Proceedings of the 10th International Conference on Applied Informatics
Eger, Hungary, January 30–February 1, 2017. pp. 211–223
doi: 10.14794/ICAI.10.2017.211

Performance Issues with Implicit Resolution in Scala

Gergely Nagy, Zoltán Porkoláb

Eötvös Loránd University, Faculty of Informatics,
Dept. of Programming Languages and Compilers
`njeasus@caesar.elte.hu`, `gsd@elte.hu`

Abstract

Scala is an emerging programming language that supports multiple programming paradigms. It has been designed to support high levels of expressiveness and to allow writing concise code. To achieve this, it supports many features, including but not limited to macros, DSLs as well as implicit type conversions and implicit argument lists to functions. Due to the wide range of language features and the advanced static type system, the Scala compiler possesses a non-trivial implementation. We have analyzed the performance characteristics of the compiler and have found that the major part of compilation time is spent on typing syntax trees. This includes implicit resolution, thus we have focused our efforts on investigating this specific language feature. We have analyzed how typical usage patterns that involve implicit resolution affect compilation times. Based on this analysis we have managed to assemble a list of recommendations for programming style and code management that allow programmers to leverage implicits to their full potential, but lead to drastically reduced compilation times.

Keywords: implicits, implicit resolution, Scala programming language, compilation performance

MSC: 68N15 Programming languages

1. Introduction

Scala is one of the most popular emerging multi-paradigm programming languages. Supporting both object-oriented and functional paradigms, Scala offers the developers a higher abstraction level to solve everyday programming tasks. One of the most exciting features of Scala is the complex type system: the compiler is able to detect more invalid constructs, even in higher semantic levels. For example, variance checks force the developer to use inheritance rules correctly [1, 2]. The advanced type system also allows the programmers to write more succinct code by using type inference or implicits. This is not only related to convenience, but also to make the code safer, more maintainable and increases developer effectiveness.

Unfortunately, the more complex type system has its cost. The complexity of the compiler itself is higher, and not independently from this, the time spent with compilation is excessive. During the development process, programmers are unnecessarily blocked by waiting for the compiler. This is also the case for incremental builds when only a small fragment of the code has been modified. This has a negative impact on the development process. As programmers are very sensitive to such interruptions, the wasted time is not only related to the compilation time itself, but also to the cognitive disruption it causes. Industry reports increased cost factors related to this phenomenon. Therefore it should be one on the list of primary importance to fix this problem.

In this paper, we analyze the root cause of slow compilation, supported by detailed measurement values. For this purpose we implemented a Scala compiler profiler toolkit. We show the main factors in implicit resolution. Based on the results, we have experimented with various solution candidates, including (temporary) relaxation of implicit resolution as well as providing coding guidelines.

The paper is structured as follows. In the first section we give a summarized overview of implicits in Scala: we describe the three types of them and give code examples. In the next part we discuss the issues with implicits. Firstly, we introduce how Scala typing works, then describe implicit resolution. In section four, we outline the problems with profiling the compiler, then introduce our solution to the problem. In the next section we detail our findings with regards to the performance of implicit resolution in the current Scala implementation. Finally, in section six we go over possible solutions to the problem and discuss the feasibility of each of them. We close the paper by summarizing our findings.

2. On Implicits in Scala

In programming terminology, the term 'implicits' covers a wide area, but everything related to it is true to the original meaning of the word. It means that various entities can be omitted from the software code and a member of the involved toolchain (usually the compiler itself) can resolve the explicit values for these entities. They have an important role in making the code more succinct, easier to understand and maintain. Implicits can take several forms, including implicit types, implicit converters or implicit argument lists. All of these are supported in Scala through the `implicit` keyword.

In this section we will discuss what these kinds of implicits mean and how they can make the code cleaner. As Scala is a block-structured programming language with strict scoping rules, implicits naturally need to adhere to these. To let the compiler automatically apply any implicits, they need to be in scope; they need to be either defined in the current scope or be imported into it. They are resolved when the compiler encounters a typing error. If such an error happens it doesn't simply quit after reporting to the output, but tries to find an entity marked with the implicit keyword that would correct the error.

2.1. Implicit classes

Implicit classes are classes that have their primary constructor avaiable for implicit conversions when the class is in scope[3]. They are a form of implicit converters, as the types of the parameters to their primary constructor are converted to themselves. Usually they are used to enrich APIs. One of the most representative examples is `RichString`:

```scala
class RichString(val self: String) {
  def apply(n: Int): Char
  def capitalize(): String
  .
  .
  .
}
```

Listing 1: Implicit class example

There are some restrictions that apply to implicit classes:

- They must be defined inside another `trait`, `class` or `object`
- They may only take one non-implicit argument in their constructor
- There can't be any other entity with the same name in scope

These restrictions are in place to make resolution unambiguous.

2.2. Implicit argument lists

In functional languages, methods can take multiple argument lists. In Scala, an argument list can be marked as implicit[4, pp. 485], meaning that the compiler will try to find values for the parameters implicitly, without explicitly expanding the given argument list; if the developer wishes so, they can still have the values explicitly defined. Implicit argument lists are often used in DSLs and high-level APIs to let developers omit repeatedly passing a common value in a code block (be it a class or a method). A good example is Scala's `Future` implementation, that needs an `ExecutionContext` to be passed:

```scala
object Future {
  def apply[T](body: => T)(implicit execctx: ExecutionContext) = {...}
}
```

Listing 2: Implicit argument list example

The usual usage pattern for `Futures` would require passing in the same `ExecutionContext` each time a `Future` is used. Instead, the developer can create an `implicit val` in the given scope that the compiler will use to resolve the second argument list.

2.3. Implicit methods

Implicit methods take a simple form of definition[4, pp. 489]:

```
implicit def int2Range(i: Int): Range = {...}
```

Listing 3: Implicit method example

Their application is a little more involved. The compiler will try to apply a method of type `S => T` marked as implicit when

- An expression `e` is of type `S` and `S` does not confirm to the expression's expected type `T`
- A selection `e.m` with `e` of type `S` if the selector `m` does not denote a member of `S`

In simpler terms, if an expression doesn't have the correct type at its place of use, or if a method or field is accessed on an instance of a type without the given name, the compiler will look for the correctly typed implicit method. As one can see, implicit methods can be used as type converters without bloating the code with explicit method applications.

3. Issues with Implicits

In the previous section we have described the three types of implicits in Scala. It can be seen easily that resolving implicits imposes a non-trivial algorithmic complexity. In this section we will give a brief introduction to the Scala compiler and how it is organized; we will also discuss the basic typing features that the language supports, then we will detail the rules and high-level implementation of implicit resolution.

3.1. Typing in Scala

As Pierce writes in his book, "a type system is a syntactic method for automatically checking the absence of certain erroneous behaviors by classifying program phrases according to the kinds of values they compute"[5]. That is, the type system of a language determines how easy it is to use it to write bug-free software. The more the type system covers, the easier it is to write correct programs using it. Functional and object-oriented programming put a lot of emphasis on their type systems and Scala is no exception. It has a strong, static type system [4, pp. 58]. The following advanced features are supported on top of more common object-oriented features:

- *Parametric polymorphism.* Generic programming: when algorithms are implemented without concrete types.
- *Type inference.* Certain type annotations don't need to be present explicitly, the compiler can deduct them based on the context. In Scala, this only works for local types.

- *Existential quantification.* When the concrete type can be omitted, only its existence matters.
- *Variance.* In parametric types, subtyping rules can be applied based on the sub or super types of the type parameter.
- *Structural types.* Structural types can be used to describe what form a type needs to support instead of referring to it by a specific symbol. This is also known as "duck typing".
- *Type views.* Type view definitions can be used to define how a user of a given type can substitute it.

As one can expect, these increase the complexity of the language as well as its implementation. The current Scala compiler consists of many phases (based on the version, between 13 and 30) [6]. One of these phases is the typer, which handles checking type annotations as well as runs the type inference algorithms. This includes *implicit resolution.* When the typer meets an error, it cannot simply give up typing the given piece of AST. Based on rules that we will describe in the next section, it needs to look for implicits that might resolve the error. This further increases both the complexity and the required resources needed to compile Scala code.

3.2. Implicit resolution

Implicit resolution happens during the type phase. There is a well-defined set of rules the compiler uses to look for applicable candidates[4, pp. 482], then selecting one of these candidates. These rules can be summarized as follows.

- *Marking rule.* Only entities marked as implicit can be considered.
- *Scope rule.* An inserted implicit conversion must be in scope as a single identifier, or be associated with the source or target type of the conversion.
- *One-at-a-time rule.* Only one implicit is applied by the compiler.
- *Explicits first.* This seem trivial, but if a value is provided explicitly, it must be used - the compiler won't try to apply implicits if typing succeeds.
- *Occasion rule.* Implicits are only used in three places: conversions to an expected type, conversions of the receiver of a selection and with implicit parameters.
- *Naming rule.* The names of implicits don't matter.

The actual implementation of implicit resolution[7] is fairly close to the rules listed above. When the typer fails to type an AST, it will start the implicit resolution process. In this, there is a set of possible candidates that are marked as implicit. This set is then reduced in a loop until only one implicit remains –that fixes the type error– or the resolution fails, which means that the typing error is valid and the compiler needs to abort.

4. How we measured

Finding performance bottlenecks in software is not always easy as the complexity of the software increases. The Scala compiler is possibly one of the most complicated Scala products. The code relies on the standard library heavily to implement ASTs. For this reason, regular JVM-based profiling tools don't necessarily identify high-level hotspots, but rather "low-level" APIs, such as list append (`List.::`) and hash value computation. This led us develop our own methodology to understand which parts of the compiler are slow to run.

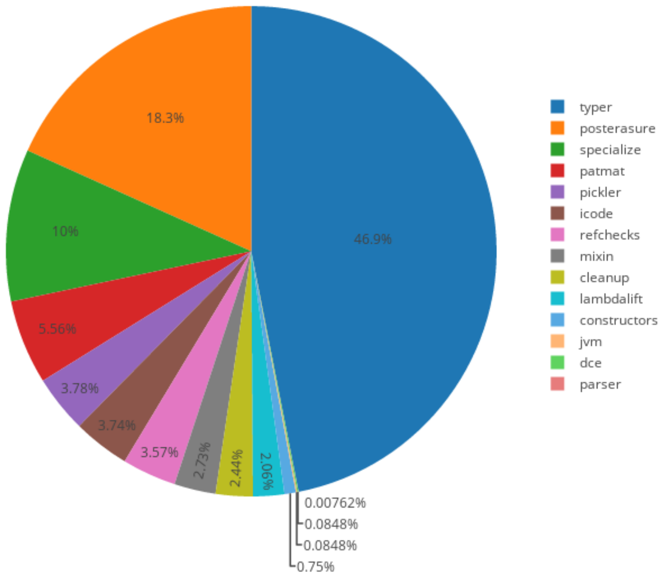

Figure 1: Average distribution of time spent on each compile phase.

4.1. Compiler plugin

We have developed a compiler plugin that measures the time it took finish each compile phase. There are phase groups that cannot be interrupted by custom plugins because the intermediate AST transformation would break. Due to the nature of the compiler, we then needed to appropriately aggregate the results for larger codebases. These measurements let us notice that the compiler, depending on some characteristics of the source, spends between 30% and 50% of its time in the typer phase. The compilation time distribution of a large, proprietary codebase can be seen on figure 1. Apart from this software we have also analyzed Scala itself –the compiler and the language's standard library. Both of these provided the same performance characteristics, and made it easy to see that the biggest performance improvement can be achieved by enhancing the typer. Since it also includes resolving implicits, we have taken a closer look at this part, and recognized it as a single point for possible improvements. To test our hypotheses, we have developed a test suite that generates synthetic code and it can be analyzed using the compiler plugin. The test suite allowed as to easily change certain aspects of implicit usage: the in-scope availability of implicits and their usage.

5. Our findings

In this section we will discuss our findings on performance characteristics of the implicit resolution in the Scala compiler. We will describe the two main factors of it, namely how many implicits are in a given scope and the amount of implicit usage. We will present data that steered us towards trying to ease up the search criteria and then providing some coding guidelines and practices with regards to implicit usage.

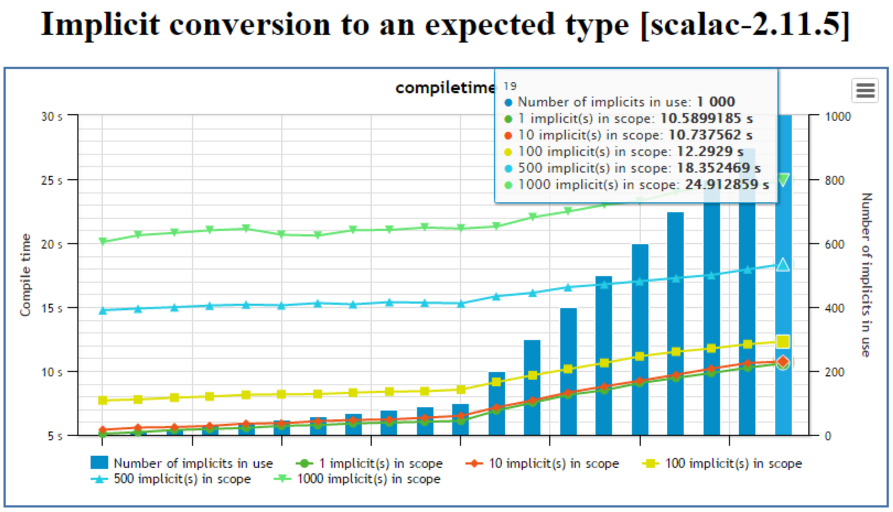

Figure 2: Compilation times with constant available implicit and increased implicit usage.

5.1. Implicit usage

The most basic variable to change is the number of implicit calls the compiler has to resolve. For the following tests we have fixed the number of available implicits in the given scope, then increased the number of required implicit applications. On figure 2, the blue bars represent the number of implicit uses, the dotted lines display how many implicits are in scope, between 1 and 1000. As it can be seen, there is no high correlation between the two variables: increasing the implicit usage doesn't affect compilation times to a great amount – the change can be attributed to the growing source code alone.

5.2. Available implicits in scope

Based on the rules listed in section 3.2, another dimension to investigate is the number of implicits available when the compiler needs to resolve them. We have generated code with an increasing number of implicits, while the actual usage of implicits remained constant. The aggregated analysis can be found on figure 3. As previously, the blue bars show the main factor we are testing: the number of

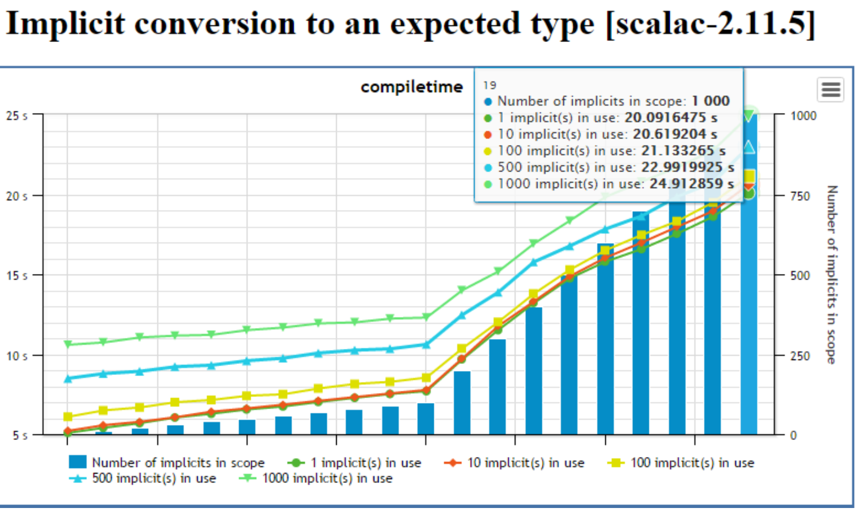

Figure 3: Compilation times with constant implicit usage and increased available implicits.

implicit definitions in scope. The dotted lines are now used to express how many implicits are actually used in the code. It is easy to see the direct correlation with the two factors for each test case. As we increase the set of definitions in which the compiler needs to search for implicit candidates, the typer phase takes proportionally longer.

5.3. Conclusions

Based on the performance analysis we have described in this section, we can draw the conclusion that the main factor that affects the performance of implicit resolution, and more widely, the typer phase in the Scala compiler, is going through all the available definitions in the scope to search for implicit candidates. There are two possible ways of optimizing this:

- Make analyzing each item faster, thus applying the operation on the whole set faster
- Decrease the size of the input set

In the following section, we will consider both approaches and describe what we have observed.

6. Approaches to improve implicit resolution performance

In this section we will go over the two approaches we have tried to better the Scala compiler's performance for implicit resolution. As we have seen in the previous section, we can make the candidate analysis faster or we can reduce the set of candidates that we need to consider. We will describe these approaches in the respective sections.

6.1. Relaxed search criteria

The inferring algorithm that is used by the implicit resolution codepath in the Scala compiler needs to decide if two types are compatible with each other in the meaning of the Liskov substitution principle[8]. This problem exposes a non-trivial algorithmic complexity, and based on some profiling information, takes considerable amount of time for every typing error. The implementation[9] has a way to *relax* (or shortcut) the search algorithm so it doesn't perform a full type check, but rather uses the much simpler `isPlausibleSubtype` method. Interestingly enough, the incoming parameter has been denoted as `fast: Boolean`, making it a trivial choice for experiments. We have assumed that this shortcut could be used for some usages of the compiler, for example for incremental compilations used in IDEs. Although it would not solve the performance issue *once and for all*, it could provide big enough improvements for software developers. We have compiled our own version of Scala and used this modified version in our performance tests. On figures 4 and 5, we can see that relaxing the search criteria does improve implicit resolution times significantly. As the line charts display, there is not much of a difference between explicitly applying the implicit function and letting the compiler find it as a candidate. We have managed to build the modified Scala version and run our performance tests with it, but as a semantic correctness test, we have also tried to compile the same version of compiler code and run the enclosed test suite with it. Unfortunately we have run into several issues. Resolving implicits involving existential types[10] has proven to be problematic:

```
[error] [...]/scala\trytobuild/scala/src/library/scala/collection \
        /parallel/ParIterableLike.scala:527: type mismatch;
[error]  found   : scala.collection.parallel.IterableSplitter[T]
[error]  required: ?{def assign(x$1: ? >: \
```

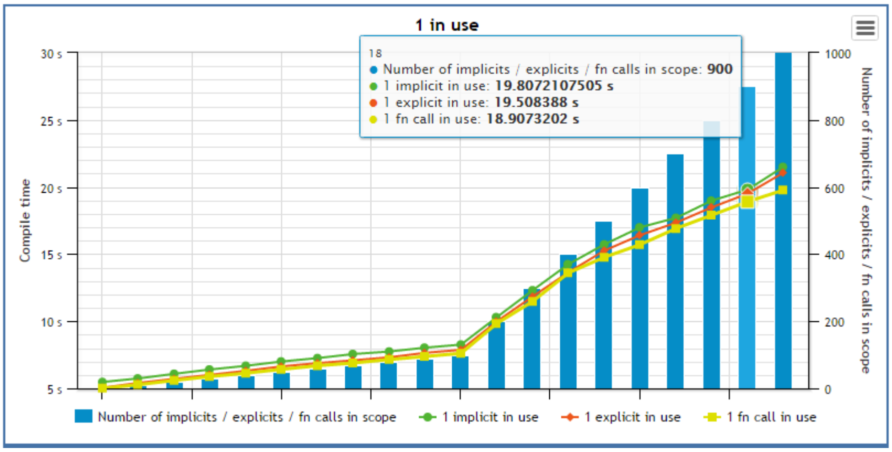

Figure 4: One explicit function call and one implicit resolutions with varying number implicits in scope.

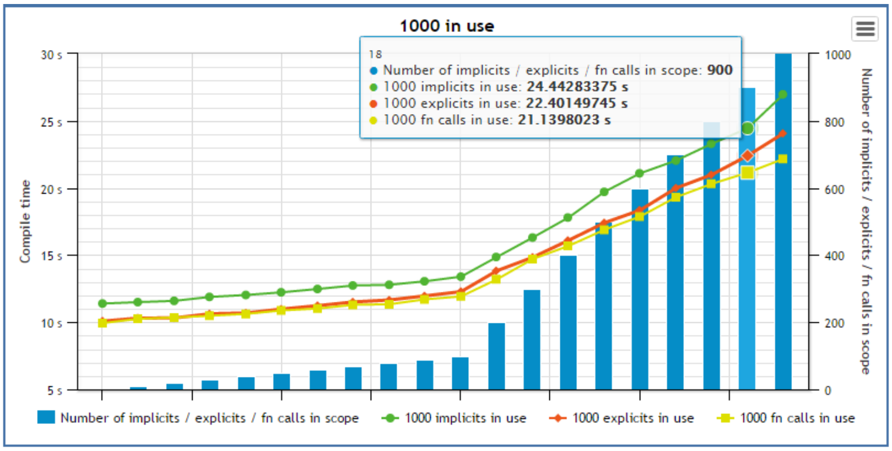

Figure 5: 1000 explicit functions call and 1000 implicit resolutions with varying number implicits in scope.

```
        scala.collection.generic.DefaultSignalling with \
        scala.collection.generic.VolatileAbort): ?}
[error] Note that implicit conversions are not applicable because \
        they are ambiguous
```

We have tried to work around these issues by not running the relaxed algorithm for these existential types, but this has proven no use. As one would assume, resolving the issue has turned out to be much more complex than we could solve, so we have abandoned this experiment.

6.2. Coding patterns

As discussed previously, reducing the size of the candidate set for implicit resolution can be another useful approach. For this reason, we would eliminate all useless implicit definitions before running the typer phase. We have considered creating a programmatic way for removing these definitions before right the typer meets them, but concluded that we would probably create an algorithm that is at least as slow as the one in the compiler. Doing this would probably not help the performance.

Instead, we have realized that the most trivial way to reduce the number of available implicits is by *not having them* in the software code. We have examined several large software projects written in Scala, and identified usage patterns for implicits that worsen the performance, but don't really help the developer or improve the code. With a little more care, these can be changed resulting in cleaner and faster-compiling code. We have created some coding patterns and guidelines that are easy to follow. They not only help with the performance, but adhering to them can possibly reduce implicit-related errors that usually turn out to be hard to find and debug.

- *No file-level imports for implicits.* A lot of times, imports are moved to the top of a source file either by IDEs or developers themselves. Normally, this improves neatness of the code, but with implicits, this necessarily bogs down the compiler. Every implicit needs to be added to the candidate set and for larger files that contain multiple classes, the size of the list of imported implicits can be significant. If there are no implicits imported at the file-level, the mixture of these classes will have no effect on each other.

- *Import implicits only in the block they are being used.* Minimizing the scope for implicits should be the ultimate goal. Since Scala is a block-structured language, it supports imports in all blocks, and due to the visibility rules, these imports can't escape the block. If implicits are only imported in methods or bodies of structures where they are used, they cannot be applied –accidentally or not accidentally– in other places, and the compiler doesn't need to check them for every type error outside of the block.

- *Don't wildcard-import implicits.* Importing all contained entities from a namespace is often useful, but it can accidentally import many implicits. The most trivial solution to the problem is only importing the necessary definitions by name.

- *Create implicits in smaller namespaces.* Providers of implicits should try to reduce the number of them in their namespaces. This encourages the previous point and lets users of implicits selectively add them to their code. This also better documents code as it's easier to track down the intent for a given implicit.

- *Don't create silver-bullet type collections of implicits.* This reinforces the previous pattern, but it's important to highlight this as many patterns we have observed dumped all seemingly related implicits in on huge namespace that got imported in many places. This should be avoided by both the providers and the clients of implicits.

7. Conclusion

We have discussed the performance characteristics and issues of implicit resolution in Scala. We have started by introducing implicits, describing the tree kinds and giving examples. We have then given a summary of how implicit resolution works, and listed the related performance issues. We have introduced our high-level profiler and test suite that were used to correctly measure compilation times of various use cases. We have examined a great number of instances of synthetic code; we have analyzed and presented the collected data. We have arrived at the conclusion that the main effect on performance is the locally available implicits form which the compiler needs to choose. We have given two possibilities on how this could improved: by either improving the performance of analyzing each item, or by reducing the starting set. As the first approach, we have discussed how speeding up the analysis could be done by relaxing the search criteria. We have modified the compiler then ran it through our test suite. This has shown great results, but unfortunately we have seen non-trivial compilation errors while compiling the Scala codebase. We haven't been able to resolve these issues. For a second improvement, we have realized the trying to programmatically eliminate bogus implicits would be too costly, so instead we have identified certain coding patterns and guidelines. We have listed five rules that are easy to follow, and they can help with significantly reducing the work the compiler needs to accomplish during implicit resolution. Not only this, but adhering to these suggestions could greatly improve accidental and hard-to-debug issues related to implicits. In the future, we would like to introduce an IDE extension that would point out instances of these rules in the code to help with identifying and fixing certain issues in real time.

References

[1] G. Castagna, "Covariance and contravariance: Conflict without a cause," *ACM Trans. Program. Lang. Syst.*, vol. 17, pp. 431–447, May 1995.

[2] F. Weber, "Getting class correctness and system correctness equivalent - how to get covariance right," 1992.

[3] "Scala online documentation." `http://docs.scala-lang.org/overviews/core/implicit-classes.html`, 2015.

[4] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: Updated for Scala 2.12.* Artima Press, 2016.

[5] B. C. Pierce, *Types and Programming Languages (MIT Press).* The MIT Press, 2002.

[6] "Scala github repository." `https://github.com/scala/scala/blob/2.12.x/src/compiler/scala/tools/nsc/settings/ScalaSettings.scala#L130`, 2017.

[7] "Scala github repository." `https://github.com/scala/scala/blob/2.12.x/src/compiler/scala/tools/nsc/typechecker/Implicits.scala`, 2017.

[8] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Trans. Program. Lang. Syst.*, vol. 16, pp. 1811–1841, Nov. 1994.

[9] "Scala github repository." `https://github.com/scala/scala/blob/2.12.x/src/compiler/scala/tools/nsc/typechecker/Implicits.scala#L536`, 2017.

[10] A. M. Pitts, "Existential types: Logical relations and operational equivalence," in *Automata, Languages and Programming*, pp. 309–326, Springer Berlin Heidelberg, 1998.