ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

scalafmt: opinionated code formatter for Scala

Ólafur Páll Geirsson

School of Computer and Communication Sciences

Master's Thesis

June 2015

Responsible
Prof. Martin Odersky
EPFL / LAMP

Supervisor
Eugene Burmako
EPFL / LAMP

Abstract

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

Contents

1 Introduction

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

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

Listing 1: Unformatted code

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


```

Listing 2: Formatted code

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

```

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

Code formatting enables large-scale refactoring. Google used ClangFormat[18], a code formatter, to migrate legacy C++ code to the modern C++11 standard[38]. ClangFormat was used to ensure that the refactored code adhered to Google's extensive C++ coding style[12]. Similar migrations can be expected in the near future for the Scala community once new dialects, such as Dotty[28], gain popularity.

Code formatting is valuable in collaborative coding environments. The Scala.js project[32] has over 40 contributors and the Scala.js coding style[7] contains over 2.600 words. Each contributor to the Scala.js project is expected to follow the coding style. Each contributed patch is manually verified against the coding style by the project maintainers. This adds a burden on both contributors and maintainers. Several maintainers of popular Scala libraries have expressed this sentiment. ENSIME[9] is a popular Scala interaction mode for text editor. Sam Halliday, a maintainer of ENSIME, says “I don’t have time to talk about formatting in code reviews. I want the machine to do it so I can focus on the design.”[14]. Akka[1] is another Scala library to build concurrent and distributed applications. Viktor Klang, a maintainer of Akka, suggests a better alternative “Code style should not be enforced by review, but by automate rewriting. Evolve the style using PRs against the rewriting config.”.[19]. With code formatters, software develoeprs can direct their full focus on writing correct, maintainable and fast code.

1.1 Whitespace style issues

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

Listing 3: Style issues

```
if  ( condition(  predicate)) {
    println("Goodbye" );
  System.exit(1) }
```

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

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

1.2 Contributions

The main contribution presented in this thesis are the following:

- scalafmt, a code formatter for the Scala programming language. At the time of this writing, scalafmt has been available for 3 months, it has been installed over 5.000 times and is already in use by several open source Scala libraries. For details on how to install and use scalafmt, refer to the scalafmt online documentation[10].
- algorithms and data structures to implement line wrapping under a maximum column-width limit. This work is presented in section 3.
- tools to develop and test code formatters. This work is presented in section 4.

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

2 Background

This chapter explains the necessary background to understand Scala and code formatting. More specifically, we motivate why Scala presents an interesting challenge for code formatters. We go into details on Scala's rich syntax and popular idioms that introduced unique challenges to the design of scalafmt. We follow up with a brief history on code formatters for both Scala as well as other programming languages. We will see that although code formatters have a long history, a new tradition of optimization based formatters started only recently in 2013.

2.1 Scala the programming language

Scala[25] 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[7], JavaScript has recently become a popular target platform for Scala developers. Even more recently, the announcement of Scala Native[31] shows that LLVM and may become yet another viable platform for Scala developers.

Scala is a popular programming language. The Scala Center estimates that more than half a million developers are using Scala[24]. Large organizations such as Goldman Sachs, Twitter, IBM and Verizon run Scala code in production systems. The 2015 Stack Overflow Developer Survey shows that Scala is the 6th most loved technology and 4th best paying technology to work with[34]. The popularity of Apache Spark[2], a cluster computing framework for large-scala data processing, has made Scala a language of choice for many developers and scientists working in big data and machine learning.

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

Listing 4: Higher order functions

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

Listing 5: Higher order functions without syntactic sugar

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

challenge to code formatting.

2.1.1 Higher order functions

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

Functions are first-class citizens in Scala. Consider listing 4. The method `twice` takes an argument `f`, which is a function from an integer to an integer. The method returns a new function that will apply `f` twice to an integer argument. This small example takes advantage of several syntactic conveniences provided by Scala. For example, in line 2 the argument `_ + 3` creates a new `Function[Int, Int]` object. The function call `f(x)` is in fact sugar for the method call `f.apply(x)` on a `Function[Int, Int]` instance. Listing 5 shows an equivalent program to listing 4 without syntactic sugar. Observe that what was expressed as a single statement in line 1 of listing 4 is expressed with multiple statements in lines 1 and 2 of listing 5.

2.1.2 Immutability

Functional programming encourages stateless functions which operate on immutable data structures and objects. An immutable object is an object that once initialized, cannot be modified. Immutability offers several benefits to software development in areas including concurrency and

Listing 6: Manipulating immutable list

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

Listing 7: Manipulating mutable list

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

testing. Listing 6 shows an example of manipulating an immutable list. Note that each `map` and `filter` operation creates a new copy of the list with the modified contents. The original list remains unchanged. Listing 7 show the equivalent operation using a mutable list. Observe that what was listing 6 is a single statement while listing 7 is multiple statements.

2.1.3 SBT build configuration

SBT[30] is an interactive build tool used by many Scala projects. SBT configuration files are written in `*.sbt` or `*.scala` files using Scala syntax and semantics. Although SBT configuration files use plain Scala, they typically use coding patterns which are different from traditional Scala programs. Listing 8 is an example project definition in SBT. Observe that the project is defined as a single statement and makes extensive use of

Listing 8: SBT project definition

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

symbolic infix operators. Due to the nature of build configurations, argument lists to can becomes unwieldy long and a single project statement can span up to dozens or even hundreds of lines of code.

2.2 Code formatters

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

2.2.1 Natural language line breaking

The science of displaying aesthetically pleasing text predates as early as 1956[15]. The first efforts involved inserting carriage returns in natural language text. Until that time, writers were responsible for manually providing carriage returns in their documents before sending them off for printing. The motivation was to “save operating labor and reduce human error”. Once type-setting became more commonplace, the methods for breaking lines of text got more sophisticated.

Knuth and Plass developed a famous line breaking algorithm[20] for LaTeX in 1981. LaTeXis a typesetting program that is popular among scientific academic circles. LaTeXis the program that was used to generate this very document. The line breaking problem was the same as in the 60s: how to optimally break a paragraph of text into lines so that the right margin is minimized. The primitive approach is to greedily fit as many words on a line as possible. However, such an approach can produce embarrassingly bad output in the worst case. Knuth's algorithm uses dynamic programming to find an optimal layout with regards to a fit function that penalizes empty space on the right margin of the paragraph. This algorithm remains a textbook example of the application of dynamic programming[8].

[1] This thesis uses code formatting wherever pretty printing is concerned. In Hughes[16] terms, 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.

Listing 9: A LISP program

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

2.2.2 ALGOL 60

Scowen[33] developed SOAP in 1971, a code formatter for ALGOL 60. The main motivation for SOAP was to make it "easier for a programmer to examine and follow a program" as well as to maintain a consistent coding style. This motivation is still relevant in modern software development. SOAP did provide a line length limit. However, SOAP would fail execution if the provided line length turned out to be too small. With hardware from 1971, SOAP could format 600 lines of code per minute.

2.2.3 LISP

In 1973, Goldstein[11] explored code formatting algorithms for LISP[21] programs. LISP is a family of programming languages and is famous for it's 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 make LISP programs an excellent ground to study code formatters.

Goldstein presented a *recursive re-predictor* algorithm in his paper. The recursive re-predictor algorithm runs a top-down traversal on the abstract syntax tree of a LISP program. While visiting each node, the algorithms tries to first obtain a *linear-format*, i.e. fit remaining body on a single line, with a fallback to *standard-format*, i.e. each argument is put on a separate line aligned by the first argument. Goldstein observes that this algorithm is practical despite the fact that its running time is exponential in the worst case. Bill Gosper used the re-predictor algorithm to implement GRINDEF[3], one of the first code formatters for LISP.

Goldstein's contributions extend beyond formatting algorithms. Firstly, in his paper he studies how to format comments. Secondly, he presents several different formatting layouts which can be configured by the users. Both relevant concerns for modern code formatters.

2.2.4 Language agnostic

Derek C. Oppen pioneered the work on language agnostic code formatting in 1980[26]. 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 the algorithm with a particular programming language. Oppen's algorithm runs in $O(n)$ time and uses $O(m)$ memory for an input program of length n and maximum allowed line length m. Besides impressive performance results, Oppen claims that a key feature of the algorithms is its streaming nature. The algorithm prints formatted lines as soon as they are input instead of waiting until the entire input stream has been read. However, Oppen's algorithm shares a worrying limitation with SOAP: it cannot handle the case when the line length is insufficiently large.

Mark van der Brand presented a library could generate a formatter given a context-free grammar[36]. Brand correctly identifies that the development of code formatters requires a lot of effort. Instead, he proposed a generic solution to the problem. 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 generated formatter could produce both ASCII formatted code as well at LaTeX markup. Since comments are typically not included a syntax tree, the presented algorithm has an elaborate scheme to infer the location of comments in the produced output. Like Oppen's algorithm, this library requires the user to plug in a preprocessor to integrate a particular programming language into the Brand's library. Unlike Oppen's algorithm, Brand does not consider line length limits in his algorithm.

John Hughes extended on Oppen's work on language agnostic formatting in term of functional programming techniques[16]. 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 algebra. Wadler[37] and Chitil[35] extend on Hughes's and Oppen's work in term of performance and programming techniques. However, this branch of work has been limited to printing data structures and not how to format format existing source code.

2.2.5 gofmt

Gofmt[13] 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[4] claims that almost all written Go code is formatted with gofmt. Moreover, 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 was adapted from listing 1 in the introduction.

Listing 10: Gofmt example input/output

```
package main

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

The output of running gofmt through listing 10 is identical to the input. This un-opinionated behavior may be desirable for many software developers, since it gives the programmer flexibility to choose a different layout for each argument list. However, this thesis will not focus on such behavior.

2.2.6 Scalariform

Scalariform[29] was released in 2010 and is a widely used code formatter for Scala. Scalariform does an excellent job of tidying common formatting errors and it 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 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[39] and

Scala.js[7] coding styles have 100 character and 80 character column limits, respectively. As we have learned from other code formatters, adding a line length setting is non-trivial and 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. Listings 11 and 12 shows an example of bin packing enabled and disabled, respectively.

Listing 11: Bin-packing

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

```

Listing 12: No bin-packing

```
// Column 35                      |
class Foo(val x: Int,
          val y: Int,
          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.

2.2.7 clang-format

Daniel Jasper triggered a new trend in optimization based coded formatters with the release of *ClangFormat*[17] in 2013. ClangFormat was developed at Google and is a code formatter for C, C++, Java, JavaScript, Objective-C and Protobuf code. Figure 1 shows the architecture of ClangFormat. The main components are the *structural parser* and the *layouter*.

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

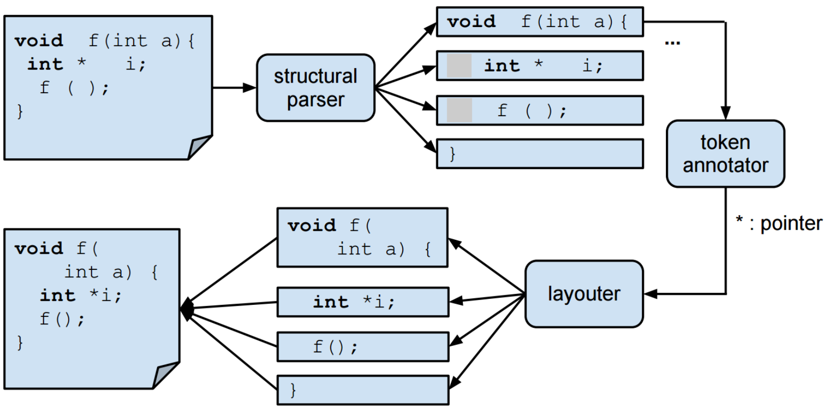

Figure 1: ClangFormat architecture

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") ); }
```

graph with the first token of an unwrapped line being the root and all paths ending at the last token of the unwrapped line. The layouter uses Dijkstra's[6] shortest path algorithm to find the layout that has the lowest penalty. To obtain good performance, the layouter uses several domain specific optimizations to minimize the search space.

Despite being seemingly language independent, ClangFormat does not leverage the language agnostic formatting techniques described section 2.2.4. 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 shows an offensively formatted C++ code snippet. Listing 14 shows the same snippet after being formatted with ClangFormat. ClangFormat is opinionated in the sense that it does not respect the user's line breaking decisions. This feature makes it possible to ensure that all code follows the same style guide, regardless of author.

Listing 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 15: Avoid dead ends

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

2.2.8 dartfmt

Dartfmt[22] was released in 2014 and follows the optimization based trend initiated by ClangFormat. Darftfmt is a code formatter for the Dart programming language, developed at Google. Like ClangFormat, dartfmt has a line length setting and is opinionated. Bob Nystrom, the author of dartfmt, discusses the design of dartfmt in an excellent post[23] on his blog. In his post, Nystrom argues that the design of a code formatters is significantly complicated by a column limit setting. The line wrapping algorithm in dartfmt employs a *best-first search*[27], 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 characters column limit. A plain best-first search would explore a variety of different formatting layouts inside the argument list of `firstCall`. However, the call to `firstCall` already fits on a line and there is no need to explore line breaks inside its argument list. The dartfmt optimized search is able to eliminate such dead ends and quickly figure out to break before the `"long argument string"` literal.

2.2.9 rfmt

The most recent addition to the optimization based formatting trend is `rfmt`[40], 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[16] 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, breaking only when the column limit is exceeded.

We'll take 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 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(")")).
```

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 all 6 blocks in the `rfmt` combinator algebra to express the choice between these formatting layouts. The variable f denotes the function name and $a_1, ...a_m$ denote the argument list. Observe that listing 20 does not express how to find the optimal layout.

To find an optimal layout, `rfmt` employs a smart dynamic programming trick. First, consider the naïve approach of enumerating all possible combinations. This leads to exponential growth which becomes intractable even for small inputs. Dynamic programming comes to the rescue by allowing us to reuse partial solutions. Instead of re-calculating the layout cost at each (starting column, block) pair, we store the result in an associative array keyed by the starting column. However, it turns out that this can still be inefficient in terms of memory and speed[2]. To overcome this limitation, Yelland – the `rfmt` author – presents an indexing scheme that makes it possible to extrapolate the layout cost even for missing keys. We refer to the original paper[40] 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[5] video recording, but then switched to Dijkstra's shortest path algorithms (which in itself is another form of dynamic programming).

3 Algorithms

3.1 Data structures

3.1.1 FormatToken

3.1.2 Split

3.1.3 Policy

3.1.4 OptimalToken

3.1.5 State

3.1.6 Indent

3.2 BestFirstSearch

3.2.1 Router

3.3 Optimizations

3.3.1 dequeueOnNewStatements

3.3.2 recurseOnBlocks

3.3.3 escapeInPathologicalCases

3.3.4 escapeInPathologicalCases

3.3.5 pruneSlowStates

3.3.6 FormatWriter

- vertical alignment
- comment formatting

- stripMargin alignment

4 Tooling

4.1 Heatmaps

4.2 Configuration

4.2.1 maxColumn

4.2.2 binPacking

4.2.3 vertical alignment

4.3 Unit tests

4.4 Property based tests

4.4.1 AST Integrity

4.4.2 Idempotency

4.5 Regressions tests

5 Evaluation

5.1 Micro benchmarks

5.2 Adoption

6 Discussion

6.1 Future work

6.2 Conclusion

References

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

[2] *Apache Spark^{TM} - 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] *CodeReviewComments for golang*. https://github.com/golang/go/wiki/CodeReviewComments. (Accessed on 06/01/2016). Oct. 2015.

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

[6] 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).

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

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

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

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

[11] 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).

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

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

[14] 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).

[15] 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).

[16] 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).

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

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

[19] 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).

[20] 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).

[21] 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).

[22] 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).

[23] 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).

[24] 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).

[25] 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).

[26] 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).

[27] 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).

[28] 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).

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

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

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

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

[33] 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).

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

[35] 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).

[36] 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).

[37] 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).

[38] 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).

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

[40] 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).