scalafmt: opinionated code formatter for Scala

Ólafur Páll Geirsson

School of Computer and Communication Sciences

A thesis submitted for the degree of Master of Computer Science at École polytechnique fédérale de Lausanne

June 2016

Responsible
Prof. Martin Odersky
EPFL / LAMP

Supervisor
Eugene Burmako
EPFL / LAMP

Abstract

Code formatters bring many benefits to software development such as enforcing a consistent coding style across teams, more effective code reviews and enabling automated large-scale refactoring. Still, code formatters can be tricky to get right. This thesis addresses how to develop a code formatter for the Scala programming language. We present scalafmt, an opinionated Scala code formatter that captures many popular Scala idioms and coding styles. This thesis introduces language-agnostic algorithms and tooling that scalafmt uses to implement advanced features such as line wrapping and configurable vertical alignment. We have validated that these techniques work well in practice. Scalafmt has been installed over 6.500 times in only 3 months and several popular open-source libraries have chosen to reformat their codebases with scalafmt.

Contents

Listings

List of Algorithms

List of Figures

List of Tables

1 Introduction

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

```
object MyApp extends App {
  Initialize(context, config(port(
    "port.http"),
    settings + custom))
}
```

Listing 2: Formatted code

```
object MyApp extends App {
  Initialize(
      context,
      config(port("port.http"),
             settings + custom))
}
```

Both snippets represent the same application. The only difference lies in where the programmer has chosen to break lines. Characters such as spaces and line breaks that do not affect the execution of the program are syntactic trivia. Although syntactic trivia has no meaning for the execution of the program, listing 2 is arguably easier to understand, maintain and extend for the software developer. A code formatter is a tool that automatically converts a program such as in listing 1 into a readable and maintainable program such as in listing 2. Code formatting brings several benefits to software development.

Code formatting enables automated large-scale refactoring. Google used ClangFormat[21], a C++ code formatter, in the process of migrating a large legacy C++98 codebase to use the modern C++11 standard[46]. The automatically refactored code was formatted with ClangFormat to ensure that it adhered to Google's strict coding style[14]. Similar migrations can be expected in the near future for the Scala community once new dialects, such as Dotty[36], gain popularity.

Code formatting is valuable in collaborative coding environments. The Scala.js project[40] has over 40 contributors and the Scala.js coding style[10] — which each Scala.js contributor is expected to know by heart — is defined at a whopping 2.600 word count. 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 members have raised this issue. ENSIME[12] is a popular Scala interaction mode for text editors such as Vim and Emacs. Sam Halliday, an ENSIME maintainer, 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."[16]. Akka[1] is a popular concurrent and distributed programming library for Scala with over 300 contributors. 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.".[22].

With scalafmt, we hope to relieve Scala developers from the burden of manipulating syntactic trivia so they can instead direct their full attention to writing correct, maintainable and fast code.

1.1 Research objective

What algorithms and data structures allow us to develop a code formatter for the Scala programming language with a *maximum line length setting*, *opinionated setting*, *vertical alignment* and *good performance*? Those features are defined as follows:

- **Maximum line length setting**: a code formatter with a maximum line length setting ensures that each line in the formatted output contains no more than a certain number of characters. Many coding styles enforce a maximum line length to ensure code is readable from different environments such as small split screens and code review interfaces.
- **Opinionated**: An opinionated setting is a prerequisite to enforce a uniform coding style. An opinionated code formatter takes liberty to disregard line breaks and other formatting decisions in the original source input to ensure that formatted source files follow the same line breaking conventions.
- **Vertical alignment**: vertical alignment is a formatting convention where redundant whitespace is added before a token to align it on the same vertical column as similar tokens from other lines. Many Scala coding styles enforce vertical alignment to enhance code readability.
- **Performance**: Users expect code formatters to run fast. In the most demanding settings, a code formatter needs to be able to format thousands of lines of code in at most a few hundred milliseconds.

1.2 Contributions

The main contributions presented in this thesis are the following:

- language-agnostic algorithms and data structures to implement line wrapping with a maximum line length setting — in our opinion, the most challenging part of developing a code formatter — as well as configurable vertical alignment. This work is presented in section 3.
- methods to optimize and test the developed algorithms. This work is presented in section 4.
- practical validation of developed algorithms and methods with the scalafmt code formatter. The empirical results of this validation are presented in section 5.

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[31] 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[39] shows that LLVM may become yet another viable target platform for Scala developers.

Scala is a popular programming language. The Scala Center — a not-for-profit organization focused on Scala open-source and education — estimates that more than half a million developers use Scala[30]. Large organizations such as Goldman Sachs, Twitter, IBM and Verizon rely on Scala code to run business critical applications. The 2015 Stack Overflow Developer Survey shows that Scala is the 6th most loved technology and 4th best paying technology to work with[42]. The popularity of Apache Spark[2], a cluster computing framework for large-scale 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 a single large statement over of multiple small statements.

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
```

2.1.1 Higher order functions

Higher order functions (HOFs) are a common concept in functional programming languages as well as mathematics. HOFs are functions that can take other functions as arguments and 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]` instance. The function call `f(x)` is in fact sugar for the method call `f.apply(x)` on the `Function[Int, Int]` instance. Listing 4 shows an equivalent program to listing 3, without using syntactic conveniences. Observe that the body of `twice` was expressed as a single statement in line 1 of listing 3 but as two independent statements in listing 4.

2.1.2 Term blocks

Scala allows term blocks to appear anywhere in a Scala code. A term block is a sequence of statements wrapped by curly braces `{}`. Listing 5 shows two examples of term blocks. Variables bounds inside a term block do not escape the block. Therefore, the variable `y` can be assigned both inside the first block as well as to the return value of the `function` call. The lightweight syntax to create term blocks in Scala make them a popular feature among Scala developers. Observe that without term blocks, the second argument to

Listing 5: Term blocks

```
val x = { // { opens a new blockk
  val y = 1
  y + 2
}
val y = function(argument1, {
  val argument2 = 2
  argument2 + 3
}, argument3)
```

Listing 6: 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))
```

`function` would be defined externally. Instead, the second argument is defined inline making the entire `function` call significantly bigger.

2.1.3 SBT

SBT[38] 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 6 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 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, most metaprogramming facilities relied on Scala compiler internals. This had several severe limitations such as too-eager desugaring resulting in loss of syntactic

Listing 7: 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] // Parse error: missing newline
> dialects.Scala211(
    """lazy val root = project""").parse[Source] // Parse error: no class/...
```

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, high-fidelity parsing 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, a disallowed feature in regular Scala programs. Then, to format SBT files requires either depending on the SBT parser or reimplementing its parsing logic. To add insult to injury, top-level statements must be separated by a blank line if you use SBT version 0.13.6 or lower; a restriction that was lifted in SBT 0.13.7. Listing 7 shows how scala.meta dialects 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 a string to strip off all insignificant syntactic details. Listing 8 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.

Node types in scala.meta trees 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, comments and other syntactic trivia. 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.

Listing 8: Serializing scala.meta trees

```
> import scala.meta._
> """ object Main extends App { self =>
        println(s"Hello $self!") // This is a comment
      }""".parse[Source].get.structure // comment
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")))))))))))
"""
```

High-fidelity parsing in scala.meta has been essential for scalafmt because we can't lose critical syntactic details such as whether for-comprehensions are used over `flatMap` method calls.

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 exhaustivity checking in the Scala pattern matcher and enabled design pattern for the *Router* explained in section 3.3.1.

2.3 Code formatting

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

2.3.1 Natural language

The science of displaying aesthetically pleasing text dates back as early as 1956[18]. The first efforts involved inserting carriage returns in natural language text. Until that time, writers had been responsible for manually

[1] This thesis uses the term *code formatting* over *pretty printing*. According to Hughes[19], 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.

providing carriage returns in their documents before sending them off for printing. The motivation behind automating this process 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[24] for LaTeX, a popular typesetting program among scientific circles. LaTeX is 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, 23].

2.3.2 ALGOL 60

Scowen[41] developed SOAP in 1971, a code formatter for the programming language 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 maximum 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[13] explored code formatting algorithms for LISP[27] programs. LISP is a family of programming languages and is famous for its parenthesized prefix notation. Listing 9 shows a program in LISP to calculate factorial numbers. The simple syntax and extensive use of parentheses as delimiters makes 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 algorithm tries to first obtain a *linear-format*, i.e. fit remaining body of that node on a single line,

Listing 9: A LISP program

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

with a fallback to *standard-format*, i.e. each child of that node is put on a separate line aligned by the first child. 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 are relevant concerns for modern code formatters.

2.3.4 Language agnostic

Derek C. Oppen pioneered the work on language agnostic code formatting in 1980[34]. 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. This feature is typically not a concern for modern code formatters. Moreover, 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 in 1996 that generates a formatter given a context-free grammar[44]. 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 in 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[19]. 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 wide use, including in the Glasgow Haskell compiler. Wadler[45] and Chitil[43] 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 existing source code.

2.3.5 Go

`gofmt`[15] 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 10 shows a Go program that uses the same layout as the "unformatted" code (listing 1) in the introduction.

Listing 10: Gofmt example input/output

```
package main

func main() int {
  Initialize(config, port(get(
    "port.http"),
    settings+custom))
}
```

The output of running `gofmt` through listing 10 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 Scala

Scalariform[37] 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[47] 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 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 and indented by 4 spaces. Listings 11 and 12 shows an example of bin packing enabled and disabled, respectively.

Listing 11: Bin-packing

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

Listing 12: 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 12 to the code in listing 11.

2.3.7 C-family

Daniel Jasper triggered a new trend in optimization based coded formatters with the release of *ClangFormat*[20] in 2013. ClangFormat is developed at Google and can format an impressive number of languages: 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*.

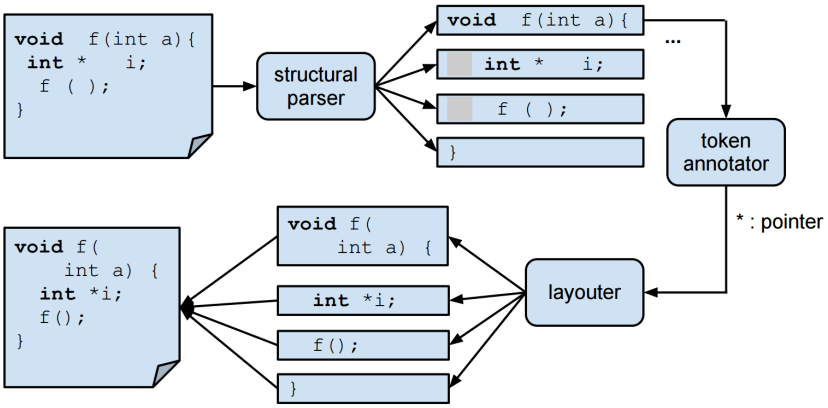

Figure 1: ClangFormat architecture

ClangFormat employs a structural parser to split source code into a sequence of *unwrapped-lines*. An unwrapped line is a statement that should 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 supporting several programming languages, 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 13: Unformatted 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 14: 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 13 shows an offensively formatted C++ code snippet. Listing 14 shows the same snippet after being formatted with ClangFormat. ClangFormat is opinionated because by default 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.

2.3.8 Dart

Dartfmt[28] was released in 2014 and follows the optimization based trend initiated by ClangFormat. Dartfmt 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 a blog post[29]. 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*[35], 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 15 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.

Listing 15: Avoid dead ends

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

2.3.9 R

The most recent addition to the optimization based formatting trend is `rfmt`[48], 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[19] 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 16 and 17 shows two different layouts to format an argument list.

Listing 16: Line block

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

Listing 17: Stack block

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

In this case, we prefer the line block from listing 16 since it requires fewer lines. However, our preference changes if the function name is longer as is shown in listings 18 and 19.

Listing 18: Line block

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

Listing 19: Stack block

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

Here, we clearly prefer the stack block in listing 19. Listing 20 shows how we use the 6 fundamental blocks in the `rfmt` combinator algebra to express the choice between these two formatting layouts.

Listing 20: 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(")")).
```

The variable f denotes the function name and $a_1, ...a_m$ denotes the argument list. Observe that listing 20 does not express how to find the optimal layout.

To find an optimal layout, `rfmt` employs a novel indexing scheme. 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 alleviates exponential growth 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[48] 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).

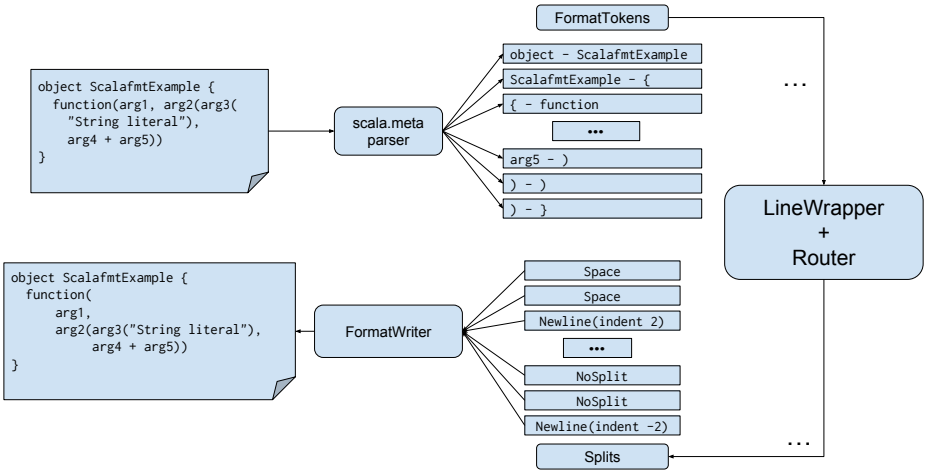

Figure 2: Scalafmt architecture

3 Algorithms

This chapter describes how scalafmt formats Scala code. We will see that scalafmt's design is inspired by ClangFormat and dartfmt. However, our design has been heavily adapted to take advantage of many Scala programming idioms.

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 21 shows the definition of the FormatToken data type.

Listing 21: 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 22 shows the definition of Decision.

Listing 22: Decision definition

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

The *splits* member represents the possible splits that the LineWrapper can choose for *formatToken*.

3.2.3 Policy

A *Policy* is an enforced formatting layout over a region. Listing 23 shows the definition of Policy.

Listing 23: Policy definition

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

3.2.4 Indent

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

Listing 24: 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 24 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 for 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*.

3.2.5 Split

A *Split* represents a (possibly empty) whitespace character to be inserted between two non-whitespace tokens. Listing 25 shows the rather intricate definition of the Split data type[4].

Listing 25: Split definition

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

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

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 *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 the best-first search. Listing 26 shows the definition of the State class and companion object.

Listing 26: 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` vector. 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.

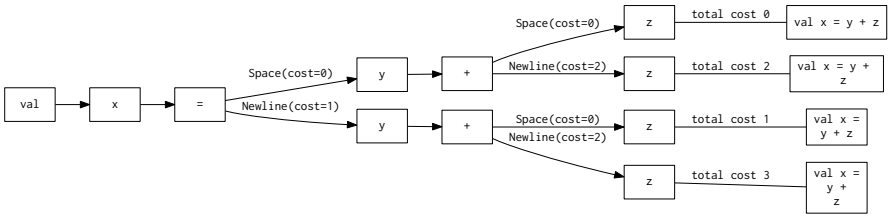

Figure 3: Example graph produced by Router

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 this figure, the Router is the planner that has chosen to open up multiple branches at = and + and only one branch for the remaining tokens. 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 27 shows how we can pattern match on a FormatToken and produce Splits.

Listing 27: 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 28 shows an example where the compiler issues a warning.

Listing 28: 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[32] come to the rescue.

The source file line number of where a Split is instantiated is automatically attached on each Split. Remember in listing 25 that the Split case class had an implicit member of type `sourcecode.Line`. Sourcecode[17] 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 29 shows how this works.

Listing 29: 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[35] 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

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.
  ???
}
```

search always chooses the cheapest splits and the 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 30 shows a tiny input that triggers the search to explore over 8 million states.

Listing 30: Exponential running time

[5] We make heavy use of mutation since graph search algorithms typically don't lend themselves well to functional programming principles.

```
// 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[6] 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 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 good 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 31.

Listing 31: Two independent statements

```
// Column 60                                                  |
statement1(argument1, argument2, argument3, argument5, argument6)
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

[6] Benchmarks reveal the best-first search visits on average one state per 10 microseconds

is done. With an empty queue, we ensure the search backtracks only as far back

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
  }
  // ...
}
```

as is needed. The `statementStarts` variable contains all tokens that begin a new statement. To collect those tokens, we traverse the syntax 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 32 shows an example where this happens.

Listing 32: 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 33 shows an example where this happens and we trigger the `recurseOnBlocks` optimization.

Listing 33: 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.

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 25 that a Split has an `optimalToken` member. Listing 34 shows the definition of OptimalToken. When the best-first search encounters a Split with a defined OptimalToken, the

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 34: OptimalToken definition

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

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 35 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 35: 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 34. 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 36 shows two formatting solutions that the Router has labelled as equally expensive. However, the fast solution is explored first by the best-first search and, hence, we call it *faster*.

Listing 36: 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
```

The `pruneSlowStates` ensures that fast solutions are prioritized over slow solutions. Of course, the Router could have assigned different costs to the line break after `g +` and `c +`. However, our experience was that such as solution would introduce unnecessary complexity in the design of the Router. Instead, the `pruneSlowStates` is able to 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 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

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
  }
}
```

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 4.2, 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 the 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[47] style guides follow the Java convention. 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. This is implemented with 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 string literals. Listing 37 shows an example usage of the `stripMargin` method.

Listing 37: 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 removed. However, the hard-fought indentation on the pipe can easily be lost when the string is moved up or down a scope during refactoring. Scalafmt can automatically fix this 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 requires semantic information to confidently determine if the `stripMargin` invocation calls the standard library method or a user-defined method.

3.5.3 Vertical alignment

It turns out that vertical alignment is incredibly popular in the Scala programming community. Vertical alignment is a formatting convention where redundant spaces are inserted before a token to put on the same vertical column as related tokens from other lines. Listing 38 shows an example of vertical alignment.

Listing 38: Vertical alignment example

```
object VerticalAlignment {
  x match {
    // Align by => and -> and //
    case 1  => 1  -> 2  // first
    case 11 => 11 -> 22 // second

    // Blank lines separate alignment blocks.
    case ignoreMe => 111 -> 222
  }

  def name   = column[String]("name")
  def status = column[Int]("status")
  val x  = 1
  val xx = 22

  libraryDependencies ++= Seq(
      "org.scala-lang" % "scala-compiler" % scalaVersion.value,
      "com.lihaoyi"    %% "sourcecode"    % "0.1.1"
  )
}
```

Observe that if we add a new library dependency that has a long name in the first column, we must add additional spaces after `"org.scala-lang"` and `"com.lihaoyi"` to preserve the vertical alignment. Many software developers speak against vertical alignment for this reason, as well as several other reasons. Nevertheless, the lack of vertical alignment in the initial release of scalafmt was a hindrance for user-adoption. Configurable vertical alignment was added to the 0.2 release of scalafmt.

Vertical alignment is implemented in the FormatWriter as an extension to the reification of Splits. Instead of reifying a Space into a single space literal, the FormatWriter builds a `tokenAligns: Map[Split, Int]` which specifies the number of additional spaces to add for each Space. Algorithm 7 shows a simplification of how the actual algorithm[7] constructs the `tokenAligns` map. The running time of this algorithm is linear to the number of tokens. In a nutshell, the algorithm builds blocks of lines that can be vertically aligned. Blocks are separated by blank lines or mismatching candidates in subsequent lines. As demonstrated in listing 38, key features of this algorithm include that users can configure arbitrary symbols to align by and lines can contain multiple columns of vertically aligned tokens. This algorithm has two main limitations. Firstly, all lines in a block must have an equal number of matching columns. Secondly, the algorithm does not infer vertical alignment forcing the user to explicitly configure which tokens should align. The first limitation can be

[7] The actual implementation is 130 lines of code, including helper methods. See https://git.io/voIG4

Algorithm 7: Vertical alignment, simplified algorithm

```
case class FormatLocation(formatToken: FormatToken, split: Split, state: State)
/** Returns true if location is eligible for vertical alignment */
def isCandidate(location: FormatLocation): Boolean
/** Returns true if all vertical alignment candidates in a and b match */
def allColumnsMatch(a: Array[FormatLocation], b: Array[FormatLocation]): Boolean
/** Returns map where the keys are (0 to block.length) and values are the
    corresponding column index where all candidates should align */
def getMaxColumns(block: Vector[Array[FormatLocation]]): Map[Int, Int]

def getAlignTokens(
    locations: Array[FormatLocation],
    alignConfiguration: Map[String, Regex]): Map[Split, Int] = {
  val finalResult = Map.newBuilder[Split, Int]
  val lines: Array[Array[FormatLocation]] = getLines(locations)
  var block = Vector.empty[Array[FormatLocation]]
  for (formatLocations <- lines) {
    val candidates: Array[FormatLocation] = formatLocations.filter(isCandidate)
    if (block.isEmpty) { // Starting a new block.
      if (candidates.nonEmpty) block = block :+ candidates
    } else {
      if (columnsMatch(block.last, candidates)) {
        block = block :+ candidates
      } else { // release alignment
        val maxColumns = getMaxColumns(block)
        for (line <- block) {
          for ((tokenToAlign, columnIndex) <- line.zipWithIndex) {
            finalResult += (tokenToAlign.split,
                            maxColumns(columnIndex) - tokenToAlign.state.column)
          }
        }
      }
    }
  }
}
```

addressed by extending the implementation of the `allColumnsMatch` method. The second limitation can be addressed by treat all tokens as candidates (i.e., remove the call to `isCandidate`) and extend `allColumnsMatch` to include tokens that are aligned in the original source.

3.5.4 Conclusion

This section introduced the data structures and algorithms that scalafmt uses to implement line wrapping and vertical alignment. Similar to dartfmt and ClangFormat, scalafmt models line wrapping like a graph search problem. Non-whitespace tokens represent nodes and each potential split forms a weighted edge to the next token. The edge with the lowest associated cost represents the best formatting output and the edge with the highest represents the least favorable formatting output. Scalafmt uses best-first search to find the best formatting output (i.e., the cheapest path from the first token in the input to the last token in the input). However, scalafmts implementation deviates quickly from there by introducing the Split, Policy and Router abstractions. The motivation for coming up with our own abstractions was to make scalafmt approachable for Scala developers to maintain and extend. For example, the use of partial functions in the Policy data type follows a unique Scala idiom that translates poorly to Dart or C++. Likewise, we believe that translating dartfmts concept of Rules — which relies heavily on mutation — would come at the price of less idiomatic Scala code. Given the extensive use of higher order functions and blocks in Scala, we struggled to find a robust way to break a source files into a sequence of unwrapped lines like ClangFormat does. Nevertheless, these abstractions are different means to the same end. We leave it to the judgment of the reader to assess which concepts are more powerful or intuitive to understand.

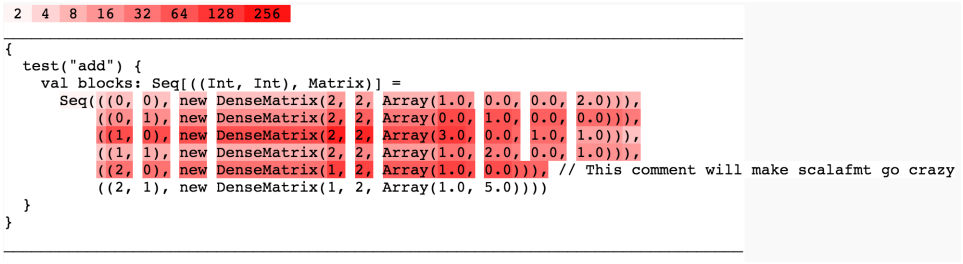

Figure 4: Example heatmap with 5.121 visisted states

4 Tooling

This chapter describes the tools that we developed while designing an implementing algorithms for scalafmt. These tools were indispensable in giving us confidence that our algorithms worked as intended.

4.1 Heatmaps

Section 3.4 introduces several extensions to algorithm 1 that were required to get good performance for scalafmt. In general, the extensions involved eliminating search states. To identify code patterns that triggered excessive search growth, we developed heatmaps.

Heatmaps are a visualization that displays which code regions are most frequently visited in the best-first search. Figure 4 shows an example heatmap. The intensity of the red color indicates how often a particular token was visited. A token highlighted by the lightest shade of red was visited twice while a token highlighted by the darkest shade of red was visited over 256 times. This figure demonstrates several of the optimizations discussed in section 3.4. Firstly, thanks to the `dequeueOnNewStatements` optimization, the background is plain white up to the `Seq`. The `Seq` gets visited twice, once when there's a space after the = and once when there's a newline. Secondly, due to the OptimalToken optimization, when the search gets into trouble it backtracks to the tuple `(0, 0)` instead of the `Seq[((Int, Int), Matrix)]` type signature. Finally, because of the strategically placed comment at the end that exceeds the column limit, the search space grows out of bounds on the fourth argument triggering the `escapeInPathologicalCases` best-effort fallback. Without heatmaps, it would be a much greater challenge to get these insights. However, these heatmaps gave us limited insights in how our changes affected the search

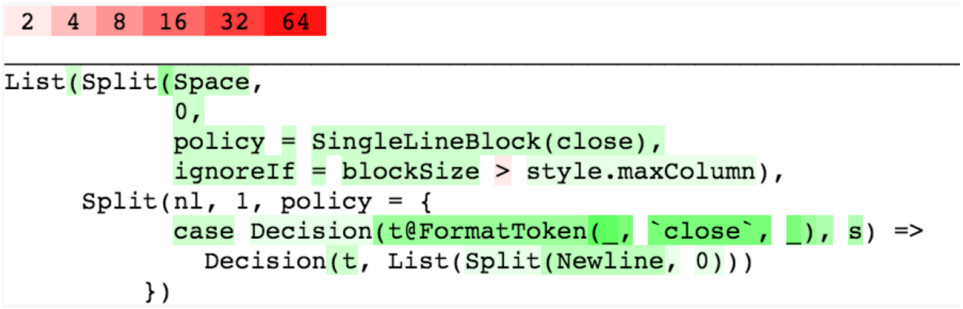

Figure 5: Example diff heatmap

space in the best-first search.

We developed an extension to heatmaps that allows us to visually compare the difference in search space between two versions of scalafmt. Figure 5 shows an example of such a diff report. The green background indicates that the new version of scalafmt makes fewer visits to those regions. Observe that the > operator has a background with a light shade of red. This means that the operator was visisted more often in the new scalafmt version. A price well worth paying considering the overall shrink in search space. To produce diff heatmaps, we first persist to a database the statistics report needed to generate a single heatmap after each test run. Then, we generate the diff heatmap by fetching two reports and calculating the difference in visits per token. If the difference is negative for a particular token — meaning we visited said token fewer times — the background is highlighted green, otherwise red.

4.2 Property based testing

Property based tests played a vital role in the development of scalafmt and gave us confidence that the algorithms from section 3 behave well against the real world input. Typically, property based tests run again randomly generated input. However, generating random source files which might be unrepresentative for human written code. Instead, we chose to collect a large sample of 1.2 million lines of code from open source Scala projects available online. The sample was compressed into a 23mb zip file[8]. Our test suite would download the sample and test three properties: *can-format*, *AST integrity* and *idempotency*.

[8] See https://github.com/olafurpg/scalafmt/releases/download/v0.1.4/repos.tar.gz

4.2.1 Can-format

The can-format property simply says that if the Scala compiler's parser is able to parse the source input file, then scalafmt should be able to format the source file. Although this may seem like a trivial property, it was by far the most effective property at finding bugs in scalafmt. Most commonly, comments in the most unexpected placed would cause the best-first search to not reach the last token in the input. An overly strict Policy was usually the culprit of such bugs, which was easy to fix thanks to our tracing techniques described in section 3.3.1.

4.2.2 AST integrity

The AST integrity property says that the abstract syntax tree of the formatted source file should be identical to the abstract syntax tree of the original input. Recall from section 2.2 that scala.meta trees can be serialized into strings. We leverage this feature to test AST integrity. Algorithm 8 shows the code needed to test AST integrity. This property catched several critical bugs. For example, in

Algorithm 8: AST integrity property

```
1 import scala.meta._
2 forAll { (code: String) =>
3   val beforeAST = code.parse[Source].show[Structure]
4   val afterAST = Scalafmt.format(code).parse[Source].show[Structure]
5   beforeAST == afterAST
6 }
```

one case, scalafmt inserted a newline after the keyword `return`, breaking the semantics of the original source code. Moreover, this property highlighted the danger of enabling the `stripMargin` alignment. Since the `stripMargin` modified the contents of regular and interpolated string literals, the AST of the formatted output changed. Knowing that scalafmt preserves the AST of the input code gives us great confidence that scalafmt will at least not introduce bugs in our users code.

4.2.3 Idempotency

The idempotency property says that if the output of formatting a source file twice should be identical to the output of formatting it once. This property is

critical for scalafmt to be used as part of any continuous integration setup. It is not at all obvious that the algorithms in section 3 fulfill the idempotency property. Our experience reveals that it is in fact very easy to accidentally introduce non-idempotent formatting rules in the Router. We did not test for idempotency until the 0.2.3 release, after users reported non-idempotent formatting behavior in scalafmt. Yet, even after we started testing against idempotency in our comprehensive test-suite, we continued to receive issues with non-idempotent formatting. This time it appears that the `escapeInPathologicalCases` strategy from section 3.4.5 was the culprit. For the next release, we plan to disable `escapeInPathologicalCases` by default in favor use its safer alternative. It turns out that 1.2 million lines of code is not a large enough sample to catch all property bugs.

5 Evaluation

Code formatting is inherently a subjective topic. This introduces a challenge when evaluating a code formatter. In this chapter, we will present measurements that we believe show the success of scalafmt. We do not measure how well software developers perceive scalafmt formatted code. Instead, we will focus on *performance benchmarks* and *user adoption*.

5.1 Performance benchmarks

This chapter measures scalafmt's raw formatting performance. We first describe our test methodology and then present results from two different benchmarks: *macro* and *micro*.

5.1.1 Setup

The benchmarks are run on a Macbook Pro (Retina, 15-inch, Mid 2014) laptop with a quad-core 2.5 GHz Intel Core i7 processor, 256 KB L2 cache per core and 6 MB shared L3 cache. The laptop has 16 GB 1600 MHz DDR3 memory. The operating system is OS X El Capitan 10.11.5. We run the benchmarks from the scalafmt commit id aff5e794 compiled against Scala 2.11.7, running on JVM version 8, update 91. For accurate measurements, all benchmarks are run with the OpenJDK Java Microbenchmark Harness (JMH)[33]. JMH takes into account a variety of parameters that affect performance on the JVM. The sbt-jmh[26] plugin makes it easy to integrate JMH with a Scala project.

To repeat the benchmarks, execute the `run-benchmarks.sh` script in the root directory of the scalafmt project.

5.1.2 Macro benchmark

The macro benchmark is designed to get insight on how scalafmt performs in a continuous integration setup. For example, it is common to assert before code review that all source files are properly formatted. For this benchmark we format the entire Scala.js codebase. The codebase contains 915 source files and over 106 thousand lines of code, excluding blank lines and comments. For accurate measurements, we run five iterations of the macro benchmark. We

Benchmark	Cores	Score	Error	Units
Parallel.scalafmt	4	14.616	± 0.632	s/op
Parallel.scalariform	4	2.810	± 0.641	s/op
Ratio		5.20		
Synchronous.scalafmt	1	35.654	± 0.459	s/op
Synchronous.scalariform	1	5.951	± 0.135	s/op
Ratio		5.99		

Table 1: Results from macro benchmark.

compare the running time with Scalariform.

Table 1 shows the results from the macro benchmark. Scalafmt is almost 6x slower than Scalariform. Why is the performance gap so big? Is this gap acceptable for continuous integration setups?

We believe two factors contribute to the fact that scalafmt is 6x slower than Scalariform. Firstly, preliminary results from profiling scalafmt reveal that micro-optimizing scalafmt could yield great performance improvements. For example, over 30% of the formatting time is dedicated to a pre-processing step — unrelated to the best-first search — that could be accomplished with minimal overhead during parsing inside scala.meta. Other experiments indicate that scalafmt may speed up yet another 30% by upgrading to the latest release of scala.meta[9]. Secondly, scalafmt's formatting algorithm is more complex. Scalafmt may try thousands of different formatting layouts to find an optimal formatting output. In contrast, Scalariform's formatting algorithm is linear.

We believe the current performance is usable in a continuous integration setup, but would benefit greatly from performance improvements. The current performance is usable because a typical diff in a code review touches only a few source files, and definitely far from the 106 thousand lines of code that we format in this macro benchmark. On a smaller diffs, the gap between Scalariform and scalafmt is less pronounced. However, it is worth considering that continuous integration setups may not have access to the same powerful hardware as we do in this benchmarks. We estimate that scalafmt requires at least a 2-3x performance improvement to be integrated in continuous

[9] Scala.meta recently went through several non-source-compatible upgrades. This has made it difficult for scalafmt to keep up with the latest release.

25th	Median	Mean	75th	90th	95th	99th	Max
16	46	106	113	248	400	945	11723

Table 2: Percentiles of lines of code per file in micro benchmark.

integration setups with less powerful hardware and projects that contains large amounts of code.

5.1.3 Micro benchmark

The micro benchmark is designed to get insight on how scalafmt performs in an interactive software developer workflow. For example, many Scala developers configure SBT to reformat source files on every compilation. Before we run the benchmark, we must find out how many lines of code a typical source file contains.

We performed a small study to learn the size of a typical source file. We collected a sample of 3.2 million lines of code from 33 open source Scala projects. Table 2 shows the distribution of file sizes in our sample. Observe that over 90 percent of all files are rather small, or under 250 lines of code. Only one percent of files contain more than 1.000 lines of code. Still, we assume developers spend quite a lot of time editing such large files.

Using the results from our small study, we choose to run the micro benchmark on four files of varying sizes: small (~ 50 LOC), medium (~300 LOC), large (~1.000 LOC) and extra large (~4.500 LOC). To minimize error margins, we run 10 warmup iterations followed by 10 measured iterations. As in the macro benchmark, we compare the running time with Scalariform. The micro benchmark is single threaded.

Table 3 shows the results from the micro benchmark. No surprise, scalafmt is again slower than Scalariform. Is this performance gap acceptable for interactive software development?

We believe this performance is usable for occasional code formatting, but not suitable for a workflow that formats on every compilation. Amazon famously showed that sales decreased by 1 percent for every 100ms increase in page load time[25]. We believe similar principles apply to scalafmt, every additional millisecond decreases the utility of scalafmt. Still, we believe with scalafmt's formatting output is appealing enough to outweigh the slow performance. As

Benchmark	Score	Error	Units
ExtraLarge.scalafmt	1423.140	± 103.360	ms/op
ExtraLarge.scalariform	219.820	± 14.450	ms/op
Ratio	6.50		
Large.scalafmt	355.819	± 17.385	ms/op
Large.scalariform	39.324	± 3.395	ms/op
Ratio	9.05		
Medium.scalafmt	79.616	± 2.013	ms/op
Medium.scalariform	15.934	± 0.441	ms/op
Ratio	5.00		
Small.scalafmt	6.968	± 0.104	ms/op
Small.scalariform	1.176	± 0.025	ms/op
Ratio	5.93		

Table 3: Results from micro benchmark.

we'll discuss in the following section, our users seem to agree.

5.2 Adoption

Scalafmt has received quite some attention since its release in early March, three months ago. In this section we present the statistics we believe demonstrate that scalafmt is — despite its young age — already proving itself useful for the Scala community. All data points are as of June 9th, 2016.

5.2.1 Installations

Scalafmt has been installed over 6.500 times. Table 4 shows the installation numbers for each official distribution channel. IntelliJ is the Jetbrains plugin repository[10]. The numbers represent absolute download numbers, not unique users. Data is not available for how many users built scalafmt from source, so it is fair to estimate that the actual number of installation is even higher. Observe that v0.2.5 was released 22 days ago, meaning it has been installed 80 times on

[10] See https://plugins.jetbrains.com/plugin/8236?pr=

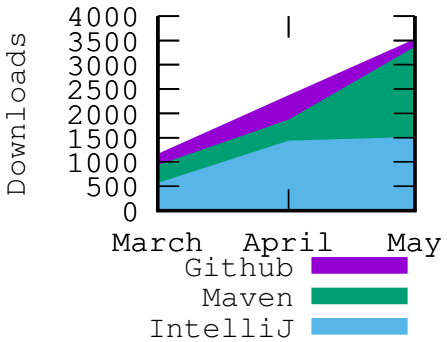

Figure 6: Scalafmt installations by month by channel

Table 4: Download numbers for scalafmt.

Channel	Version	Installations
IntelliJ	v0.2.5	847
	All	3.273
Maven	v0.2.5	788
	All	2.657
Github	v0.2.5	102
	All	929
Sum	v0.2.5	1.737
	All	6.859

average per day since its release. Extrapolating from the v0.2.5 installations numbers, we expect that scalafmt currently has roughly 1.000 active users.

Figure 6 shows the growth in installations by month. Observe that growth has doubled with each new month. Github represented proportionally many installs in the first month but only represents a small fraction of installation in May. Maven installations quadrupled in May, taking the lead from the IntelliJ plugin in April. We believe this increase in Maven installations is caused by projects installing the scalafmt SBT plugin for every test run in a continuous integration setup.

5.2.2 Other

We present interesting data points from a variety of disparate data sources:

- Several popular open-source libraries have reformatted their codebases with scalafmt. Table 5 shows an incomplete list libraries that have so far taken the jump. Observe that all libraries take advantage of vertical alignment. Moreover, each library customize on top of the base `default` style. It is worth mentioning that all libraries except Scala.js dom are relatively new. We believe more mature libraries are slower to adopt such new technology.

[11] https://github.com/scala-native/scala-native
[12] https://github.com/scala-js/scala-js-dom
[13] https://github.com/47deg/fetch
[14] https://github.com/paulp/psp-std

Project	Customized coding style
Scala Native[11]	`defaultWithAlign` base style, 80 character column limit, Java docstrings.
Scala.js dom[12]	`Scala.js` base style: 80 character column limit, vertical alignment on case arrows, bin packed arguments/parameters/parent constructors, Java docstrings.
Fetch[13]	`defaultWithAlign` base style, 100 character column limit, Scala docstrings.
psp-std[14]	Customized vertical alignment, 160 character column limit, 2 space continuation indent, spaces in import curly braces, Scala docstrings.

Table 5: Open source libraries that have reformatted their codebase with scalafmt and their customized settings.

- The scalafmt code repository has received contributions from 8 external contributors. Several of these contributions added non-trivial features to scalafmt, including new configuration flags and extensions to the Router.
- 34 unique users, excluding the author, have opened a total of 138 tickets on the scalafmt issue tracker.
- The scalafmt Gitter[15] instant messaging channel has 47 members. The channel is used to informally discuss bugs, new features and more.
- The user documentation website[16] has been visited 5.422 times with an average visit duration of 98 seconds.

[15] See https://gitter.im/olafurpg/scalafmt
[16] See http://scalafmt.org

6 Future work

Scalafmt is not free from issues. The main areas for improvements regard how to more soundly produce an optimal formatting layout and how to obtain better performance in an interactive developer workflow.

We experienced a long tail of problems while getting the best-first search to reach the last token for Scala code that we found in the wild. If the best-first search cannot reach the last token, scalafmt cannot format the file. Although the concept of policies does give a lot of flexibility to concisely express different formatting layouts, our experience is that it can be easy to create overly strict policies that eliminate all active search states. It is worth to explore more principal approaches on how to define formatting layouts so that we can guarantee that the search is sound and successfully completes every time. We believe `rfmts` approach of combining the convenience of combinators with algebraic properties and optimized dynamic programming for excellent performance opens an interesting venue to solve this problem.

Incremental formatting provides an opportunity to get enormous performance improvements in an interactive developer workflow. Instead of formatting an entire source file on every invocation, incremental formatting reuses output from a previous invocation to only reformat lines that have changed. We believe that incremental formatting could cut down the formatting time for a large source file by several orders of magnitude, for example from 2s to 20ms. This would result in a huge improvement in user experience. Users could configure their text editors to reformat on every key press, if they so please.

7 Conclusion

We set out to implement a code formatter for the Scala programming language that supports several important features: an opinionated setting, a maximum line length setting, vertical alignment and fast performance. We have presented data structures and algorithms that enabled us to develop scalafmt, a Scala code formatter that supports the first three required features and goes far towards achieving good performance. Benchmarks reveal that scalafmt can format over 100 thousand lines of code in only 15 seconds. However, scalafmt is still 6x slower than Scalariform, an alternative Scala code formatter. User adoption of scalafmt indicates that scalafmt's features are valuable and that the current performance is acceptable for many software developers. We believe there is plenty of room for improvements on making scalafmt go further and meet the needs of the even the most demanding users.

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 and Denys Shabalin. *scala.meta*. http://scalameta.org/. (Accessed on 06/06/2016). 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] 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).

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

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

[16] 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).

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

[18] 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).

[19] 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).

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

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

[22] 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).

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

[24] 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).

[25] Ron Kohavi and Roger Longbotham. “Online experiments: Lessons learned”. In: *Computer* 40.9 (2007), pp. 103–105.

[26] Konrad Malawski. *ktoso/sbt-jmh: "Trust no one, bench everything." - sbt plugin for JMH (Java Microbenchmark Harness)*. https://github.com/ktoso/sbt-jmh. (Accessed on 06/07/2016).

[27] 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).

[28] 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).

[29] 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).

[30] 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).

[31] 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).

[32] 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.

[33] *OpenJDK: jmh*. http://openjdk.java.net/projects/code-tools/jmh/. (Accessed on 06/07/2016).

[34] 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).

[35] 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).

[36] 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).

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

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

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

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

[41] 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).

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

[43] 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).

[44] 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).

[45] 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).

[46] 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).

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

[48] 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).