scalafmt: opinionated code formatter for Scala

Ólafur Páll Geirsson

School of Computer and Communication Sciences

Master's Thesis

June 2015

Responsible

Prof. Martin Odersky

EPFL / LAMP

Supervisor

Eugene Burmako

EPFL / LAMP

Abstract

Automatic code formatters bring many benefits to software development, yet they can be tricky to implement. This thesis addresses the problem of developing a code formatter for the Scala programming language that captures many popular coding styles. Our work has been limited to formatting Scala code. Still, we have developed algorithms and tools, which we believe can be of interest to developers of code formatters for other programming languages.

Contents

1 Introduction

The main motivation of this study is to bring scalafmt, a new Scala code formatter, to the Scala community. The goal is to capture many popular coding styles so that a wide part of the Scala community can enjoy the benefits that come with automatic code formatting.

Without code formatters, software developers are responsible for manipulating all syntactic trivia in their programs. What is syntactic trivia? Consider the Scala code snippets in listings 1 and 2.

Listing 1: Unformatted code

```
// Column 35                        |
object ScalafmtExample {
  function(arg1, arg2(arg3(
    "String literal", arg4, arg5),
    arg6 + arg7))
}
```

Listing 2: Formatted code

```
// Column 35                        |
object ScalafmtExample {
  function(
      arg1,
      arg2(arg3("String literal",
                arg4,
                arg5),
           arg6 + arg7))
}
```

Both snippets represent the same program. The only difference lies in their syntactic trivia, that is where spaces and line breaks are used. Although the whitespace does not alter the execution of the program, listing 2 is arguably easier to read, understand and maintain for the software developer. The promise of code formatters is to automatically convert any program that may contain style issues, such as in listing 1, into a readable and consistent looking program, such as in listing 2. Automatic code formatting offers several benefits.

Code formatting enables large-scale refactoring. Google used ClangFormat[11], a code formatter, to migrate legacy C++ code to the

modern C++11 standard[28]. ClangFormat was used to ensure that the refactored code adhered to Google's extensive C++ coding style[7]. Similar migrations can be expected in the near future for the Scala community once new dialects, such as Dotty[18], gain popularity.

Code formatting is valuable in collaborative coding environments. The Scala.js project[21] has over 40 contributors and the Scala.js coding style[22] contains over 2.600 words. Each contributor to the Scala.js project is expected to follow the coding style. Each contributed patch is manually verified against the coding style by the project maintainers. This adds a burden on both contributors and maintainers. Several maintainers of popular Scala libraries have expressed this sentiment. ENSIME[4] is a popular Scala interaction mode for text editor. Sam Halliday, a maintainer of ENSIME, says "I don't have time to talk about formatting in code reviews. I want the machine to do it so I can focus on the design."[8]. Akka[1] is another Scala library to build concurrent and distributed applications. Viktor Klang, a maintainer of Akka, suggests a better alternative "Code style should not be enforced by review, but by automate rewriting. Evolve the style using PRs against the rewriting config.".[12]. With code formatters, software develoeprs can direct their full focus on writing correct, maintainable and fast code.

1.1 Whitespace style issues

Mechanical style issues are source code issues that can be fixed automatically. For example, consider listing 3.

Listing 3: Style issues

```
1 if  ( condition(  predicate)) {
2     println("Goodbye" );
```

```
3     System.exit(1) }
```

This code snippet leaves a lot left to be desired. Mechanical issues include the redundant spaces around parentheses, unnecessary semicolon after the `println` and the inconsistent indentation in the body of the if statement. Non-mechanical issues may include the fact the program prints an error message to the standard output and exits the process, instead of throwing an exception.

This thesis only addresses the whitespace style issues. That is, to automatically fix the spaces and newline characters between the non-whitespace tokens in the original source code. We will leave it to the software developer to decide whether the unnecessary semicolon should remain in the source file or if the program should throw an exception.

1.2 Contributions

The main contribution presented in this thesis are the following:

- scalafmt, a code formatter for the Scala programming language. At the time of this writing, scalafmt has been available for 3 months, it has been installed over 5.000 times and is already in use by several open source Scala libraries. For details on how to install and use scalafmt, refer to the scalafmt online documentation[5].
- algorithms and data structures to implement line wrapping under a maximum column-width limit. This work is presented in section 3.
- tools to develop and test code formatters. This work is presented in section 4.

The scalafmt formatter itself may only be of direct interest to the Scala community. However, much work in this thesis is not specifically tied to Scala and we hope can inspire the design of code formatters for other programming languages.

2 Background

This chapter explains the necessary background to understand Scala and code formatting. More specifically, we motivate why Scala presents an interesting challenge for code formatters. We go into details on Scala's rich syntax and popular idioms that introduced unique challenges to the design of scalafmt. We follow up with a brief history on code formatters for both Scala as well as other programming languages. We will see that although code formatters have a long history, the last 5 years have brought a lot of research in optimization based formatters, which scalafmt's design takes inspiration from.

2.1 Scala the programming language

Scala[16] is a general purpose programming language that was first released in 2004. Scala combines features from object-oriented and functional programming paradigms, allowing maximum code reuse and extensibility.

Scala can run on multiple platforms. Most commonly, Scala programs compile to bytecode and run on the JVM. With the releases of Scala.js[22], JavaScript has recently become a popular target platform for Scala developers. Even more recently, the announcement of Scala Native[20] shows that LLVM and may become yet another viable platform for Scala developers.

Scala is a popular programming language. The Scala Center estimates that more than half a million developers are using Scala[15]. Large organizations such as Goldman Sachs, Twitter, IBM and Verizon run Scala code in production systems. The 2015 Stack Overflow Developer Survey shows that Scala is the 6th most loved technology and 4th best paying technology to

Listing 4: Higher order functions

```
1 def twice(f: Int => Int) = (x: Int) => f(f(x))
2 twice(_ + 2)(6)  // 10
```

work with[24]. The popularity of Apache Spark[2], a cluster computing framework for large-scala data processing, has made Scala a language of choice for many developers and scientists working in big data and machine learning.

Scala is a programming language with rich syntax and many idioms. The following chapters discuss in detail several prominent syntactic features and idioms of Scala. Most importantly, we highlight coding patterns that encourage developers to write larger statements instead of many small statements. In section 3, we explain why large statements introduce a challenge to code formatting.

2.1.1 Higher order functions

Higher order functions (HOFs) are a common concept in functional programming languages and mathematics. HOFs are functions that can take other functions as arguments and can return functions as return values. Languages that provide a convenient syntax to manipulate HOFs are said to make functions first-class citizens.

Functions are first-class citizens in Scala. Consider listing 4. The method `twice` takes an argument `f`, which is a function from an integer to an integer. The method returns a new function that will apply `f` twice to an integer argument. This small example takes advantage of several syntactic conveniences provided by Scala. For example, in line 2 the argument `_ + 3`

Listing 5: Higher order functions without syntactic sugar

```
1 def twice(f: Function[Int, Int]) =
2   new Function[Int, Int]() { def apply(x: Int) = f.apply(f.apply(x)) }
3 twice(new Function[Int, Int]() { def apply(x: Int) = x + 2 }).apply(6)  // 10
```

Listing 6: Manipulating immutable list

```
1 val input = List(1, 2, 3)
2 val output = input.map(_ + 1)    // List(2, 3, 4)
3                   .filter(_ > 2) // List(3, 4)
```

creates a new `Function[Int, Int]` object. The function call `f(x)` is in fact sugar for the method call `f.apply(x)` on a `Function[Int, Int]` instance. Listing 5 shows an equivalent program to listing 4 without syntactic sugar. Observe that what was expressed as a single statement in line 1 of listing 4 is expressed with multiple statements in lines 1 and 2 of listing 5.

2.1.2 Immutability

Functional programming encourages stateless functions which operate on immutable data structures and objects. An immutable object is an object that once initialized, cannot be modified. Immutability offers several benefits to software development in areas including concurrency and testing. Listing 6 shows an example of manipulating an immutable list. Note that each `map` and `filter` operation creates a new copy of the list with the modified contents. The original list remains unchanged. Listing 7 show the equivalent operation using a mutable list. Observe that what was listing 6 is a single statement while listing 7 is multiple statements.

Listing 7: Manipulating mutable list

```
val input = List(1, 2, 3)
val output = mutable.ListBuffer.empty[Int] // mutable list
input.foreach { elem =>
  if (elem + 1 > 2) { // filter
    output += elem + 1 // map
  }
}
output // ListBuffer(3, 4)
```

Listing 8: SBT project definition

```
lazy val core = project
  .settings(allSettings)
  .settings(
    moduleName := "scalafmt-core",
    libraryDependencies ++= Seq(
      "com.lihaoyi" %% "sourcecode" % "0.1.1",
      "org.scalameta" %% "scalameta" % Deps.scalameta))
```

2.1.3 SBT build configuration

SBT[19] is an interactive build tool used by many Scala projects. SBT configuration files are written in `*.sbt` or `*.scala` files using Scala syntax and semantics. Although SBT configuration files use plain Scala, they typically use coding patterns which are different from traditional Scala programs. Listing 8 is an example project definition in SBT. Observe that the project is defined as a single statement and makes extensive use of symbolic infix operators. Due to the nature of build configurations, argument lists to can becomes unwieldy long and a single project statement can span up to dozens or even hundreds of lines of code.

2.2 Code formatters

In this chapter, we look at a variety of code formatters that have been developed.

2.2.1 Natural language

The science of displaying aesthetically pleasing text predates as early as 1956[9]. The first efforts involved printing natural language text and finding the right places to insert carriage returns to break long lines. As soon as programming languages evolved, software developers began to apply the similar ideas to source code.

LaTeX

2.2.2 ALGOL 60

Scowen[23] developed SOAP in 1971, a code formatter for ALGOL 60. The main motivation for SOAP was to make it "easier for a programmer to examine and follow a program" as well as to maintain a consistent coding style. This motivation is still relevant in modern software development. SOAP did provide a line length limit. However, SOAP would fail execution if the provided line length turned out to be too small. With hardware from 1971, SOAP could format 600 lines of code per minute.

2.2.3 LISP

In 1973, Goldstein[6] explored code formatting algorithms for LISP[14] programs. LISP is a family of programming languages and is famous for it's

Listing 9: A LISP program

```
(defun factorial (n)
  (if (= n 0) 1
      (* n (factorial (- n 1)))))
```

parenthesized prefix notation. Listing 9 shows a program in LISP to calculate factorial numbers. The simple syntax, extensive use of parentheses and nested nature of LISP programs makes them an excellent ground to study code formatters.

Goldstein presented a *recursive re-predictor* algorithm in his paper. The recursive re-predictor algorithm runs a top-down traversal on the abstract syntax tree of a LISP program. While visiting each node, the algorithms tries to first obtain a *linear-format*, i.e. fit remaining body on a single line. If there isn't enough space on the line to fit a linear format, the algorithm falls back to *standard-format* where each argument is put on a separate line aligned by the first argument. Goldstein observes that this algorithm is practical despite the fact that its running time is exponential in the worst cases. Bill Gosper used the re-predictor algorithm to implement GRINDEF[3], one of the first code formatters for LISP. Goldstein's contributions extend beyond formatting algorithms. Firstly, in his paper he studies how to format comments. Secondly, he presents several different formatting layouts which can be configured by the users. Both of those concerns are relevant for modern code formatters.

2.2.4 Language agnostic

Derek C. Oppen pioneered the work on language agnostic code formatting in 1980[17]. Oppen presented an algorithm is not tied to particular

programming language. The algorithms runs in linear time to the size of the input program and is incredibly memory efficient. Users provide a preprocessor to integrate the algorithm with a particular programming language. Besides impressive performance results, Oppen claims that a key feature of the algorithms is its streaming nature. The algorithm prints formatted lines as soon as they are input instead of waiting until the entire input stream has been read. However, Oppen's algorithm shares a worrying limitation with SOAP: it cannot handle the case when the line length is insufficiently large.

Mark van der Brand presented a library could generate a formatters given context-free grammar[26]. Brand correctly identifies that the development of code formatters requires a lot of effort and a generic solution may be possible. Beyond the usual motivations for developing code formatters, Brand mentions that formatters "relieve documentation writers from typesetting programs by hand". Like Oppen's algorithm, this library requires the user to translate code from a particular programming language into the library's constructs. The library borrows the concept of a *box* from Knuth[13].

John Hughes extended on Oppen's work on language agnostic formatting in term of programming techniques[10]. Hughes presented a design of a *pretty-printing* library that leverages the functional programming paradigm. Hughes says his library is a pretty-printer and not code formatters since the library does not consider how to format existing source code. Instead, the library prints data structures directly from their values. Wadler[27] and Chitil[25] extend on Hughes's and Oppen's work. However, only in term of functional programming techniques instead of formatting or layout algorithms.

2.2.5 clang-format

2.2.6 dartfmt

2.2.7 gofmt

2.2.8 rustfmt

2.2.9 scalariform

There already exists a widely used code formatter for Scala called Scalariform[**scalariform**]. However, Scalariform lacks the ability to enforce a column width limit, which we consider necessary to capture many popular Scala coding styles.

3 Algorithms

3.1 Data structures

3.1.1 FormatToken

3.1.2 Split

3.1.3 Policy

3.1.4 OptimalToken

3.1.5 State

3.1.6 Indent

3.2 BestFirstSearch

3.2.1 Router

- Router
- Search.

3.3 Optimizations

3.3.1 dequeueOnNewStatements

3.3.2 recurseOnBlocks

3.3.3 escapeInPathologicalCases

3.3.4 escapeInPathologicalCases

3.3.5 pruneSlowStates

3.3.6 FormatWriter

- vertical alignment
- comment formatting
- stripMargin alignment

4 Tooling

4.1 Heatmaps

4.2 Configuration

4.2.1 maxColumn

4.2.2 binPacking

4.2.3 vertical alignment

4.3 Unit tests

4.4 Property based tests

4.4.1 AST Integrity

4.4.2 Idempotency

4.5 Regressions tests

5 Evaluation

5.1 Micro benchmarks

5.2 Adoption

6 Discussion

6.1 Future work

6.2 Conclusion

References

[2] *Apache Spark™ - Lightning-Fast Cluster Computing*. URL: http://spark.apache.org/ (visited on 05/29/2016).

[3] *Bill Gosper*. URL: http://gosper.org/bill.html (visited on 05/31/2016).

[4] *ENSIME*. URL: http://ensime.github.io/ (visited on 05/29/2016).

[5] Olafur Pall Geirsson. *Scalafmt - code formatter for Scala*. URL: http://scalafmt.org (visited on 05/29/2016).

[6] Ira Goldstein. *Pretty-printing Converting List to Linear Structure*. Massachusetts Institute of Technology. Artificial Intelligence Laboratory, 1973. URL: http://www.softwarepreservation.net/projects/LISP/MIT/AIM-279-Goldstein-Pretty_Printing.pdf (visited on 05/29/2016).

[7] *Google C++ Style Guide*. URL: https://google.github.io/styleguide/cppguide.html (visited on 05/28/2016).

[8] Sam Halliday. *I don't have time to talk about formatting in code reviews. I want the machine to do it so I can focus on the design.* microblog. May 2016. URL: https://twitter.com/fommil/status/727879141673078785 (visited on 05/29/2016).

[9] R. W. Harris. "Keyboard standardization". In: 10.1 (1956), p. 37. URL: http://massis.lcs.mit.edu/archives/technical/western-union-tech-review/10-1/p040.htm (visited on 05/29/2016).

[10] John Hughes. "The design of a pretty-printing library". In: *Advanced Functional Programming*. Springer, 1995, pp. 53–96. URL: http://link.springer.com/chapter/10.1007/3-540-59451-5_3 (visited on 01/06/2016).

[11] Daniel Jasper. *clang-format*. URL: http://llvm.org/devmtg/2013-04/jasper-slides.pdf (visited on 04/20/2016).

[12] Viktor Klang. *Code style should not be enforced by review, but by automate rewriting. Evolve the style using PRs against the rewriting config.* microblog. Feb. 2016. URL: https://twitter.com/viktorklang/status/696377925260677120 (visited on 05/29/2016).

[13] Donald E. Knuth and Michael F. Plass. "Breaking paragraphs into lines". In: *Software: Practice and Experience* 11.11 (1981), pp. 1119–1184. URL: http://onlinelibrary.wiley.com/doi/10.1002/spe.4380111102/abstract (visited on 05/31/2016).

[14] John McCarthy. "Recursive functions of symbolic expressions and their computation by machine, Part I". In: *Communications of the ACM* 3.4 (1960), pp. 184–195. URL: http://dl.acm.org/citation.cfm?id=367199 (visited on 05/31/2016).

[15] Martin Odersky and Heather Miller. *The Scala Center*. Mar. 2016. URL: http://www.scala-lang.org/blog/2016/03/14/announcing-the-scala-center.html (visited on 05/29/2016).

[16] Martin Odersky et al. *The Scala language specification*. 2004. URL: http://www-dev.scala-lang.org/old/sites/default/files/linuxsoft_archives/docu/files/ScalaReference.pdf (visited on 05/31/2015).

[17] Dereck C. Oppen. "Prettyprinting". In: *ACM Trans. Program. Lang. Syst.* 2.4 (Oct. 1980), pp. 465–483. ISSN: 0164-0925. DOI: 10.1145/357114.357115. URL:

http://doi.acm.org/10.1145/357114.357115 (visited on 04/18/2016).

[18] Tiark Rompf and Nada Amin. "From F to DOT: Type Soundness Proofs with Definitional Interpreters". In: *arXiv:1510.05216 [cs]* (Oct. 2015). arXiv: 1510.05216. URL: http://arxiv.org/abs/1510.05216 (visited on 05/28/2016).

[19] *sbt - The interactive build tool*. URL: http://www.scala-sbt.org/ (visited on 05/28/2016).

[20] *scala-native/scala-native*. URL: https://github.com/scala-native/scala-native (visited on 05/29/2016).

[21] *Scala.js*. URL: http://www.scala-js.org/ (visited on 05/29/2016).

[22] *Scala.js coding style*. URL: https://github.com/scala-js/scala-js/blob/master/CODINGSTYLE.md (visited on 05/28/2016).

[23] R. S. Scowen et al. "SOAP—A program which documents and edits ALGOL 60 programs". In: *The Computer Journal* 14.2 (1971), pp. 133–135. URL: http://comjnl.oxfordjournals.org/content/14/2/133.short (visited on 05/29/2016).

[24] *Stack Overflow Developer Survey 2015*. URL: http://stackoverflow.com/research/developer-survey-2015 (visited on 05/29/2016).

[25] S. Doaitse Swierstra and Olaf Chitil. "Linear, bounded, functional pretty-printing". In: *Journal of Functional Programming* 19.01 (Jan. 2009), pp. 1–16. ISSN: 1469-7653. DOI: 10.1017/S0956796808006990. URL: http://journals.cambridge.org/article_S0956796808006990 (visited on 04/20/2016).

[26] Mark Van Den Brand and Eelco Visser. “Generation of formatters for context-free languages”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5.1 (1996), pp. 1–41. URL: http://dl.acm.org/citation.cfm?id=226156 (visited on 01/06/2016).

[27] Philip Wadler. “A prettier printer”. In: *The Fun of Programming, Cornerstones of Computing* (2003), pp. 223–243. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.635&rep=rep1&type=pdf (visited on 04/20/2016).

[28] Hyrum Wright et al. “Large-Scale Automated Refactoring Using ClangMR”. In: (2013). URL: https://research.google.com/pubs/pub41342.html (visited on 04/21/2016).