
# 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** **6**
1.1 Contributions . . . 7

**2 Background** **8**
2.1 Scala the programming language . . . 8
2.1.1 Higher order functions . . . 9
2.1.2 Immutability . . . 9
2.1.3 SBT build configuration . . . 10
2.2 scala.meta . . . 11
2.3 Code formatters . . . 13
2.3.1 Natural language . . . 13
2.3.2 ALGOL 60 . . . 13
2.3.3 LISP . . . 14
2.3.4 Language agnostic . . . 14
2.3.5 gofmt . . . 15
2.3.6 Scalariform . . . 16
2.3.7 clang-format . . . 17
2.3.8 dartfmt . . . 19
2.3.9 rfmt . . . 19

**3 Algorithms** **22**
3.1 Design . . . 22
3.2 Data structures . . . 23
3.2.1 FormatToken . . . 23
3.2.2 Decision . . . 23
3.2.3 Policy . . . 23
3.2.4 Indent . . . 24
3.2.5 Split . . . 25
3.2.6 State . . . 25
3.3 LineWrapper . . . 26

3.3.1 Router . . . 26
3.3.2 Best-first search . . . 28
3.4 Optimizations . . . 30
3.4.1 dequeueOnNewStatements . . . 30
3.4.2 recurseOnBlocks . . . 32
3.4.3 OptimalToken . . . 33
3.4.4 pruneSlowStates . . . 35
3.4.5 escapeInPathologicalCases . . . 36
3.5 FormatWriter . . . 37
3.5.1 Docstring formatting . . . 38
3.5.2 stripMargin alignment . . . 38
3.5.3 Vertical alignment . . . 39

**4 Tooling** **40**
4.1 Heatmaps . . . 40
4.2 Traceability . . . 40
4.3 Configuration . . . 40
4.3.1 maxColumn . . . 40
4.3.2 binPacking . . . 40
4.3.3 vertical alignment . . . 40
4.4 Testing . . . 40
4.5 Unit tests . . . 40
4.6 Property based tests . . . 40
4.6.1 AST Integrity . . . 40
4.6.2 Idempotency . . . 40
4.7 Regressions tests . . . 40

**5 Evaluation** **41**
5.1 Micro benchmarks . . . 41
5.2 Adoption . . . 41

**6 Discussion** **41**
6.1 Future work . . . 41
6.2 Conclusion . . . 41

## Listings

1 Unformatted code . . . 6
2 Formatted code . . . 6
3 Higher order functions . . . 9

4 Higher order functions expanded . . . 9
5 Manipulating immutable list . . . 10
6 Manipulating mutable list . . . 10
7 SBT project definition . . . 10
8 Parsing different Scala dialects with scala.meta . . . 11
9 Serializing scala.meta trees . . . 12
10 A LISP program . . . 14
11 Gofmt example input/output . . . 16
12 Bin-packing . . . 17
13 No bin-packing . . . 17
14 Unformatted C++ code . . . 18
15 ClangFormat formatted C++ code . . . 19
16 Avoid dead ends . . . 19
17 Line block . . . 20
18 Stack block . . . 20
21 Formatting layout for argument lists . . . 21
19 Line block . . . 21
20 Stack block . . . 21
22 FormatToken definition . . . 23
23 Decision definition . . . 23
24 Policy definition . . . 24
25 Indent definition . . . 24
26 Split definition . . . 25
27 State definition . . . 26
28 Pattern matching on FormatToken . . . 27
29 Unreachable code . . . 27
30 Extracting line number from call site . . . 28
31 Exponential running time . . . 29
32 Two independent statements . . . 30
33 Overeager dequeueOnNewStatements . . . 31
34 recurseOnBlocks example . . . 32
35 OptimalToken definition . . . 33
36 OptimalToken example . . . 34
37 Slow states . . . 35
38 stripMargin example . . . 38

## List of Algorithms

1 Scalafmt best-first search, first approach . . . 29
2 dequeueOnNewStatements optimization . . . 31

3 recurseOnBlocks optimization . . . . . 33
4 OptimalToken optimization . . . . . 34
5 pruneSlowStates optimization . . . . . 35
6 best-effort fallback strategy . . . . . 37

## List of Figures

1 ClangFormat architecture . . . . . 17
2 Scalafmt architecture . . . . . 22
3 Example graph produced by Router . . . . . 27

# 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))
}
```

**Listing 2: Formatted code**

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

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[22], a code formatter, to migrate legacy C++ code to the modern C++11 standard[44]. ClangFormat was used to ensure that the refactored code adhered to Google's extensive C++ coding style[15]. Similar migrations can be expected in the near future for the Scala community once new dialects, such as Dotty[34], gain popularity.

Code formatting is valuable in collaborative coding environments. The Scala.js project[38] has over 40 contributors and the Scala.js coding style[10] – which each Scala.js contributor is expected to know by heart – is written at a whooping 2.600 words. Each contributed patch is manually

verified against the coding style by the project maintainers. This adds a burden on both contributors and maintainers. Several prominent Scala community member have raised this issue. ENSIME[12] 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."[17]. Akka[1] is 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.".[23]. With code formatters, software developers are not burdened by whitespace trivia and can instead direct their full attention on writing correct, maintainable and fast code.

## 1.1 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 has already been adopted by several open source Scala libraries. For details on how to install and use scalafmt, refer to the scalafmt online documentation[13].
- 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, we hope the design of scalafmt can be inspiration to code formatter developers working with 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 history on code formatters that have been developed over the last 70 years. We will see that although code formatters have a long history, a new tradition of optimization based formatters – which scalafmt proudly joins – started only recently in 2013.

## 2.1 Scala the programming language

Scala[30] 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[10], JavaScript has recently become a popular target platform for Scala developers. Even more recently, the announcement of Scala Native[37] 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[29]. Large organizations such as Goldman Sachs, Twitter, IBM and Verizon run Scala code in business critical systems. The 2015 Stack Overflow Developer Survey shows that Scala is the 6th most loved technology and 4th best paying technology to work with[40]. 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

Listing 3: Higher order functions

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

Listing 4: Higher order functions expanded

```
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
```

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 as well as return functions as 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 3. 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` 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 4 shows an equivalent program to listing 3 without using syntactic conveniences. Observe that what was expressed as a single statement in line 1 of listing 3 is expressed with multiple statements in lines 1 and 2 of listing 4.

### 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

**Listing 5: Manipulating immutable list**

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

**Listing 6: 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)
```

benefits to software development in areas including concurrency and testing. Listing 5 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 6 show the equivalent operation using a mutable list. Observe that listing 5 is a single statement while listing 6 contains multiple statements.

### 2.1.3 SBT build configuration

SBT[36] 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 7 is an example project definition in SBT. Observe that

**Listing 7: 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))
```

**Listing 8: Parsing different Scala dialects with scala.meta**

```
import scala.meta._
dialects.Sbt0137(
  """lazy val root = project.dependsOn(core)
     lazy val core = project""").parse[Source] // OK
dialects.Sbt0136(
  """lazy val root = project.dependsOn(core)
     lazy val core = project""").parse[Source] // Missing blank line
// Default dialect, regular Scala compilation unit
"""lazy val root = project""".parse[Source] // No top-level statements
```

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 over dozens or even hundreds of lines of code.

## 2.2 scala.meta

Scala.meta[5] is a metaprogramming toolkit for Scala. Before scala.meta, the state-of-the-art metaprogramming facilities relied on the Scala compiler internals. This had several severe limitations such as too-eager desugaring resulting in loss of syntactic details from the original source code. Scala.meta was designed to overcome these limitations and offer a more robust platform to develop metaprogramming tools for Scala. Several key features of scala.meta have made it an invaluable companion in the development of scalafmt. Most notably among these features are dialect agnostic syntax trees, syntax tree serialization, input fidelity and algebraically typed tokens.

Scala.meta provides facilities to tokenize and parse a variety of different Scala dialects. One such dialect is SBT configuration files, discussed in section 2.1.3. SBT adds custom support for top-level statements in `*.sbt` files, which would otherwise result in a parse error using the Scala compiler parser. To add insult to injury, top-level statements must be separated by a blank line if you use an SBT version lower than 0.13.6; a restriction that was lifted in SBT 0.13.7. Listing 8 show how to scala.meta makes it trivial to accommodate this zoo of nuances. The result after parsing is a dialect agnostic scala.meta tree structure.

The structure of scala.meta trees can be serialized to strip off all syntactic

**Listing 9: Serializing scala.meta trees**

```
import scala.meta._
> """ object Main extends App { self =>
        println(s"Hello $self!") // This is a comment
      }""".parse[Source].get.structure
res0: String = """
Source(Seq(Defn.Object(Nil, Term.Name("Main"), Template(Nil, Seq(Ctor.Ref.Name("App
    ")), Term.Param(Nil, Term.Name("self"), None, None), Some(Seq(Term.Apply(Term.
    Name("println"), Seq(Term.Interpolate(Term.Name("s"), Seq(Lit("Hello "), Lit
    ("!")), Seq(Term.Name("self")))))))))))
"""
```

details. Listing 9 shows how to serialize the tree structure of a simple hello world application. For example, observe that the comment has been stripped away. As we discuss in section 4, this feature was instrumental in testing scalafmt.

Tree node types in scala.meta preserve absolute fidelity with the original source file. This means we can obtain all syntactic details from a tree node such as whether a for comprehension uses parentheses or curly braces as delimiters, whitespace positions and comments. The Scala compiler is infamous for desugaring for-comprehensions into `map/withFilter/flatMap` applications during the parse phase. This made it impossible to implement metaprogramming tasks such as code formatting. Input source fidelity in scala.meta has been essential to implement scalafmt because we need to preserve some syntactic details like where for-comprehension and blank lines are used.

Tokens in scala.meta are strongly typed. Traditional object-oriented libraries treat tokens as a single type with multiple methods such as `isComma/isFor` which returns true if a token instance is a comma or a `for` keyword. However, scala.meta leverages algebraic data types in Scala to represent each different kind of token as a separate type. This feature plays nicely with the pattern matching capabilities of Scala and enabled design pattern for the *Router* described in section 3.3.1.

## 2.3 Code formatters

Code formatting and pretty printing[1] has a long tradition. In this chapter, we look at a variety of tools and algorithm that have been developed over the last 70 years.

### 2.3.1 Natural language

The science of displaying aesthetically pleasing text predates as early as 1956[19]. The first efforts involved inserting carriage returns in natural language text. Until that time, writers had been responsible for manually providing carriage returns in their documents before sending them off for printing. The motivation was to "save operating labor and reduce human error". Once type-setting became more commonplace, the methods for breaking lines of text got more sophisticated.

Knuth and Plass developed in 1981 a famous line breaking algorithm[25] for LaTeX, a popular typesetting program among scientific academic circles. LaTeXis the program that was used to generate this very document. The line breaking problem was the same as in the 60s: how to optimally break a paragraph of text into lines so that the right margin is minimized. The primitive approach is to greedily fit as many words on a line as possible. However, such an approach can produce embarrassingly bad output in the worst case. Knuth's algorithm uses dynamic programming to find an optimal layout with regards to a fit function that penalizes empty space on the right margin of the paragraph. This algorithm remains a textbook example of an application of dynamic programming[11, 24].

### 2.3.2 ALGOL 60

Scowen[39] 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.

[1] This thesis uses the term *code formatting* over *pretty printing*. According to Hughes[20], pretty printing is a subset of code formatting where the former is only concerned with presenting data structures while the latter is concerned with the harder problem of formatting existing source code — the main topic of this thesis.

**Listing 10: A LISP program**

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

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.

### 2.3.3 LISP

In 1973, Goldstein[14] explored code formatting algorithms for LISP[26] programs. LISP is a family of programming languages and is famous for it's parenthesized prefix notation. Listing 10 shows a program in LISP to calculate factorial numbers. The simple syntax and extensive use of parentheses as delimiters makes make LISP programs 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, with a fallback to *standard-format*, i.e. 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 case. 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 relevant concerns for modern code formatters.

### 2.3.4 Language agnostic

Derek C. Oppen pioneered the work on language agnostic code formatting in 1980[32]. A language agnostic formatting algorithm can be used for a

variety of programming languages instead of being tied to a single language. Users provide a preprocessor to integrate a particular programming language with the algorithm. Oppen's algorithm runs in $O(n)$ time and uses $O(m)$ memory for an input program of length $n$ and maximum column width $m$. 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 that generates a formatter given a context-free grammar[42]. Beyond the usual motivations for developing code formatters, Brand mentions that formatters "relieve documentation writers from typesetting programs by hand". The focus on documentation is reflected by the fact that the generated formatter could produce both ASCII formatted code as well at LaTeX markup. Since comments are typically not included a syntax tree, the presented algorithm has an elaborate scheme to infer the location of comments in the produced output. Like Oppen's algorithm, this library requires the user to plug in a preprocessor to integrate a particular programming language into the Brand's library. Unlike Oppen's algorithm, Brand does not consider line length limits in his algorithm.

John Hughes extended on Oppen's work on language agnostic formatting in term of functional programming techniques[20]. Hughes presented a design of a *pretty-printing* library that leverages combinators with algebraic properties to express formatting layouts. Hughes claims that such a formal approach was invaluable when designing the pretty-printing library, which has seen been widely used including in the Glasgow Haskell compiler. Wadler[43] and Chitil[41] extend on Hughes's and Oppen's work in term of performance and programming techniques. However, this branch of work has been limited to printing data structures and not how to format format existing source code.

### 2.3.5 gofmt

`gofmt`[16] is a code formatter for the Go programming language, developed at Google. `gofmt` was released in the early days of Go in 2009 and is

noteworthy for its heavy adoption by the Go programming community. Official Go documentation[6] claims that almost all written Go code, at Google and elsewhere, is formatted with `gofmt`. Besides formatting, `gofmt` is used to automatically migrate Go codebases from legacy versions to new source-incompatible releases. However, `gofmt` supports neither a column limit nor an opinionated setting. Line breaks are preserved in the user's input. For example, listing 11 shows a Go program that uses the same layout as the "unformatted" code (listing 1) in the introduction.

**Listing 11: Gofmt example input/output**

```
package main

func main() int {
  function(arg1, arg2(arg3(
    "String literal"),
    arg4+arg5))
}
```

The output of running `gofmt` through listing 11 is identical to the input. This un-opinionated behavior may be considered desirable by many software developers. However, this thesis is only concerned with opinionated code formatting.

### 2.3.6 Scalariform

Scalariform[35] was released in 2010 and is a widely used code formatter for Scala. Like `gofmt`, Scalariform does an excellent job of tidying common formatting errors. Moreover, Scalariform supports a variety of configuration options. Scalariform is also impressively fast, it can format large files with over 4.000 lines of code in under 250 milliseconds on a modern laptop. However, Scalariform shares the same limitations with `gofmt`: it lacks a line length and opinionated setting.

Firstly, the line length setting is necessary to implement many popular coding styles in the Scala community. For example, the Spark[45] and Scala.js[10] coding styles have 100 character and 80 character column limits, respectively. As we see in other code formatters, adding a line length setting is non-trivial and doing so would require a significant redesign of Scalariform.

Secondly, the lack of an opinionated setting makes it impossible to enforce

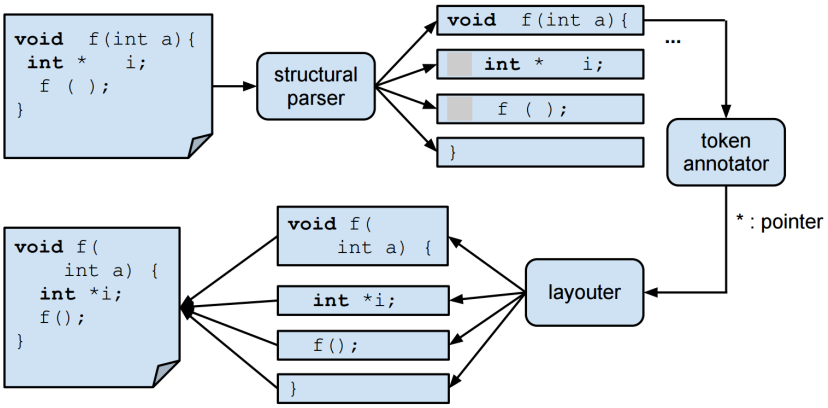


Figure 1: ClangFormat architecture

certain coding styles. For example, the Scala.js coding style enforces *bin-packing*, where arguments should be arranged compactly up to the column length limit. Listings 12 and 13 shows an example of bin packing enabled and disabled, respectively.

**Listing 12: Bin-packing**

```
1 // Column 35                          |
2 class Foo(val x: Int, val y: Int,
3     val z: Int)
4
```

**Listing 13: No bin-packing**

```
1 // Column 35                          |
2 class Foo(val x: Int,
3           val y: Int,
4           val z: Int)
```

Since Scalariform preserves the line breaking decisions from the input, Scalariform has no setting to convert formatted code like in listing 13 to the code in listing 12.

### 2.3.7 clang-format

Daniel Jasper triggered a new trend in optimization based coded formatters with the release of *ClangFormat*[21] in 2013. ClangFormat is developed at Google and can format C, C++, Java, JavaScript, Objective-C and Protobuf code. Figure 1 shows the architecture of ClangFormat. The main components are the *structural parser* and the *layouter*.

ClangFormat employs a structural parser to split source code into a sequence of *unwrapped-lines*. An unwrapped line is a statement that should

**Listing 14: Unformatted C++ code**

```
1 int main(int argc,char const*argv[]) { Defn.Object(Nil, "ClangFormat", Term.Name("
      State"), Foo.Bar( Template(Nil, Seq( Ctor.Ref.Name("ClangLogger")), Term.Param(
      Nil, Name.Anonymous(), None, None)) ), Term.Name("clang-format") ); }
```

fit on a single line if given sufficient line length. A key feature of unwrapped lines is that they should not influence other unwrapped lines. The parser is lenient and parses even syntactically invalid code. The parsed unwrapped lines are passed onto the layouter.

The ClangFormat layouter uses a novel approach to implement line wrapping. Each line break is assigned a penalty according to several rules such as nesting and token type. At each token, the layouter can choose to continue on the same line or break. This forms an acyclic weighted directed graph with tokens representing vertices and splits (e.g., space, no space or line break) representing edges. The first token of an unwrapped line is the root of the graph and all paths end at the last token of the unwrapped line. The layouter uses Dijkstra's[9] shortest path algorithm to find the layout that has the lowest penalty. To obtain good performance, the layouter uses several domain specific optimizations to minimize the search space.

Despite being seemingly language independent, ClangFormat does not leverage the language agnostic formatting techniques described section 2.3.4. Support for each language has been added as ad-hoc extensions to the ClangFormat parser and layouter. ClangFormat supports a variety of configuration options, including 6 out-of-the-box styles based on coding styles from Google, LLVM and other well-known organizations.

A notable feature of ClangFormat is that it's opinionated. ClangFormat produces well-formatted output for even the most egregiously formatted input. Listing 14 shows an offensively formatted C++ code snippet. Listing 15 shows the same snippet after being formatted with ClangFormat. ClangFormat is opinionated in the sense that it does not respect the user's line breaking decisions. This feature makes it possible to ensure that all code follows the same style guide, regardless of author.

**Listing 15: ClangFormat formatted C++ code**

```
int main(int argc, char const *argv[]) {
  Defn.Object(Nil, "ClangFormat", Term.Name("State"),
              Foo.Bar(Template(Nil, Seq(Ctor.Ref.Name("ClangLogger")),
                               Term.Param(Nil, Name.Anonymous(), None, None))),
              Term.Name("clang-format"));
}
```

**Listing 16: Avoid dead ends**

```
// Column 35                    |
function(
    firstCall(a, b, c, d, e),
    secondCall("long argument string"));
```

### 2.3.8 dartfmt

Dartfmt[27] was released in 2014 and follows the optimization based trend initiated by ClangFormat. Darftfmt is a code formatter for the Dart programming language, developed at Google. Like ClangFormat, dartfmt has a line length setting and is opinionated. Bob Nystrom, the author of dartfmt, discusses the design of dartfmt in an excellent post[28] on his blog. In his post, Nystrom argues that the design of a code formatters is significantly complicated by a column limit setting. The line wrapping algorithm in dartfmt employs a *best-first search*[33], a minor variant of the shortest path search in ClangFormat. As with ClangFormat, a range of domain-specific optimizations were required to make the search scale for real-world code. Listing 16 shows an example of such an optimization, *avoiding dead ends*. The snippets exceeds the 35 character column limit. A plain best-first search would perform a lot of redundant search inside the argument list of `firstCall`. However, `firstCall` already fits on a line and there is no need to explore line breaks inside its argument list. The dartfmt optimized search is able to eliminate such dead ends and quickly figure out to break before the `"long argument string"` literal.

### 2.3.9 rfmt

The most recent addition to the optimization based formatting trend is `rfmt`[46], a code formatter for the statistical programming environment *R*.

The formatter was released in 2016 – after the background work on this thesis started – and like its forerunners is also developed at Google. `rfmt` makes an interesting contribution in that it combines the algebraic combinator approach from Hughes[20] and the optimization based approach from LaTeX and ClangFormat.

The algebraic combinator approach makes it easy to express a variety of formatting layouts. `rfmt` uses 6 layout combinators or *blocks* as they are called in the report. The blocks are the following:

- $TextBlock(txt)$: unbroken string literal.
- $LineBlock(b_1, b_2, \ldots, b_n)$: horizontal combination of blocks.
- $StackBlock(b_1, b_2, \ldots, b_n)$: vertical combination of blocks.
- $ChoiceBlock(b_1, b_2, \ldots, b_n)$: selection of a best block.
- $IndentBlock(n, l)$: indent block $b$ by $n$ spaces.
- $WrapBlock(b_1, b_2, \ldots, b_n)$: Fit as many blocks on each line as possible, break when the column limit is exceeded and align by the first character in $b_1$.

We'll use an example to show how these relatively few combinators allow an impressive amount of flexibility. Listings 17 and 18 shows two different layouts to format an argument list.

**Listing 17: Line block**

```
// Column 35                    |
function(argument1, argument2,
         argument3, argument4,
         argument5, argument6)
```

**Listing 18: Stack block**

```
// Column 35                    |
function(
  argument1, argument2, argument3,
  argument4, argument5, argument6
)
```

In this case, we prefer the line block from listing 17 since it requires fewer lines. However, our preference changes if the function name is longer as is shown in listings 19 and 20.

**Listing 21: Formatting layout for argument lists**

```
ChoiceBlock(LineBlock(LineBlock(TextBlock(f), TextBlock("("))),
                      WrapBlock(a1, ... , am),
                      TextBlock(")"),
            StackBlock(LineBlock(TextBlock(f), TextBlock("("))),
                       IndentBlock(4, WrapBlock(a1, ... , am)),
                       TextBlock(")")).
```

**Listing 19: Line block**

```
1 // Column 35                        |
2 functionNameIsLonger(argument1,
3                      argument2,
4                      argument3,
5                      argument4,
6                      argument5,
7                      argument6)
8
```

**Listing 20: Stack block**

```
1 // Column 35                        |
2 functionNameIsLonger(
3   argument1, argument2, argument3,
4   argument4, argument5, argument6
5 )
6
7
8
```

Here, we clearly prefer the stack block in listing 20. Listing 21 shows how we use the 6 fundamental blocks in the `rfmt` combinator algebra to express the choice between these formatting layouts. The variable $f$ denotes the function name and $a_1, ...a_m$ denote the argument list. Observe that listing 21 does not express how to find the optimal layout.

To find an optimal layout, `rfmt` employs a smart dynamic programming trick. First, it is possible enumerate all layout combinations like the re-predictor algorithm does in section 2.3.3. This leads to exponential growth which turns out to be a problem for some cases. Dynamic programming comes to the rescue by allowing us to reuse partial solutions. Instead of re-calculating the layout cost at each (starting column, block) pair, we store the result in an associative array keyed by the starting column. However, it turns out that this can still be inefficient in terms of memory and speed[2]. To overcome this limitation, Yelland – the `rfmt` author – presents an indexing scheme that makes it possible to extrapolate the layout cost even for missing keys. We refer to the original paper[46] for details. This novel approach enables `rfmt` to format even the most pathologically nested code in near instant time.

[2] In fact, ClangFormat started with a similar approach, as explained in this[7] video recording, but then switched to Dijkstra's shortest path algorithms (which in itself is another form of dynamic programming).

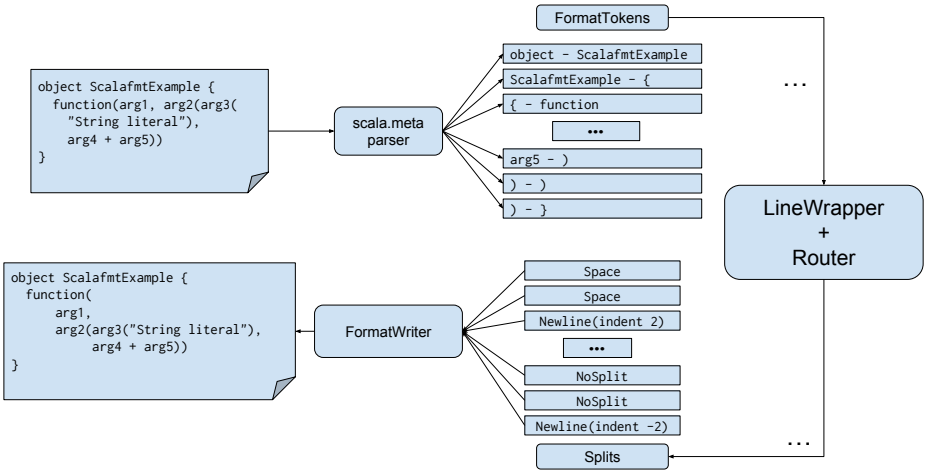


Figure 2: Scalafmt architecture

# 3 Algorithms

This chapter describes how scalafmt formats Scala code. We will see that scalafmt's design is inspired by ClangFormat and dartfmt. However, we believe our design makes a valuable contribution in that it leverages functional programming principles to maximise code reuse and extensibility.

## 3.1 Design

Figure 2 shows a broad architectural overview of scalafmt. First, scalafmt parses a source file using scala.meta. Next, we feed a sequence of *FormatToken* data types into a *LineWrapper*. The LineWrapper uses a *Router* to construct a weighted directed graph and run a best-first search to find an optimal formatting layout for the whole file. Finally, the LineWrapper feeds a sequence of *Split* data types into the *FormatWriter*, which constructs a new reformatted source file. The following sections explain these data types and abstractions in detail.

## 3.2 Data structures

Scalafmt leverages a few carefully designed data structure to allow an implementation that emphasizes correctness and maintainability.

### 3.2.1 FormatToken

A *FormatToken* is a pair of two non-whitespace tokens. Listing 22 shows the definition of the FormatToken data type.

**Listing 22: FormatToken definition**

```
1 case class FormatToken(left: Token, right: Token, between: Vector[Whitespace])
```

As shown in the architecture overview in figure 2, each token except the beginning and end of file tokens appear twice in the sequence of FormatTokens: once as the `left` member and once as the `right` member. In a nutshell, the job of the LineWrapper is to convert each FormatToken into a *Split*

### 3.2.2 Decision

A Decision is a pair of a FormatToken and a sequence of Splits. Listing 23 shows the Definition of decision.

**Listing 23: Decision definition**

```
1 case class Decision(formatToken: FormatToken, splits: Seq[Split])
```

The *splits* member represents the possible splits that we can take at *formatToken*.

### 3.2.3 Policy

A *Policy* is an enforced formatting layout over a region. Listing 24 shows the definition of Policy.

**Listing 24: Policy definition**

```
case class Policy(f: PartialFunction[Decision, Decision],
                  expire: Token)
```

A Policy is a partial function that should be applied to future Decisions up until the *expire* token. Policies easily compose using the Scala standard library `orElse` and `andThen` methods on PartialFunction[3]. Policies enable a high-level way to express arbitrary formatting layouts over a region of code.

### 3.2.4 Indent

An *Indent* describes indentation over a region of code.

**Listing 25: Indent definition**

```
sealed abstract class Length
case class Num(n: Int) extends Length
case object StateColumn extends Length

case class Indent[T <: Length](length: T, expire: Token, inclusive: Boolean)
```

Listing 25 shows the definition of Indent along with the algebraic data type *Length*. Length can either be `Num(n)` where $n$ represents an explicit number of spaces to indent by or `StateColumn` which is a placeholder the number of spaces required to vertically align by the current column. Indent is type parameterized by Length so that, at some point, we can replace StateColumn placeholders with Nums to obtain a concrete number. For example, given a scala.meta tree `expr`, the definition `Indent(Num(2), expr.tokens.last, inclusive=true)` increases the indentation level by 2 spaces up to and including the last token of `expr`. The `inclusive` member is set to false when the indentation should expire before the expire token, for example in a block wrapped by curly braces, since the closing curly brace should not be indented by 2 spaces. The StateColumn placeholder is required to allow memoization of Splits, which is critical for performance reason as explained in section 3.3.1 on the *Router*.

[3] Careful eyes will observe that Policy is in fact a monoid with the empty partial function as identity and function composition as associative operator.

### 3.2.5 Split

A *Split* represents a (possibly empty) whitespace character to be inserted between two non-whitespace tokens. Listing 26 shows the rather intricate definition of the Split data type[4].

**Listing 26: Split definition**

```
case class Split(modification: Modification,
                 cost: Int,
                 policy: Policy,
                 optimalAt: Option[OptimalToken],
                 indents: Vector[Indent[Length]])(
    implicit val line: sourcecode.Line)
```

The Split data type went through several generations of design before reaching its current structure. Each member serves an important role. The most important member of the Split type is the *modification*. A modification must be one of `NoSplit`, `Space` and `Newline`. The *cost* member represents the penalty for choosing this split. The *optimalToken* member enables an optimization explained in section 3.4.3. The *indents* member contains the indentation layers that this splits adds. The *line* member allows a powerful debugging technique explained in section 3.3.1. The *policy* and *indents* members are explained in sections 3.2.3 and 3.2.4, respectively.

### 3.2.6 State

A *State* is a partial formatting solution during by the best-first search. Listing 27 shows the definition of the State class and companion object.

[4] For clarity reasons, a few less important members have been removed from the actual Split definition.

**Listing 27: State definition**

```
case class State(splits: Vector[Split],
                 totalCost: Int,
                 policies: Vector[Policy],
                 indents: Vector[Indent[Num]],
                 indentation: Int,
                 column: Int,
                 formatOff: Boolean) extends Ordered[State] {

  def compare(that: State): Int
}

object State {
  def nextState(currentState: State, formatToken: FormatToken, split: Split): State
}
```

Observe the similarity of State and Split. A State contains various summaries calculated from the `splits` member. The summaries are necessary for performance reasons in the best-first search. Observe that the indents are type parameterized by `Num`, meaning they only contain concrete indentations and no `StateColumn` indents. The `indentation` member is the sum of all currently active indents and `column` represents how many characters have been consumed since the last newline. The State class extends the `Ordered` trait to allow for efficient polling from a priority queue. The `compare` method orders States firstly by their `totalCost` member, secondly by `splits.length` – how many FormatTokens have been formatted – and finally breaking ties by the `indentation`. The method `nextState` calculates a penalty for characters that overflow the column limit and prepares The method is implemented as efficiently as possible since the method is on a hot path in the best-first search.

, new summaries to instantiate a new State.

## 3.3 LineWrapper

The LineWrapper is responsible for turning FormatTokens into Splits. To accomplish this, the LineWrapper employs a *Router* and a best-first search.

### 3.3.1 Router

The Router's role is to produce a Decision given a FormatToken. Figure 3 shows all possible formatting layout for the small input `val x = y + z`. In

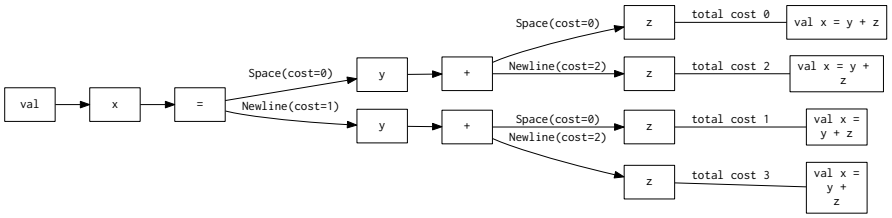


Figure 3: Example graph produced by Router

this figure, the Router is the planner that chooses which nodes open up multiple branches (= and +) and which nodes have one exit edge only. This is no easy task since a FormatToken can be any pair of two tokens. How do we go about implementing a Router?

The Router is implemented as one large pattern match on a FormatToken. Listing 28 shows how to pattern match on a FormatToken.

**Listing 28: Pattern matching on FormatToken**

```
formatToken match {
  case FormatToken(_: Keyword, _) => Seq(Split(Space, 0))
  case FormatToken(_, _: `=`)     => Seq(Split(Space, 0))
  case FormatToken(_: `=`, _)     => Seq(Split(Space, 0)
                                         Split(Newline, 1))
  // ...
}
```

The pattern `_: `=`` matches a scala.meta token of type `=`. The underscore _ ignores the underlying value. Keyword is a super-class of all scala.meta keyword token types. Now, a good observer will notice that this pattern match can quickly grow unwieldy long once you account for all of Scala's rich syntax. How does this solution scale? Also, once the match grows bigger how can we know from which case each Split origins? It turns out that Scala's pattern matching and scala.meta's algebraically typed tokens are able to help us.

The Scala compiler can statically detect unreachable code. If we add a case that is already covered higher up in the pattern match, the Scala compiler issues a warning. For example, listing 29 shows an example where the compiler issues a warning.

**Listing 29: Unreachable code**

```
formatToken match {
  case FormatToken(_, _: Keyword) => Seq(Split(Space, 0))
  // ...
  case FormatToken(_, _: `else`)  => Seq(Newline(, 0)) // Unreachable code!
}
```

Here, we accidentally match on a FormatToken with an `else` keyword on the right which will never match because we have a broader match on a Keyword higher up. In this small example, the bug may seem obvious but once the Router grows bigger the compiler becomes unmissable. However, this still leaves us with the second question of finding the origin of each Split. Scala macros[4] and implicits[31] come to the rescue.

The source file line number of where a Split is instantiated is automatically attached on each Split. Remember in listing 26 that the Split case class had an implicit member of type `sourcecode.Line`. Sourcecode[18] is a tiny Scala library to extract source code metadata from your programs. The library leverages Scala macros and implicits to unobtrusively surface useful information such as line number of call sites. Listing 30 shows how this works.

**Listing 30: Extracting line number from call site**

```
Split(Space, 0) /* expands into */ Split(Space, 0)(sourcecode.Line(1))
```

When a `sourcecode.Line` not passed explicitly as an argument to Split's constructor, the Scala compiler will trigger its implicit search to fill the missing argument. The `sourcecode.Line` companion contains an implicit macro that generates a Line instance from an extracted line number. Take a moment to appreciate how these two advanced features of the Scala programming language enable a very powerful debugging technique. The scalafmt router implementation contains 88 cases and spans over 1.000 lines of code. The ability to trace the origin of each Split has been indispensable in the development of the Router.

### 3.3.2 Best-first search

The Decisions from the Router produce a directed weighted graph, as demonstrated in figure 3. To find the optimal formatting layout, our challenge is to find the cheapest path from the root node to a final node. The best-first[33] algorithm is an excellent fit for the task.

Best-first search is a graph search algorithm to efficiently traverse a directed weighted graph. The objective is reach the final token and once we reach there, we terminate the search because we're guaranteed no other solution is better. Algorithm 1 shows a first attempt[5] to adapt a best-first search algorithm to the data structures and terminologies introduced so far. In the best case, the search always chooses the cheapest splits and the

**Algorithm 1:** Scalafmt best-first search, first approach

```
/** @returns Splits that produce and optimal formatting layout */
def bestFirstSearch(formatTokens: List[FormatTokens]): List[Split] = {
  val Q = mutable.PriorityQueue(State.init(formatTokens.head))
  while (Q.nonEmpty) {
    val currentState = Q.pop
    if (currentState.formatToken == formatTokens.last) {
      return currentState.splits // reached the final state.
    } else {
      val splits = Router.getSplits(currentState.formatToken)
      splits.foreach { split =>
        Q += State.nextState(currentState, split)
      }
    }
  }
  // Error: No formatting solution found.
  ???
}
```

algorithm runs in linear time. Observe that the router is responsible for providing well-behaved splits so that we never hit on the error condition after the while loop. Excellent, does that mean the search is complete? Absolutely not, this implementation contains several serious performance issues.

Algorithm 1 is exponential in the worst case. For example, listing 31 shows a tiny input that triggers the search to explore over 8 million states.

**Listing 31: Exponential running time**

```
// Column 60                                                  |
a + b + c + d + e + f + g + h + i + j + k + l +
m + n + o + p + q + r + s + t + v + w + y +
// This comment exceeds column limit, no matter what path is chosen.
z
```

Even if we could visit 1 state per microsecond — reality is closer to 10

[5] Unfortunately, we make heave use of mutation since graph search algorithms typically don't lend themselves well to functional programming principles.

states/microsecond — the search will take almost 1 second to complete. This is unacceptable performance to format only 2 lines of code. Of course, we could special-case long comments, but that would only provide us a temporary solution. Instead, like with ClangFormat and dartfmt, we must apply several domain specific optimizations. In the following section, we discuss the optimizations that have shown to work well for scalafmt.

## 3.4 Optimizations

This section explains the most important domain-specific optimizations that were required to get acceptable performance for scalafmt. We will see that some optimizations are quite ad-hoc and require some creative workarounds.

### 3.4.1 dequeueOnNewStatements

Once the search reaches the beginning of a new statement, empty the priority queue. Observe that the formatting layout for each statement is independent from the formatting layout of the previous statement. Consider listing 32.

**Listing 32: Two independent statements**

```
1 // Column 60                                                  |
2 statement1(argument1, argument2, argument3, argument5, argument6)
3 statement2(argument1, argument2, argument3, argument5, argument6)
```

Both statements exceed the column limit, which means that the search must back-track to some extent. However, once the search reaches `statement2` we have already found an optimal formatting layout for `statement1`. When we start backtracking in `statement2`, there is no need to explore alternative formatting layouts for `statement1`. Instead, we can safely empty the search queue once we reach the `statement2` token.

The `dequeueOnNewStatements` optimization is implemented by extending algorithm 1 with an if statement. Algorithm 2 shows a rough sketch of how this is done. With an empty queue, we ensure the search backtracks only as far back as is needed. The `statementStarts` variable contains all tokens that begin a new statement. To collect those tokens, we traverse the syntax

**Algorithm 2:** dequeueOnNewStatements optimization

```
// ...
val statementStarts: Set[Token]
while (Q.nonEmpty) {
  val currentState = Q.pop
  if (statementStarts.contains(currentState.formatToken.left)) {
    Q.dequeueAll // empty search queue
  }
  // ...
}
```

tree of the input source file and select the first tokens of each statement of a block, each case in a partial function, enumerator in a for comprehension and so forth. The actual implementation is quite elaborate and is left out of this thesis for clarity reasons. Unfortunately, our optimization has one small problem.

Algorithm 2 may dequeue too eagerly inside nested scopes, leading the search to hit the error condition. Listing 33 shows an example where this happens.

**Listing 33: Overeager dequeueOnNewStatements**

```
// Column 50                                        |
function1(argument1, { case `argument2` => 11 }, argument3 // forced newline
          argument4)
```

Remember that each case of a partial function starts a new statement. The `dequeueOnNewStatements` optimization will dequeue the queue on the first state that reaches the `case` token. In this example, the first state to reach the `case` token will have a strict Policy that disallows newlines up until the closing parenthesis. However, we must insert a newline after the comment. This causes the search to terminate too early and reach the error condition. By inspecting where this problem occurred, we came up with a simple rule to identify regions where the `dequeueOnNewStatements` optimization should be disabled. The simple rule is to never run `dequeueOnNewStatements` inside a pair of parentheses. In section 4, we discuss techniques we used to be confident that this rule indeed works as intended. In the following section (3.4.2) we explain the `recurseOnBlocks` optimization, which allows us to reenable `dequeueOnNewStatements` for selected regions inside parentheses.

### 3.4.2 recurseOnBlocks

If the `dequeueOnNewStatements` optimization is disabled and we start a new block delimited by curly braces, recursively run the best-first search inside the block. The intuition here is that by recursively running the best-first search, we keep the priority queue small at each layer of recursion. This allows us to run aggressive optimizations such as `dequeueOnNewStatements`.

The `recurseOnBlocks` optimization enables scalafmt to handle idiomatic Scala code where large bodies of higher order functions and blocks are passed around as arguments. Remember from section 2.1 that Scala makes it syntactically convenient to in higher order functions as arguments to other functions. Listing 34 shows an example where this happens and we trigger the `recurseOnBlocks` optimization.

**Listing 34: recurseOnBlocks example**

```
function(argument1, { higherOrderFunctionArgument =>
  statement1
  // ...
  statementN
})
```

The `dequeueOnNewStatements` optimization is disabled inside argument list. The priority queue grows out bounds because the higher order function can have an arbitrary number of statements.

To implement the `recurseOnBlocks` optimization, we add an extension to algorithm 1. Algorithm 3 shows a rough sketch of how `recurseOnBlocks` is implemented. We change the signature to accept a starting State and token where we stop the search. Observe that we guard against infinite recursion by not making a recursive call on `start.formatToken`. With `recurseOnBlocks` and `dequeueOnNewStatements`, we have solved most problems caused by independent statements affecting the formatting layouts of each other. Next, we leverage recursion again to help the search queue stay small.

**Algorithm 3:** recurseOnBlocks optimization

```
def bestFirstSearch(start: State, stop: Token): List[Split] = {
  val Q = mutable.PriorityQueue(start)
  while (Q.nonEmpty) {
    val currentState = Q.pop
    if (currentState.formatToken.left == stop) {
      return currentState
    } else if (currentState.formatToken != start.formatToken &&
               currentState.formatToken.left.isInstanceOf[`{`]) {
      bestFirstSearch(currentState, closingCurly(currentState.formatToken.left))
    }
   // ...
  }
}
```

**Listing 35: OptimalToken definition**

```
case class OptimalToken(token: Token, killOnFail: Boolean = false)
```

### 3.4.3 OptimalToken

An OptimalToken is a hint from a Split to the best-first search that enables the search to early eliminate competing Splits. Recall from listing 26 that a Split has an `optimalToken` member. Listing 35 shows the definition of OptimalToken. When the best-first search encounters a Split with a defined OptimalToken, the best-first search makes an attempt to reach that token with a budget of 0 cost. If successful, the search can eliminate the competing Splits. If unsuccessful and the `killOnFail` member is true, the best-first search eliminates the Split. Otherwise, the best-first search continues as usual.

By eliminating competing branches, we drastically minimize the search space. Listing 36 shows an example where the OptimalToken can be applied. Scalafmt supports 4 different ways to format call-site function applications. This means that there will be $4^N$ number of open branches when the search reaches `UserObject` $N$. To overcome this issue, we define an OptimalToken at the closing parenthesis. The best-first search successfully fits the argument list of each `UserObject` on a single line, and eliminates the 3 other competing branches. This makes the search run in linear time as opposed to exponential.

**Listing 36: OptimalToken example**

```
// Column 50                                        |
Database(
    UserObject(name1, age1),
    UserObject(name2, age2),
    // ...
    UserObject(nameN, ageN) // comment will always exceed column limit
)
```

To implement the `OptimalToken` optimization, we add an extension to algorithm 3. Algorithm 4 sketches how the extension works. The

**Algorithm 4:** OptimalToken optimization

```
def bestFirstSearch(start: State, stop: Token, maxCost: Int): List[Split] = {
  // ...
  val splits = Router.getSplits(currentState.formatToken)
  var optimalFound = false
  splits.withFilter(_.cost < maxCost).foreach { split =>
    val nextState = State.nextState(currentState, split)
    split.optimalToken match {
      case Some(OptimalToken(expire, killOnFail)) =>
        val nextNextState = bestFirstSearch(nextState, expire, maxCost = 0)
        if (nextNextState.expire == expire) {
          optimalFound = true
          Q += nextNextState
        } else if (!killOnFail) {
          Q += nextState
        }
      case _ if !optimalFound =>
        Q += nextState
    }
  }
  // ...
}
```

`bestFirstSearch` method has a new `maxCost` parameter, which is the highest cost that a new splits can have. Next, if a Split has defined an `OptimalToken` we make an attempt to format up to that token. If successful, we update `optimalFound` variable to eliminate other Splits from being added to the queue. If unsuccessful and `killOnFail` is true, we eliminate the Split that defined the OptimalToken. A straightforward extension to this algorithm would be to add a `maxCost` member to the `OptimalToken` definition from listing 35. However, this has not been necessary for scalafmt.

### 3.4.4 pruneSlowStates

The pruneSlowStates is a optimization that eliminates states that progress slowly. A state progresses slowly if it visits a token after other equally or less expensive states. The insight is that if two equally expensive states visit the same token, the first state to visits that token typically produces a better formatting layout.

By eliminating slow states, we obtain a better formatting output in addition to minimizing the search space. Listing 37 shows two equally expensive formatting solutions where one solution is fast and other is slow.

**Listing 37: Slow states**

```
// Column 30                |

// Fast state
a + b + c + d + e + f + g +
h + i + j
// slow state
a + b + c +
d + e + f + g + h + i + j
```

Of course, the line break after `g +` could be more expensive than the line break after `c +`. Instead, the Router is free to assign an identical cost to both line breaks and let `pruneSlowStates` take care of eliminating the slow state.

The `pruneSlowStates` is implemented as a extension to algorithm 4. Algorithm 5 shows a rough sketch of how the extension works.

**Algorithm 5:** pruneSlowStates optimization

```
// ...
val fastStates: mutable.Map[FormatToken, State]
while (Q.nonEmpty) {
  val currentState = Q.pop
  if (fastStates.get(currentState.formatToken)
                .exists(_.cost <= currentState.state) {
    // do nothing, eliminate currentState because it's slow.
  } else {
    if (!fastStates.contains(currentState.formatToken)) {
      // currentState is the fastest state to reach this token.
      fastStates.update(currentState.formatToken, currentState)
    }
    // continue with algorithm
  }
}
```

Observe that this algorithm is transparent to the Router. No special annotations are required from Splits.

### 3.4.5 escapeInPathologicalCases

Alas, despite our best efforts to keep the search space small, some inputs can still trigger exponential running times. The `escapeInPathologicalCases` optimization is our last resort to handle such pathological inputs. How do we detect that the search has encountered such a challenging input?

We detect the search space is growing out of bounds by tallying the number of visits per token. If we visit the same token $N$ times, we can estimate the current branching factor to be around $log_2(N)$. In scalafmt, we tune $N$ to be 256 so that the best-first search can split into two or more paths for up to 8 tokens. When a token has been visited more than 256 times, we trigger the `escapeInPathologicalCases` optimization. In the following paragraphs section, we present two alternative fallback strategies: *leave unformatted* and *best-effort*.

The simplest and most obvious fallback strategy is to leave the pathologically nested code unformatted. This can be implemented by backtracing to the first token of the current statement and then reproduce the formatting input up to the last token of said statement. This method is guaranteed to run linearly to the size of the input. The responsibility is left to the software developer to a manually format her code, removing all the benefits of code formatting. However, in some cases the software developer may prefer to get a decent yet suboptimal formatting output.

The best-effort fallback strategy applies heuristics to give a decent but suboptimal formatting output. When a token is visited for the 256th time, we select two candidate states from the search queue and eliminate all other states. The first state is the fastest state — the state that has reached furthest into the token stream — that is not bound a prohibitive single line policy. Single line policies are policies that eliminate newline Splits. The second state is the current state — the slow state that visited the token for the 256th time. The intuition is that the fast state has good formatting output so far but for is stuck on a challenging token for some reason. The slow may paid a hefty penalty early causing it to move slowly but maybe the early penalty will yield a better output in the end.

Algorithm 6 shows an example of the best-effort strategy can be implemented as an extension to algorithm 1. The `isSafe` method on State

**Algorithm 6:** best-effort fallback strategy

```
var fastestState: State
val visits: mutable.Map[FormatToken, Int].withDefaultValue(0)
while (Q.nonEmpty) {
  val currentState = Q.pop
  visits.update(currentState.formatToken, 1 + visits(currentState.formatToken))
  if (currentState.length > fastestState.length && currentState.isSafe) {
    fastestState = currentState
  }
  if (visits(currentState.formatToken) == MAX_VISITS_PER_TOKEN) {
    Q.dequeueAll
    Q += fastestState
    Q += currentState
    visits.clear()
  } else {
    // continue with algorithm
  }
}
```

returns true if the state contains prohibitive policies, derived from annotated metadata in Splits from the Router. Observe that this algorithm will reapply the best-effort fallback until the search reaches the final token. In scalafmt, we bound the number of this can happen with a final fallback to the unformatted strategy.

The unformatted and best-effort fallback strategies offer different trade-offs. The unformatted strategy works well in a scenario where a software developer is available to manually fix formatting errors. The best-effort strategy works well on computer generated code where just a modicum of formatting still greatly aid the legibility of the code. Unfortunately, as we discuss in section 4.4, we struggled to guarantee idempotency using the best-effort strategy. This limitation renders the best-effort strategy useless in environments where code formatters are used to enforce a consistent coding style across a codebase.

### 3.5 FormatWriter

Recall from figure 2, the FormatWriter receives splits from the best-first search and produces a final output presented to the user. In addition to reifying Splits, the FormatWriter runs three post-processing steps:

*docstring formatting*, *vertical alignment* and *stripMargin alignment*.

### 3.5.1 Docstring formatting

Docstrings are used by software developers to document a specific part of code. Like in Java, docstrings in Scala start with the `/**` pragma and end with `*/`. However, unlike in Java, the Scala community is split on whether new lines inside docstrings should align by the first or the second asterisk. The official Scala Style Guide[8] dictates that new lines should align by the second asterisk while the Java tradition is to align by the first asterisk. The Scala.js[10] and Spark[45] style guides follow the Java conventions. To accommodate all needs, scalafmt allows the user to choose either style. To enforce that the asterisks are aligned according to the user's preferences, the FormatWriter rewrites docstring tokens using simple regular expressions.

### 3.5.2 stripMargin alignment

The Scala standard library adds a `stripMargin` extension method on strings. The method helps Scala developers write multiline interpolated and regular strings literals. Listing 38 shows an example usage of the `stripMargin` method.

**Listing 38: stripMargin example**

```
1 object StripMarginExample {
2   """Multiline string are delimited by triple quotes in Scala.
3     |You can write as many lines as you want.""".stripMargin
4 }
```

After calling the method, the indentation and `|` character on line 3 are conveniently stripped away. However, the hard-fought indentation can easily be lost when the string is moved up or down a scope during refactoring. Scalafmt can automatically fix the issue. In the FormatWriter, scalafmt rewrites string literals to automatically align the `|` characters with the opening triple quotes `"""`. This setting is disabled by default since scalafmt has only syntactic information and cannot determine if the `stripMargin` invocation calls the standard library method or a user-defined method.

### 3.5.3 Vertical alignment

# 4 Tooling

## 4.1 Heatmaps

## 4.2 Traceability

## 4.3 Configuration

### 4.3.1 maxColumn

### 4.3.2 binPacking

### 4.3.3 vertical alignment

## 4.4 Testing

## 4.5 Unit tests

## 4.6 Property based tests

### 4.6.1 AST Integrity

### 4.6.2 Idempotency

## 4.7 Regressions tests

# 5 Evaluation

## 5.1 Micro benchmarks

## 5.2 Adoption

# 6 Discussion

## 6.1 Future work

## 6.2 Conclusion

# References


[1] *Akka*. URL: http://akka.io/ (visited on 05/29/2016).

[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] Eugene Burmako. "Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming". In: *Proceedings of the 4th Workshop on Scala*. ACM. 2013, p. 3.

[5] Eugene Burmako. *scala.meta*. http://scalameta.org/. (Accessed on 06/03/2016). Apr. 2016.

[6] *CodeReviewComments for golang*. https://github.com/golang/go/wiki/CodeReviewComments. (Accessed on 06/01/2016). Oct. 2015.

[7] Daniel Jasper. *clang-format - Automatic formatting for C++*. https://www.youtube.com/watch?v=s7JmdCfI__c. (Accessed on 06/02/2016).

[8] David Copeland Daniel Spiewak. *Scala Style Guide*. http://docs.scala-lang.org/style/. (Accessed on 06/06/2016).

[9] Edsger W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271. URL: http://www.springerlink.com/index/uu8608u0u27k7256.pdf (visited on 06/01/2016).

[10] Sébastien Doeraene. *Scala.js Coding Style*. 2015. URL: https://github.com/scala-js/scala-js/blob/master/CODINGSTYLE.md (visited on 05/28/2016).

[11] Stuart Dreyfus. "Richard Bellman on the birth of dynamic programming". In: *Operations Research* 50.1 (2002), pp. 48–51.

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

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

[14] 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).

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

[16] Robert Griesemer. *gofmt - The Go Code Formatter*. https://golang.org/cmd/gofmt/. (Accessed on 06/01/2016). June 2009.

[17] 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).

[18] Li Haoyi. *sourcecode - Scala library providing "source" metadata to your program*. https://github.com/lihaoyi/sourcecode. (Accessed on 06/04/2016). Feb. 2016.

[19] 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).

[20] 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).

[21] Daniel Jasper. *clang-format*. Mar. 2014. URL: http://llvm.org/devmtg/2013-04/jasper-slides.pdf (visited on 04/20/2016).

[22] Daniel Jasper. *ClangFormat*. 2013. URL: http://clang.llvm.org/docs/ClangFormat.html (visited on 06/01/2016).

[23] 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).

[24] Jon Kleinberg and Éva Tardos. *Algorithm design*. Pearson Education India, 2006.

[25] 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).

[26] 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).

[27] Bob Nystrom. *dart_style - An opinionated formatter/linter for Dart code*. Sept. 2014. URL: https://github.com/dart-lang/dart_style (visited on 06/01/2016).

[28] Bob Nystrom. *The Hardest Program I've Ever Written*. Sept. 2015. URL: http://journal.stuffwithstuff.com/2015/09/08/the-hardest-program-ive-ever-written/ (visited on 04/14/2016).

[29] 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).

[30] 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).

[31] Bruno CdS Oliveira, Adriaan Moors, and Martin Odersky. "Type classes as objects and implicits". In: *ACM Sigplan Notices*. Vol. 45. 10. ACM. 2010, pp. 341–360.

[32] 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).

[33] Judea Pearl. "Heuristics: intelligent search strategies for computer problem solving". In: (1984). URL: http://www.osti.gov/scitech/biblio/5127296 (visited on 06/01/2016).

[34] 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).

[35] Matt Russell. *Scalariform*. 2010. URL: http://scala-ide.org/scalariform/ (visited on 05/28/2016).

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

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

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

[39] 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).

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

[41] 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).

[42] 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).

[43] 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).

[44] 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).

[45] Reynold Xin. *Spark Scala Style Guide*. Mar. 2015. URL: https://github.com/databricks/scala-style-guide (visited on 06/01/2016).

[46] Phillip M. Yelland. *A New Approach to Optimal Code Formatting*. Tech. rep. Google, inc., 2016. URL: http://research.google.com/pubs/archive/44667.pdf (visited on 04/20/2016).