ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

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 get right. This thesis addresses the problem of developing a code formatter for the Scala programming language that captures the language's most popular idioms and coding styles. Our work has been limited to formatting Scala code. Still, we have developed data structures, algorithms and tools that we believe can applied to develop code formatters for a variety of other programming languages.

Contents

1 Introduction

The main motivation of this study is to bring a robust, configurable and opinionated code formatter to the Scala community. Although there already exist Scala code formatters, we consider that they do not capture many popular coding styles which are present in the Scala community.

Code formatters are used by software developers to apply stylistic formatting conventions on their source code. Without code formatters, software developers are responsible for manipulating all syntactic trivia in their code. Syntactic trivia includes where to put spaces, insert newlines and how far to indent code fragments. Consider the Scala code snippets in listings 1 and 2.

Listing 1: Unformatted code

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

Listing 2: Formatted code

```scala
// 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 formatting. Listing 2 is arguably easier to read and understand. For example, listing 1 makes it unclear which function call `arg6` belongs to.

Moreover, in collaborative environments without code formatters, the responsibility is left to code reviewers to verify that each contributed software patch adheres to the organizations coding style conventions.

This thesis addresses the problem of automatically converting any Scala program, such as in listing 1, to a well formatted program, such as in listing 2.

The motivation to use code formatters is manifold. Code formatters enforce a consistent coding style in a code base that's touched by multiple developers. By delegating mundane formatting tasks to a formatter, software developers are able to invest their attention to

Code formatting enables automated refactoring tools, see [2]. Code formatting also lowers the barrier to entry for novice programmers, who are not aware of the coding style conventions.

Code formatters have existed for a long time. In more recent years we've seen many new opimization based code formatters, such as clang-format, dartfmt and rfmt. Optimization based code formatters define a fit function for a given layout. A good layout has low cost while a bad layout has a high cost. The formatter runs an algorithm to choose the layout with the lowest code. This idea is not new, for example TeX uses a fit function to optimally break lines in text documents.

Much work has been put into developing language agnostic code formatters or pretty printers. Researchers of such work correctly highlights that developing a custom code formatter for each language requires a lot of effort. It may be possible that we find a perfect way to develop language independent code formatters, see [**mps_article**]. Still, today no such framework exists and people who write Scala code each day are still choosing to manually format their code. The goal of this thesis is not to address language agnostic code formatting.

1.1 Problem statement

This thesis addresses the problem of developing a code formatter, `scalafmt`, for the Scala programming language. This formatter should:

- capture Scala's most common idioms and popular coding styles, most importanly a line length limit.
- be opinionated, it should be able to produce nice looking code even for the most egregiously formatted input.
- be fast, <200ms for 99 percentile of source files, <2s for rest.

1.2 Contributions

The main contributions presented in this thesis are the following:

- We have developed a framework of data structures, algorithms and tooling that help developing code formatters. This work is presented in section X.
- We have applied this framework to develop `scalafmt`, a code formatter for the Scala programming language.

2 Background

2.1 Scala the programming language

Throughout the paper we assume familiarity with the basics of the Scala Programming Language [1].

2.1.1 Pattern matching

2.1.2 Higher order functions

2.1.3 For comprehensions

2.1.4 Dialects

2.1.5 Metaprogramming with scala.meta

- Tree nodes have parent links.
- Token classes are types.

2.2 Code formatting

2.2.1 gofmt

2.2.2 rustfmt

2.2.3 dartfmt

2.2.4 clang-format

2.2.5 scalariform

3 scalafmt

3.1 Data structures

3.1.1 FormatToken

3.1.2 Split combinator

3.1.3 Policy combinator

3.1.4 OptimalToken combinator

3.1.5 State

3.1.6 Indent

3.2 Algorithms

Pipeline of three stages: pre-processing, line wrapping and post-processing.

3.2.1 Pre-processing???

- AST vs. token stream

rustfmt considers that all eventually converge to a hybrid of the two. Indeed, clang-format has an elaborate parser.

3.2.2 BestFirstSearch

- Router
- Search.

3.2.3 FormatWriter

- vertical alignment
- comment formatting
- stripMargin alignment

3.3 Heatmaps

3.4 Configuration

3.4.1 maxColumn

3.4.2 binPacking

3.4.3 vertical alignment

3.5 Optimizations

3.5.1 dequeueOnNewStatements

3.5.2 recurseOnBlocks

3.5.3 escapeInPathologicalCases

3.5.4 escapeInPathologicalCases

3.5.5 pruneSlowStates

3.6 Unit tests

3.7 Property based tests

3.7.1 AST Integrity

3.7.2 Idempotency

3.8 Regressions tests

4 Evaluation

4.1 Micro benchmarks

4.2 Adoption

5 Related work

5.1 Combinator based

1. Houghes 1995
2. Wadler 1999

5.2 Optimization-oriented

1. clang-format Dijkstra's 2010
2. dartfmt Best-first search 2014
3. rfmt 2015

1. Optimal line breaking
2. Oppen

6 Discussion

6.1 Future work

6.2 Conclusion

References

[1] 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).

[2] 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).