are exactly the same as for the previous rule, since the use of end markers is not commonly seen as standard practice and should be adjusted to the requirements of individual users/projects.

[1] In the current version of this rule, end markers are disabled due to bugs explained in Section 5.1.

Chapter 3

Implementation

We provide the user with four syntactic rules which allow them to explore the new Scala 3 syntax and refactor their codebases accordingly. These rules are named:

- `Scala3ControlSyntax`
- `Scala2ControlSyntax`
- `IndentationSyntax`
- `AddEndMarkers`

This chapter presents the implementation details for each rule.

3.1 Rule 1: Scala 3 Control Syntax

Implementation-wise, the rule locates the necessary tokens (the parentheses, if they exist), removes them and adds the new token (keyword, if it does not exist).

For the last case, if the rule can be applied (if there is only one `catch` clause), the unnecessary whitespace (spaces, tabs and newline characters) between the end of the `try` block and the beginning of the `catch` clause is removed. We also remove the curly braces if they were found.

3.2 Rule 2: Scala 2 Control Syntax

The implementation mirrors the implementation of `Scala3ControlSyntax`. We make use of the tree-like representation of control structures of Scalafix, which enables us to locate the condition/-generator part of a control structure, remove the additional keyword(s) and add parentheses.

To reverse catch inlining, we locate the `catch` clause of the `try/catch` block, insert a line break if necessary and indent it to the right to match the indentation level of the `try` clause.

3.3 Rule 3: Significant Indentation

The rule is implemented as follows:

- First, the input text file is converted to a sequence of lines, which are split into lists of tokens ending with the line break character. The leading indentation (at the start of each line) is separated from the list of non-whitespace tokens.
- This representation of indents is saved twice as a *mutable* and *immutable* sequence of indentation tokens.
- For every brace-containing structure that can be converted to the significant indentation syntax:
  - The pre-conditions for applying the rules are checked. The code describes many edge cases where the rule should not be applied (for instance, empty blocks or method argument clauses should not have their braces removed).
  - The indentation level of the current line is compared with the indentation of the parent structure. If the current indentation is incorrect (less than or equal to the correct indentation of the parent), the default indentation is added to the start of all lines of the current subtree. Alternatively, the correct indentation is set to the original indentation.
  - The mutated (fixed) indentation is compared to the original indentation used in the input file. If they coincide, no patch is applied (the lines are already correctly indented).
  - After the indentation has been fixed, the curly braces can be safely removed.
  - If the user opted for adding end markers, they are applied where necessary.

In the following subsections, two implementation peculiarities which deviate from standard practice of Scalafix rules and Scala code in general are described.

3.3.1 Mutable and immutable representations

As a rule, Scala code generally operates on immutable values to conform to standard Functional Programming practices. Why did we decide to represent the indentations with a mutable and immutable structure at the same time?

The reason for this approach is twofold. First, as specified in the Solution chapter, we wanted to implement the rules in a way that introduces the *smallest amount of changes* to the input code. Our rule does not aim to reformat the code according to a certain coding style: there already exist solutions for that in Scalafix. Our goal is to preserve the stylistic preferences of the user while generating code that compiles correctly and works as intended. Second, it allowed the implementation to be more readable and clear for future maintainers and everyone who is interested in how the code actually works. The major source of complexity for this rule was the indentation of *nested structures*: while fixing the indentation of a parent block we have to make sure that the indentation of inner blocks is also valid. At the same time, we cannot simply add more indentation tokens to every nesting level, which would make the resulting code unreadable. Therefore, in order to keep track of all changes, we use a mutable sequence of indentation tokens. It is important to note that this is the only place where mutability is introduced: this ensures that the code is semantically transparent and fairly modular, which increases comprehensibility and makes the implementation/debugging more straightforward.

3.3.2 Custom type for representing indentation: `RLEIndent`

In order to represent the leading indentation of every line of code, we have designed a custom class that stores the input string as a run-length encoded sequence of indentation characters (spaces and tabs). Apart from the usual conversion helper methods (reading the indentation from a string, converting an `RLEIndent` to a string), this class provides comparison and equality operators which make the implementation of the rule more readable.

As a side note, this feature proved to be extremely helpful during the later stages of debugging. This encoded representation of indentation is similar to what can be seen in the errors returned by the compiler, and it allows for easy comparison and inspection of indentation. For instance, `List((3,Space),(1,Tab),(1,Space))` is more readable than `"  \t "`.

3.4 Rule 4: Adding End Markers

The rule is implemented as follows:

- For every subtree requiring an end marker:
  - The first token of the subtree is located. The end marker will be put at the same level of indentation.
  - The end marker name is computed. In general, for control structures and anonymous entities the name mirrors the type of the subtree (e.g. `end while`, `end extension`). For named definitions/declarations, the end marker name matches the `name` parameter of the subtree (`end MyClass`).
  - The end marker name is inserted after the last token of the given subtree at the correct indentation level on a new line.

Contrary to the previous rule, this rule applies end markers to all structures where an end marker can be placed (whether it uses braces or not). Therefore, the user should apply only one of these rules depending on their particular needs.

Chapter 4

Evaluation of results

In order to evaluate the correctness of these rules, we have employed a wide variety of tests.

Most of these tests were written by hand; we attempted to cover standard use cases that commonly arise in Scala code, as well as addressing rarer edge cases which might reflect unusual coding styles. The focus has always been on retaining the same code structure while ensuring the syntactic correctness of the code, see Listings 4.1 and 4.2. On top of that, running the first two rules on a small Scala 3 project returned the exact same input code, which proves the *reversible* nature of these rules.

```
  for (x <- xs) yield {
    val myFactor = 2
                        myFactor * x
  }

  if (true) {
      if (true) {
println("true")
    }
  }
```

Listing 4.1: Before applying the rules `IndentationSyntax` and `Scala3ControlSyntax`.

```
  for x <- xs yield
    val myFactor = 2
                        myFactor * x

  if true then
      if true then
        println("true")
```

Listing 4.2: After applying the rules `IndentationSyntax` and `Scala3ControlSyntax`.

For the two final rules (indentation syntax and end markers) the tests were primarily based on tests from the Scala compiler, as well as the Scala 3 documentation.

Finally, the largest "production code" test that confirmed the correctness of our code was running the syntax rules on the code of the Scala 3 compiler itself. The code that was produced compiled without errors. After applying the `IndentationSyntax` rule to the Dotty commit `59eddae77e`, 454 files changed, with 6140 lines inserted and 11707 lines removed.

Chapter 5

Further work

Due to time constraints, we had to prioritize and design the rules to reflect the most prominent differences of Scala 2 and Scala 3 syntax. This section provides a list of further improvements that can be applied to extend this project.

5.1 Remaining bugs

The main bug that lead us to disable the end markers in the `IndentationSyntax` rule can be seen in Listings 5.1 and 5.2 (adding end markers in `IndentationSyntax` and `AddEndMarkers` is identical).

```
1  val x = 1
2  val y = x match
3    case 1 => "1"
```

Listing 5.1: Before applying the rule `AddEndMarkers`.

```
1  val x = 1
2  val y = x match
3    case 1 => "1"
4  end match
5  end y
```

Listing 5.2: After applying the rule `AddEndMarkers`.

It is clear that only one of these end markers is valid: the `match` expression does not start on a new line, so the end marker should not be applied. Furthermore, in the case of nested structures the user probably wants only one end marker that would terminate the outermost syntax tree. If we were to reimplement this rule to avoid producing multiple end markers in a row, we would have to traverse the parent trees of the current syntax tree to locate the outermost structure that we want to terminate (the fact that Scalameta [5], the parser used by Scalafix, does not treat end markers as parts of a given tree but rather lists them as a separate entity in the sequence of statements complicates the matter). Given that end markers are not yet commonplace in Scala code, we decided to fix more pressing bugs and leave this for further work.

5.2 Refactoring to match Scalafix predefined functions

The code quality can be noticeably improved by rewriting the Scalafix patches using more idiomatic Scalafix methods. For instance, matching pairs of parentheses and braces are currently found manually, but there exist methods for that in Scalafix. These minor improvements can make the code more readable and maintainable.

5.3 More configuration on rules

We attempted to make the rules fit the demands of most users, but the rules can be modified to be configured even further. For instance, the default indentation that is currently provided by the user as a string can be parsed from a compact textual representation (e.g., `2s` or `1t` for 2 spaces or 1 tab). It can also be inferred from a given code sample. The end marker rule configuration can also be extended to ask the user how to treat nested structures (see Listings 5.1 and 5.2). The first two rules (involving control structures) do not require any further configuration, since the two syntaxes are unambiguously distinct and well-defined (see control structures in Appendix A).

5.4 More tests

To consolidate the correctness of the code and provide a guarantee for further development, the test suite can be extended. The tests can cover more edge cases and model more idiomatic Scala 2 code (new tests can be based on older Scala code found in open source repositories).

5.5 Fewer braces

Scala 3.3 has introduced a new extension of the braceless syntax called *fewer braces* [6]. The idea is that braceless syntax used for block expressions and templates can now be applied to function arguments where braces were previously required (see Listing 5.3).

```
xs.map:
  x =>
    val y = x - 1
    y * y
```

Listing 5.3: An example of the "fewer braces" syntax.

Since this feature is very recent and not commonly seen in Scala sources, we decided not to implement it and focus on templates and blocks instead. Future iterations of the current project can

work on treating this new syntax as well.

5.6 Optics for converting custom types

The custom type `RLEIndent` explained in Section 3.3.2 currently requires explicit conversion from/to a string representation of indentation. The *Monocle* library [7] can be used to abstract this conversion. In particular, we can make use of the `Iso` optic since there is a clear isomorphism between compressed and uncompressed whitespace representations.

5.7 Fully reversible rules

In Chapter 4 we briefly mentioned the fact that the first two rules are *reversible*: the second rule serves as the inverse of the first. This idea can be extended to the remaining rules: a new rule can be added for reverting a given code file to the syntax with braces, adding a pair of braces around every indented template or block of statements. Additionally, another rule can be used to remove end markers.

Chapter 6

Conclusion

This project report has presented a novel way of formatting a Scala 2 codebase to comply with the new Scala 3 syntax.

The project criteria identified in Section 1.1 have been fulfilled:

- Our solution allows the user to experiment with the new syntax and revert the changes instantaneously.
- The user can configure the rules to their particular needs.
- The naming of the rules and their parameters clearly indicate their purpose.
- The rewriting rules only modify the necessary tokens: they do not remove or misplace other parts of the code, preserving the original coding style.
- The code runs in a reasonable time frame even for large projects such as the Scala 3 compiler.

Therefore, we believe that these rewriting rules can be used in existing Scala 2 projects and facilitate the ongoing transition to Scala 3.

Bibliography

[1] URL: https://www.scala-lang.org/.

[2] URL: https://dotty.epfl.ch/.

[3] URL: https://scalameta.org/scalafmt/.

[4] URL: https://scalacenter.github.io/scalafix/.

[5] URL: https://scalameta.org/.

[6] URL: https://docs.scala-lang.org/sips/fewer-braces.html.

[7] URL: https://www.optics.dev/Monocle/.

[8] URL: https://scala-lang.org/files/archive/spec/2.13/13-syntax-summary.html.

[9] URL: https://docs.scala-lang.org/scala3/reference/syntax.html.

[10] URL: https://docs.scala-lang.org/scala3/book/introduction.html.

Appendix A

Grammar

This appendix provides the context-free grammar of control structures and indented blocks, as seen in Scala 2 and Scala 3. This is a subset of the full EBNF grammar that can be found in the language specification. See [8] and [9].

A.1 Scala 2

Control structures (`if` statements) in Scala 2:

```
Expr1           ::=  [‘inline’] ‘if’ ‘(’ Expr ‘)’ {nl} Expr [[semi] ‘else’ Expr]
```

Templates in Scala 2:

```
ClassTemplate   ::=  [EarlyDefs] ClassParents [TemplateBody]
TemplateBody    ::=  [nl] ‘{’ [SelfType] TemplateStat {semi TemplateStat} ‘}’
```

Block expressions and case clauses in Scala 2:

```
BlockExpr       ::=  ‘{’ CaseClauses ‘}’
                  |  ‘{’ Block ‘}’
```

A.2 Scala 3

Control structures (`if` statements) in Scala 3:

```
Expr1           ::=  [‘inline’] ‘if’  Expr ‘then’ Expr [[semi] ‘else’ Expr]
```

Templates in Scala 3:

```
ClassDef        ::=  id ClassConstr [Template]
Template        ::=  InheritClauses [TemplateBody]
TemplateBody    ::=  :<<< [SelfType] TemplateStat {semi TemplateStat} >>>
```

Block expressions and case clauses in Scala 3:

```
BlockExpr       ::=  <<< (CaseClauses | Block) >>>
```

Indentation tokens in Scala 3:

```
colon           ::=  ’:’
<<< ts >>>      ::=  ‘{’ ts ‘}’
                  |  indent ts outdent
:<<< ts >>>     ::=  [nl] ‘{’ ts ‘}’
                  |  colon indent ts outdent
```

Appendix B

Code examples

This appendix provides code examples that illustrate the grammar rules found in Appendix A. These code examples are heavily based on [10].

Control structures (`if` statements):

```
if (x < 0) {
  println("negative")
} else if (x == 0) {
  println("zero")
} else {
  println("positive")
}
```

Listing B.1: Control structures (Scala 2).

```
if x < 0 then
  println("negative")
else if x == 0 then
  println("zero")
else
  println("positive")
```

Listing B.2: Control structures (Scala 3).

Template (trait) definitions:

```
trait AddService {
  def add(a: Int, b: Int)  = a + b
}
trait MultiplyService {
  def mult(a: Int, b: Int) = a * b
}
```

Listing B.3: Defining a template (trait) in Scala 2.

```
trait AddService:
  def add(a: Int, b: Int)  = a + b
trait MultiplyService:
  def mult(a: Int, b: Int) = a * b
```

Listing B.4: Defining a template (trait) in Scala 3.

Block expressions / case clauses:

```
val result = i match {
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
}
```

Listing B.5: Case clauses (Scala 2).

```
val result = i match
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
```

Listing B.6: Case clauses (Scala 3).