

Scala Implicits Are Everywhere

A large-scale study of the use of Scala implicits in the wild

FILIP KŘIKAVA, Czech Technical University in Prague, CZ

HEATHER MILLER, Carnegie Mellon University, USA

JAN VITEK, Czech Technical University in Prague and Northeastern University, USA

The Scala programming language offers two distinctive language features *implicit parameters* and *implicit conversions*, often referred together as *implicits*. Announced without fanfare in 2004, implicits have quickly grown to become a widely and pervasively used feature of the language. They provide a way to reduce the boilerplate code in Scala programs. They are also used to implement certain language features without having to modify the compiler. We report on a large-scale study of the use of implicits in the wild. For this, we analyzed 7,280 Scala projects hosted on GitHub, spanning over 8.1M call sites involving implicits and 370.7K implicit declarations across 18.7M lines of Scala code.

CCS Concepts: • Software and its engineering → Language features; General programming languages.

Additional Key Words and Phrases: Implicit parameters, implicit conversions, corpora analysis, Scala

ACM Reference Format:

Filip Křikava, Heather Miller, and Jan Vitek. 2019. Scala Implicits Are Everywhere: A large-scale study of the use of Scala implicits in the wild. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 163 (October 2019), 28 pages. <https://doi.org/10.1145/3360589>

1 INTRODUCTION

“...experienced users claim that code bases are train wrecks because of overzealous use of implicits.”

—M. Odersky, 2017

“...can impair readability or introduce surprising behavior, because of a subtle chain of inference.”

—A. Turon, 2017

“Any sufficiently advanced technology is indistinguishable from magic.”

—A.C. Clarke, 1962

Programming language designers strive to find ways for their users to express programming tasks in ways that are both concise and readable. One approach to reduce boilerplate code is to lean on the compiler and its knowledge and understanding of the program to fill in the “boring parts” of the code. The idea of having the compiler automatically provide missing arguments to a function call was first explored by Lewis et al. [2000] in Haskell and later popularized by Scala as *implicit parameters*. *Implicit conversions* are related, as they rely on the compiler to automatically adapt data structures in order to avoid cumbersome explicit calls to constructors. For example, consider the following code snippet: `"Just like magic!"`.enEspanol Without additional context one would expect the code not to compile as the `String` class does not have a method `enEspanol`. In Scala, if the compiler is able to find a method to convert a `String` object to an instance of a class that has the required method (which resolves the type error), that conversion will be inserted silently by the compiler and, at runtime, the method will be invoked to return a value, perhaps `"Como por arte de magia!"`.

Implicit parameters and conversions provide ways to (1) extend existing software [Lämmel and Ostermann 2006] and implement language features outside of the compiler [Miller et al. 2013], and (2) allow end-users to write code with less boilerplate [Haoyi 2016]. They offload the task

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2019 Copyright held by the owner/author(s).

2475-1421/2019/10-ART163

<https://doi.org/10.1145/3360589>

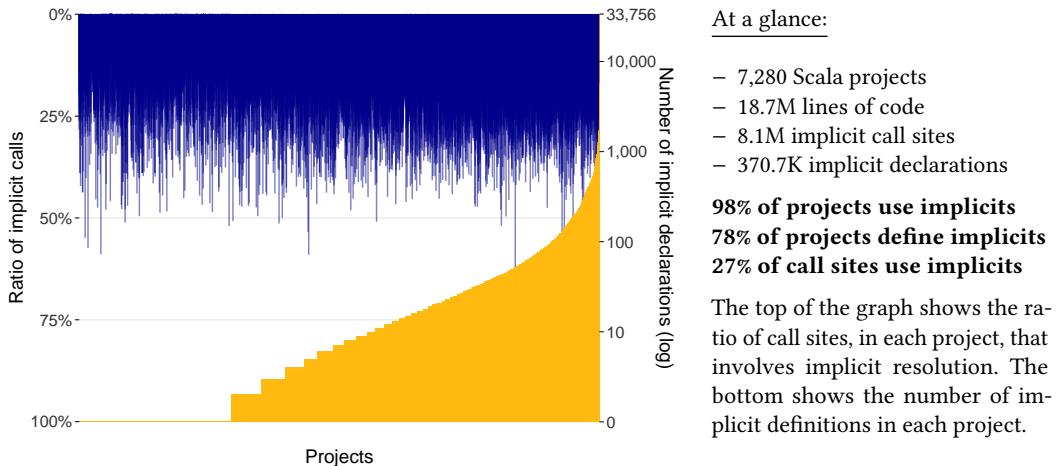


Fig. 1. Implicit usage across our corpus

of selecting and passing arguments to functions and converting between types to the compiler. For example, the `enEspanol` method from above uses an implicit parameter to get a reference to a service that can do the translation: `def enEspanol(implicit ts:Translator):String`. Calling a function that has implicit arguments results in the omitted arguments being filled from the context of the call based on their types. Similarly, with an implicit conversion in scope, one can seamlessly pass around types that would have to be otherwise converted by the programmer.

The Good: A Powerful Tool. It is uncontroversial to assert that implicits changed how Scala is used. Implicits gave rise to new coding idioms and patterns, such as type classes [Oliveira et al. 2010]. They are one of a few key features which enable embedding Domain-Specific Languages (DSLs) in Scala. They can be used to establish or pass context (e.g., implicit reuse of the same threadpool in some scope), or for dependency injection. Implicits have even been used for computing new types and proving relationships between them [Miller et al. 2014; Sabin 2019]. The Scala community adopted implicits enthusiastically and uses them to solve a host of problems. Some solutions gained popularity and become part of the unofficial programming lexicon. As usage grew, the community endeavored to document and teach these idioms and patterns by means of blog posts [Haoyi 2016], talks [Odersky 2017] and the official documentation [Suereth 2013]. While these idioms are believed to be in widespread use, there is no hard data on their adoption. How widespread is this language feature? And what do people do with implicits? Much of our knowledge is folklore based on a handful of popular libraries and discussion on various shared forums.

Our goal is to document, for language designers and software engineers, how this feature is really used in the wild, using a large-scale corpus of real-world programs. We provide data on how they are used in popular projects engineered by expert programmers as well as in projects that are likely more representative of how the majority of developers use the language. This paper is both a retrospective on the result of introducing this feature into the wild, as well as a means to inform designers of future language of how people use and misuse implicits.

The Bad: Performance. While powerful, implicits aren't without flaws. Implicits have been observed to affect compile-time performance; sometimes significantly. For example, a popular Scala project reported a three order-of-magnitude speed-up when developers realized that an implicit conversion was silently converting Scala collections to Java collections only to perform a single

```

case class Card(n:Int, suit:String) {
  def isInDeck(implicit deck: List[Card]) =
    deck contains this
}
implicit def intToCard(n:Int) = Card(n, "club")
implicit val deck = List(Card(1, "club"))

1.isInDeck

```

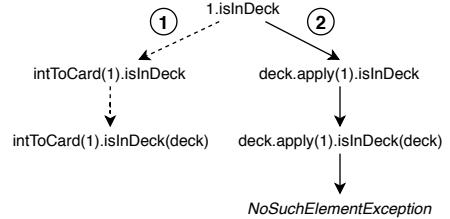


Fig. 2. Instead of injecting a conversion to `intToCard` (1), the compiler injects `deck.apply` (2) since `List[A]` extends (transitively) `Function[Int, A]`. An exception is thrown because the deck contains only one element (<http://scalapuzzlers.com/>)

operation that should have been done on the original object.¹ Another project reported a 56 line file taking 5 seconds to compile because of implicit resolution. Changing one line of code to remove an implicits, improved compile time to a tenth of second [Torreborre 2017]. Meanwhile, faster compilation is the most wished-for improvement for future releases of Scala [Lightbend 2018]. Could implicit resolution be a significant factor affecting compilation times across the Scala ecosystem?

The Ugly: Readability. Anecdotally, there are signs that the design of implicits can lead to confusing scenarios or difficult-to-understand code. Figure 2 illustrates how understanding implicit-heavy code can place an unreasonable burden on programmers². In this example, the derivation chosen by the compiler leads to an error which requires understanding multiple levels of the type hierarchy of the `List` class. Such readability issues have even lead the Scala creators to reconsider the design of Scala’s API-generation tool, Scaladoc. This was due to community backlash [Marshall 2009] following the introduction of the Scala 2.8 Collections library [Odersky and Moors 2009]—a design which made heavy use of implicits in an effort to reduce code duplication. The design caused a proliferation of complex method signatures across common data types throughout the Scala standard library, such as the following implementation of the `map` method which was displayed by Scaladoc as: `def map[B,That](f:A=>B)(implicit bf:CanBuildFrom[Repr,B,That]):That`. To remedy this, Scaladoc was updated with *use-cases*,³ a feature designed to allow library authors to manually override method signatures with simpler ones in the interest of hiding complex type signatures often further complicated by implicits. The same `map` signature thus appears as follows in Scaladoc after simplification with a `@usecase` annotation: `def map[B](f: (A) => B): List[B]`

This Work. To understand the use of implicits across the Scala ecosystem, we have built an open source and reusable pipeline to automate the analysis of large Scala code bases, compute statistics and visualize results. We acquired and processed a corpus of 7,280 projects from GitHub with over 8.1M implicit call sites and more than 370.7K implicit declarations across 18.7M non-empty lines of Scala code. We observed over 98.2% projects using implicits, and 78.2% projects declaring implicits. With close to 27.2% of call sites requiring implicit resolution, implicits are the most used feature of Scala. Figure 1 summarizes the usage of implicits in our corpus. Our results document which idioms and patterns are popular and in application, library and tests. We provide data on the compilation time cost of implicits and the complexity of implicits. Our artifact is available at:

<https://doi.org/10.5281/zenodo.3369436>

¹Documented in <https://github.com/mesosphere/marathon/commit/fbf7f29468bda2ec29b7fbf80b6864f46a825b7a>.

²For example, an entire book is devoted to so-called “puzzlers,” or “enigmatic Scala code that behave highly contrary to expectations” which “will entertain and enlighten even the most accomplished developer” [Phillips and Serifovic 2014]

³cf. <https://docs.scala-lang.org/overviews/scaladoc/for-library-authors.html>

2 AN OVERVIEW OF SCALA IMPLICITS

Scala is a statically typed language that bridges the gap between object-oriented and functional programming. Implicits were included in the first release in 2004. In that version *implicit conversions* were used to solve the late extension problem; namely, given a class *C* and a trait *T*, how to have *C* extend *T* without touching or recompiling *C*. Conversions add a wrapper when a member of *T* is requested from an instance of *C*. Scala 2.0 added *implicit parameters* in 2006.

2.1 Implicit Conversions

Implicit conversion provides a way to use a type where another type is required without resorting to an explicit conversion. They are applied when an expression does not conform to the type expected by its context or when a called method is not defined on the receiver type. A conversion is defined with an implicit function or class, or an implicit value of a function type (e.g., `implicit val x:A=>B`).

Implicit conversions are not specific to Scala. They also appear in languages such as C++ or C#. The difference is that conversions are typically defined in the class participating in the conversion, while in Scala the implicit conversions can be defined in types unrelated to the conversion types. This allows programmers to selectively import conversions. For example it is possible to define an implicit conversion from a `String` to an `Int`:

```
implicit def string2int(a: String): Int = Integer.parseInt(a)
val x: Int = "2"
```

Implicit conversions are essential to provide seamless interoperability with Java which was important in the early days of Scala. Conversions are also one of the main building blocks for constructing embedded Domain-Specific Languages (DSLs). For example, the following code snippet adds some simple time unit arithmetic that feels natural in the language.

```
case class Duration(time: Long, unit: TimeUnit) {
  def +(o: Duration) = Duration(time + unit.convert(o.time, o.unit), unit)
}
implicit class Int2Duration(that: Int) {
  def seconds = new Duration(that, SECONDS); def minutes = new Duration(that, MINUTES)
}
5.seconds + 2.minutes //Duration(125L, SECONDS)
```

2.2 Implicit Parameters

A method or a constructor can define implicit parameters. The arguments to these parameters will be filled in by the compiler at every call site with the most suitable values in the calling context. For example, a function `def sub(x:Int)(implicit y:Int)=x-y` with implicit parameter *y* can be called with `sub()` provided that the compiler can find an implicit such as `implicit val n=1`. The compiler looks for implicits in the current lexical scope and if there are no eligible identifiers then it searches the implicit scope of the implicit parameter's type (associated companion objects⁴ and packages). If a value is found, the compiler injects it into the argument list of the call. If multiple values are found and none of them is more specific than the others, an ambiguity compilation error is thrown. An error is also raised when no eligible candidate is found. Importantly, besides having the correct type, an implicit value from a lexical scope must be accessible using its simple name (without selecting from another value using dotted syntax). This means that normal rules for name binding including shadowing apply. Implicit values (`val`), variables (`var`), objects (`object`) and functions (`def`) without explicit parameters can all be used to fill implicit parameters. An implicit parameter of a function type `A => B` can be used as an implicit conversion in the method body. For example

⁴A companion object is a singleton associated with a class used to define static fields and methods.

`def get[T, C](xs: C, n: Int)(implicit conv: C => Seq[T]): T = xs(n)` can be called with any type `C`, as long as there is implicit conversion in scope that can convert `C` into a sequence.

2.3 Idioms and Patterns

Over time, programmers have put implicits to many uses. This section describes the most widely discussed implicit idioms. This list is based on our understanding of the state of practice. It is not expected to be exhaustive or definitive.

2.3.1 Late Trait Implementation. This idiom is a solution for the late extension problem, and was the original motivation for adding implicits to Scala in the first place. To add a new trait to an existing class, one can define a one-parameter conversion that returns an instance of the trait.

```
implicit def call2Runnable(x: Callable[_]): Runnable = new Runnable { def run = x.call }
```

This snippet adds the `Runnable` interface to any type that implements `Callable`. Conversions can also take implicit parameters, they are then referred to as *conditional* conversions.

```
implicit def call2Future[T](x: Callable[T])(implicit ctx: ExecutionContext): Future[T]
```

For example, the above defines a late trait implementation that is only applicable if there exists an execution context in scope.

2.3.2 Extension Methods. Extension methods allow developers to add methods to existing classes. They are defined with an `implicit def` that converts objects to a new class that contains the desired methods. Scala 2.10 added syntactic sugar to combine conversion and class declaration in the `implicit class` construct. The conversion takes a single non-implicit parameter as shown in the following snippet where `zip` is added to any `Callable`.

```
implicit class XtensionCallable[T](x: Callable[T]) {
  def zip[U](y: Callable[U]): Callable[(T, U)] = () => (x.call, y.call)
}
val c1 = () => 1; val c2 = () => true; val r = c1.zip c2 // r: Callable[(Int, Boolean)]
```

An extension method is convenient as it allows to write `c1.zip c2` instead of `zip(c1, c2)`. It is an important feature for embedded DSLs. On the other hand, unlike static methods, it is harder to read. Without knowing the complete code base it is difficult to know where a calling method is defined and how the definition got into the current scope. Extension methods can also be conditional. For example, we can add a `def schedule(implicit c: ExecutionContext)` method that will run the callable on the implicitly provided execution context if it is present. If there is none, the developer will get a compile-time error “*cannot find an implicit ExecutionContext ... import scala.concurrent.ExecutionContext.global.*”. This is because the `ExecutionContext` is annotated with `@implicitNotFound`, a Scala annotation allowing one to customize the compile-time error message that should be outputted in the case no implicit value of the annotated type is available.

2.3.3 Type Classes. Oliveira et al. [2010] demonstrated how to use implicit parameters to implement type classes [Wadler and Blott 1989]. Fig. 3a defines a trait `Show` that abstracts over pretty-printing class instances. The function `show` can be called on instances `T`, for which there is an implicit value of type `Show[T]`. This allows us to retrospectively add support to classes we cannot modify. For example, given a class `Shape(sides: Int)` from a 3rd party library, we can define the implicit value `ShapeShow` to add pretty printing (Fig. 3b). This is an implicit object that extends `Show` and implements `show`. Thus when `show` is called with an explicit argument of type `Shape`, for example `show(Shape(5))`, the compiler adds the implicit `ShapeShow` as the implicit argument `ev`, resulting in `show(Shape(5))(shapeShow)`. Since functions can be used as implicit parameters, we can generalize this example and create an implicit allowing us to show a sequence of showable instances. In the following snippet, `listShow` is

```

trait Show[T] {
  def show(x: T): String
}

def show[T](x: T)(implicit ev: Show[T]) =
  ev.show(x)

```

(a)

```

case class Shape(n: Int)

implicit object shapeShow extends Show[Shape] {
  def show(x: Shape) = x.n match {
    case 3 => "a triangle"; case 4 => "a square"
    case _ => "a shape with $n sides"
  }
}

```

(b)

```

implicit def listShow[T](implicit ev: Show[T]) = new Show[List[T]] {
  def show(x: List[T]) = x.map(x => ev.show(x)).mkString("a list of [", ", ", ", ", "]")
}

```

(c)

Fig. 3. Type classes

a generic type class instance that combined with an instance of type `Show[T]` returns a type class instance of type `Show[List[T]]` (Fig. 3c). Thus, a call to `show(List(Shape(3), Shape(4)))` is transformed to `show(List(Shape(3), Shape(4)))(listShow[Shape](shapeShow))`, with two levels of implicits inserted. This implicit type class derivation is what makes type classes very powerful. The mechanism can be further generalized using implicit macros to define a *default* implementation for type class instances that do not provide their own specific ones [Miller et al. 2014; Sabin 2019].

2.3.4 Extension Syntax Methods. Type classes define operations on types, when combined with extension methods it is possible to bring these operations into the corresponding model types. We can extend the `Show[T]` type class and define an extension method

```
implicit class ShowOps[T](x: T)(implicit s: Show[T]) { def show = s.show(x) }
```

allowing one to write directly `shape(3).show` instead of `show(Shape(3))`. The `ShowOps[T]` is a conditional conversion that is only applied if there is an instance of the `Show[T]` in scope. This allows library designers to use type class hierarchies instead of the regular sub-typing. The name *extension syntax methods* comes from the fact that developers often lump these methods into a package called *syntax*.

2.3.5 Type Proofs. Implicit type parameters can be used to enforce API rules at a compile time by encoding them in types of implicit parameters. For example, `flatten` is a method of `List[A]` such that given an instance `xs: List[List[B]]`, `xs.flatten` returns `List[B]` concatenating the nested lists into a single one. This is done with an implicit parameter:

```
class List[A] { def flatten[B](implicit ev: A => List[B]): List[B] }
```

Here, `A => List[B]` is an implicit conversion from `A` to `List[B]`. It can also be viewed as a predicate that must be satisfied at compile time in order for this method to be called. We can define an implicit function `implicit def isEq[A]: A=>A = new =>[A,A]{}` that will act as generator of proofs such that `A` in `A => List[B]` is indeed `List[B]`. Therefore, a call `List(List(1)).flatten` will be expanded to `List(List(1)).flatten(isEq[List[Int]])` since `A` is a `List` while `List(1).flatten` will throw a compile time exception: “*No implicit view available from Int => List[B]*”.

2.3.6 Contexts. Implicit parameters can reduce the boilerplate of threading a context parameter through a sequence of calls. For example, the methods in `scala.concurrent`, the concurrency library in Scala’s standard library, all need an `ExecutionContext` (e.g., a thread pool or event loop) to execute

their tasks upon. The following code shows the difference between explicit and implicit contexts.

<pre>val ctx = ExecutionContext.global val f1 = Future(1)(ctx) val f2 = Future(2)(ctx) val r = f1.flatMap(r1 => f2.map(r2 => r1 + r2)(ctx))(ctx)</pre> <p style="text-align: center;">With <i>explicit</i> context</p>	<pre>implicit val ctx = ExecutionContext.global val f1 = Future(1) val f2 = Future(2) val r = for(r1 <- f1; r2 <- f2) yield r1 + r2</pre> <p style="text-align: center;">With <i>implicit</i> context</p>
---	---

On the left, an explicit context is passed around on every call to a method on `Future`, while on the right much of the clutter is gone thanks to implicits. This de-cluttering hides the parameters and makes calls to `map` and `flatMap` more concise. The idiom consists of the declaration of an implicit context (usually as an `implicit val`), and the declaration of the functions that handle it.

2.3.7 Anti-patterns: Conversions. A widely discussed anti-pattern is the conversions between types in unrelated parts of the type hierarchy. The perceived danger is that any type can be automatically coerced to a random type unexpectedly; e.g., imagine a conversion from `Any` to `Int` introduced into the root of a big project. One could imagine such a conversion wreaking havoc in surprising places in a code base and being difficult to track down. Another anti-pattern is conversions that go both ways [Odersky 2017]. Since conversions are not visible, it is difficult to reason about types at a given call site as some unexpected conversion could have happened. An example is the, now deprecated, JAVA collection conversion. In an earlier iteration, Scala defined implicit conversions between JAVA collections and its own, such as:

```
implicit def asJavaCollection[A](it: Iterable[A]): java.util.Collection[A]
implicit def collectionAsScalaIterable[A](i: java.util.Collection[A]): Iterable[A]
```

As they were often imported together using a wildcard import `collection.JavaConversions._`, it was easy to mistakenly invoke a JAVA method on a Scala collection and vice-versa silently converting the collections from one to another. Furthermore, in this case, these conversions also change semantics as the notion of equality in JAVA collections is different from Scala collections (reference vs. element equality). Since implicit conversions can introduce some pitfalls, the compiler issues a warning when compiling an implicit conversion definition. It can be suppressed by an import (or a compiler flag) which is usually automatically done by an IDE and thus diminishing the utility of these warnings.

2.4 Complexity

Implicits help programmers by hiding the “boring parts” of programming tasks, the plumbing that does not require skill or attention. The problem is that, as the above idioms demonstrate, implicits are also used for subtle tasks. Their benefits can turn into drawbacks. One way to measure the potential complexity of implicits is to look at the work done by the compiler. When implicits work, programmers need not notice their presence. But when an error occurs, the programmer suddenly has to understand the code added by `scalac`. For example, a comparison of two tuples `(0,1)<(1,2)` gets expanded to `orderingToOrdered((0,1))(Tuple2[Int, Int])<(1,2)`. The compiler injects two additional calls (`orderingToOrdered` implicit conversion, `Tuple2` type class) with two implicit arguments (`Int`). The question is how much of this *filling* there is.

Tooling can help navigate the complexity added by implicits. The plugin for IntelliJ IDEA has a feature that can show implicit hints, including the implicit resolution in the code editor. This effectively reveals the injected code making it an indispensable tool for debugging. However, turning the implicit hints on severely hinders the editor performance, creating a significant lag when working

with implicits-heavy files. The second problem with this is that the IntelliJ compiler is not the same as scalac, and often implicit resolution disagrees between the two compiler implementations.

Another common problem that hinders understanding is related to implicit resolution. Eligible implicits for both conversions and parameters are searched in two different scopes. The search starts in the lexical scope that includes local names, enclosing members and imported members and continues in the implicit scope that consists of all companion objects associated with the type of the implicit parameter. The advantage of the implicit scope is that it does not need to be explicitly imported. This prevents errors caused by missing imports for which, due to the lack of global implicit coherence, the compiler cannot give a better error message than a type mismatch, “*member not found*” or “*could not find implicit value*”. Implicit scope has a lower priority allowing users to override defaults by an explicitly importing implicit definition into the lexical scope. A consequence of this is that an import statement can change program semantics. For example, in the code below contains two late trait implementations of a trait `T` for a class `A: C1` defined in implicit scope of the class `A`, and `c2` defined in an unrelated object `O`:

```
trait T { def f: Int }
class A; object A { implicit class C1(a: A) extends T { def f = 1 } }
object O { implicit class C2(a: A) extends T { def f = 2 } }
new A().f
```

At the call to `f`, the compiler will use the `C1` conversion resolved from the implicit scope so the result will be 1. However, if later there is an `import O._` before the call site, the expression will return 2. The import will bring `C2` into the lexical scope prioritizing `C2` over `C1`.

Further, implicits defined in the lexical scope follow the name binding rules and thus can be shadowed by explicit local definitions. For instance, adding any definition with a name `c2` (e.g., `val c2 = null`) into the scope before the call to `f` will result in returning again 1, since the imported `O.C2` implicit will be shadowed by this local definition. In the case `C1` did not exits, the compiler will simply emit “*value f is not a member of A*” error. To avoid this, library authors try to obfuscate the implicits names which in turn affects the ergonomics. A notable example is in the Scala standard library where the implicit providing a proof that two types are in a sub-type relationship is named `$conforms` in order to prevent a potential shadowing with a locally defined `conforms` method.⁵

2.5 Overheads

Implicit resolution together with macro expansion can sometimes significantly increase compilation time. To illustrate the problem, consider the JSON serialization of algebraic data types using the `circe.io`⁶, a popular JSON serialization library. We define two ADTs: `case class A(x: String)` and `case class B(xs: List[A], ys: List[A])`, and a method to print out their JSON representation:

```
def print(a: A, b: B) = println(a.asJson, b.asJson)
```

The `asJson` method is an extension method defined in the `circe.io` as `def asJson(implicit encoder: Encoder[T]): Json`. It uses an implicit parameter of type `Encoder[T]` effectively limiting its applicability to instances that define corresponding encoder. For the code to compile, two encoders `Encoder[A]` and `Encoder[B]` that turn `A` and `B` into `Json` are needed. The `circe.io` library gives three options for creating the encoder: manual, semi automated and automated.

The manual encoding involves implementing the single method in `Encoder`, manually creating an instance of `Json` with the appropriate fields (*cf.* Listing. 4a). While simple, it is a boilerplate code. The semi-automated solution delegates to `derivedEncoder` that synthesizes the appropriate type at compile time through implicit type class derivation and macros (*cf.* Listing. 4b). The fully

⁵Reported in Scala issue #7788, *cf.* <https://github.com/scala/bug/issues/7788>

⁶*cf.* <https://github.com/circe/circe>

```

object manual {
  implicit val eA: Encoder[A] = (a: A) =>
    obj("x"=>str(a.x))
  implicit val eB: Encoder[B] = (b: B) =>
    obj("xs" -> arr(b.xs.map(_.asJson)),
        "ys" -> arr(b.ys.map(_.asJson)))
}

object semiauto {
  import io.circe.generic.semiauto._

  implicit val eA: Encoder[A] = deriveEncoder[A]
  implicit val eB: Encoder[B] = deriveEncoder[B]
}

import io.circe.generic.auto._

(c) automated

```

Fig. 4. Type class derivations

automated solution, does not require extra code at the client side beside importing its machinery (*cf.* Listing. 4c). Compile time is affected by the choice of approach; taking the manual as a base line, semi-automated is 2.5x slower and automated is 3.8x slower.

The reason for this compile-time slow-down is the increase in the number of implicit resolutions triggered and macro expansion as shown in Table 1. The difference between the automated and semi-automated is that the former *caches* the derived instances in the implicit values `eA` and `eB` and so the `eA` which is synthesized before `eB` will be reused for deriving `eB`. The automated derivation synthesizes new instances for each application. In this simple example, it generates 140 additional lines of code at the `println`⁷. Caching of derived type classes was already reported to significantly improve the compilation time of various projects [Cantero 2018; Torreborre 2017]. One difficulty is that since the implicit scope is *invisible*, it is harder to figure out which implicits are derived where and which are causing slowdowns. Currently, the only way is to use a *scalac-profiling*⁸, compiler plugin which outputs more detailed statistics about implicit resolution and macro expansion.

Table 1. Count of implicit resolutions and macro expansions, and timing of the typer phase in scalac 2.12.8 with -Ystatistics:typer flag.

	Implicits	Macros	Time
Manual	13	0	.1s
Semi	35	51	.3s
Auto	52	78	.5s

3 SCALA ANALYSIS PIPELINE

We have implemented a data analysis pipeline targeting large-scale analysis of Scala programs. To the best of our knowledge, this is the only pipeline able to scale to thousands of projects. Our infrastructure can be extended for other analyses and it is available in open source.

Figure 5 gives an overview of the pipeline; every step shown in the figure is fully automated. The first step is to download projects hosted on GitHub. Next, gather basic metadata and in particular infer the build system each project uses. Incompatible projects are discarded in the next step. These are projects that do not meet the technical requirements of the analysis tools. The fourth step is to use the DéjàVu tool [Lopes et al. 2017] to filter out duplicate projects. The fifth step is to attempt to compile the corpus and generate semantic information. The final step is to load the extracted data and analyze them. The pipeline is run in parallel using GNU parallel [Tange et al. 2011] but the analysis is resource intensive. On our server (Intel Xeon 6140, 2.30GHz with 72 cores and 256GB of RAM) we were not able to compile more than 12 projects in parallel.

⁷Measured in the expanded code obtained from -Xprint:typer compiler flag

⁸ cf. <https://github.com/scalacenter/scalac-profiling>

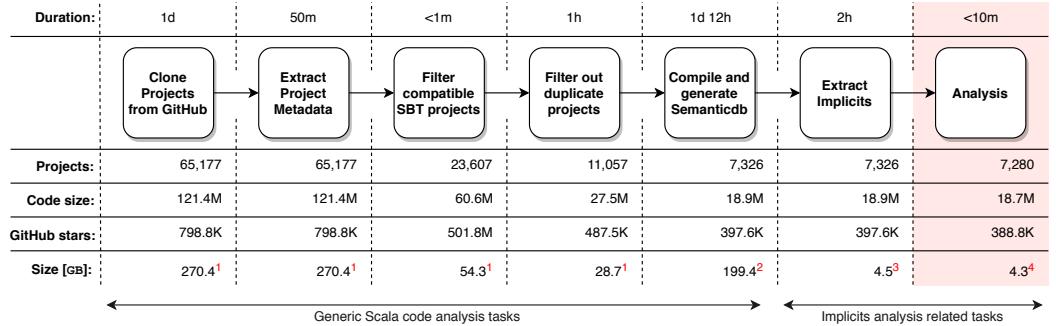


Fig. 5. Scala Analysis pipeline. (1) is the size of source code, (2) is the size of source plus compiled code and generated SEMANTICDB, (3) is the size of extracted implicits data model, (4) is the size of exported CSV files. The code size include tests.

The pipeline is reusable for other semantic analyses on Scala code bases, as only the last two steps relate specifically to implicits. At the end of the *Compile and generate SEMANTICDB* task, the corpus contains built projects with extracted metadata and SEMANTICDB files—these SEMANTICDB files contain syntactic information as well as semantic information (Scala symbols and types).

The pipeline logs all the steps for each project and provide an aggregated summaries. The analysis is done in R, and even though it is possible to load Google Protocol Buffers into R, it is not practical. Thus, we first aggregate the extracted data and export them into CSV format, which is more natural to work with in R. This is implemented in ~500 lines of make files and ~5K of R code. The implicit extractor is written in ~7.2K lines of Scala code.

The pipeline uses SCALAMETA⁹, a library that provides a high-level API for analyzing programs. One part of this library is a compiler plugin that for each compilation unit produces a data model with syntactic and semantic information. This includes a list of defined and referenced symbols as well as synthetic call sites and parameters injected by the compiler. The result is stored in a binary SEMANTICDB¹⁰ file (in Google Protocol Buffer serialization format). It can also extract symbol information from compiled classes allowing us to find implicits defined in external project dependencies. Note that this analysis would have not been possible with only syntactic information; compile-time information like types is required to match up call site and declaration site due to the fact that implicits themselves are type-directed rewritings performed by the compiler at type-checking time.

Based on this we have built a tool that extract implicit declarations and call sites. There are two limitations with SCALAMETA: it is limited to certain versions of Scala (2.11.11 in the 2.11 branch and 2.12.4 in the 2.12 branch), and it does not support *white-box macros* (*i.e.*, macros without precise signatures in the type system before their expansion) [Burmako 2017].

Another thing to consider when using SEMANTICDB is that it requires compiling the projects. The Scala compiler is about an order of magnitude slower than a Java compiler¹¹ and the SEMANTICDB compiler plugin adds additional overhead. For our analysis SBT is used to rebuild each project three times. There is no easy way around this. As noted above, lightweight, syntax-based approaches using regular expressions or pattern matching over AST nodes would not work because the call

⁹ cf. <https://scalameta.org/>

¹⁰ cf. <https://scalameta.org/docs/semanticdb/specification.html>

¹¹ cf. <https://stackoverflow.com/a/3612212/219584>

sites that use implicits are not visible in the source/AST, and to identify these patterns requires resolving terms and types from the declaration- and use-sites.

Scala projects are compiled by build tools which are responsible for resolving external dependencies. We chose SBT as it is the most-used tool in the Scala world. Since version 0.13.5 (August 2014), it supports custom plugins which we use to build an extractor of metadata. Next to the version information and source folder identification, the extracted metadata gives us information about project internal and external dependencies. This is necessary for assembling project's class-path that is used to resolve symbols defined outside of the project.

3.1 Implicit Extraction

The SEMANTICDB model contains low-level semantic information about each compilation unit. This includes synthetics, trees added by compilers that do not appear in the original source (e.g., inferred type arguments, for-comprehension desugarings, `C(...)` to `C.apply(...)` desugarings, implicit parameters and call sites). These trees are defined as transformations of pieces of the original Scala AST and as such they use quotes of the original sources. For example, the following Scala code:

```
import ExecutionContext.global; Future(1)
```

will have two synthetic trees injected by the compiler:

- `ApplyTree(OriginalTree(1,60,1,86), IdTree("EC.global"))`
- `TypeApplyTree(SelectTree(OriginalTree(1,60,1,83), IdTree("Future.apply()")), TypeRef("Int"))`

In this form, SEMANTICDB is not convenient for higher-level queries about the use of implicits. In order to do this, we transform SEMANTICDB into our own model that has declarations and call sites resolved. This is done in two steps. First, we extract implicit declarations by traversing each compilation unit and collecting declarations with the `implicit` modifier. For each declaration, we resolve its type using the symbol information from the SEMANTICDB and the project class path. This is done recursively in the case the declaration type has parents. Next, we look into the synthetic trees and extract inserted implicit function applications. Together with the project metadata, both declaration and call sites are stored in a tree-like structure using the Google Protocol Buffer format.

In our example, the extractor will produce 13 declarations and one implicit call site including:

```
// def apply[T](body : => T)(implicit executor : EC)
- Declaration("Future.apply()", DEF, ret=Ref("Future.apply().[T]"), params=List(
  ParamList(Param("body", Ref("Future.apply().[T]"))),
  ParamList(Param("executor", Ref("EC"), isImplicit=true))))
// implicit val global : EC
- Declaration("EC.global", VAL, ret=Ref("EC",List()), isImplicit=true)
  // Future . apply [Int ](1) (EC. global )
- CallSite("Future.apply()", typeArgs=Ref("Int"), implicitArgs=Ref("EC.global"))
```

Such model can be queried using the standard Scala collection API. For example, we can list a project's `ExecutionContext` declarations and the corresponding call sites that use them as follows:

```
val declarations = proj.declarations filter (dcl =>
  dcl.isImplicit && dcl.isVal && dcl.returnType.isKindOf("EC"))
val callsites = {
  val ids = declarations.map(_.declarationId).toSet
  proj.implicitCallsites filter (cs =>
    cs.implicitArguments exists (arg => ids contains arg))
}
```

The extractor is run per project in parallel and the results are merged into one binary file. This file can be streamed into a number of processors that export information about declarations, call sites, implicit conversions and implicit parameters into CSV files.

4 PROJECT CORPUS

For this paper we analyzed 7,280 projects consisting of 18.7M lines of Scala code (including 5.9M lines of tests and 2.2M lines of generated code). Most projects are small, the median is 677 lines of code, but the corpus also includes projects with over 100K lines of source code. 4,197 projects use Scala 2.11 but they account for less code (43.8%) and fewer stars (33.7%). For the remainder of the paper we partition our corpus in four categories: **small apps** are project with fewer than 1,000 LOC, **large apps** are projects with more than 1,000 LOC, **libraries** are projects that are listed on Scaladex. We also extract the test code from all projects into the **tests** category. Scaladex is a package index of projects published in Maven Central and Bintray repositories. These labels are somewhat ad-hoc as there is not always a strong reason behind the addition of a project to Maven Central or Bintray. However, manual inspection suggests that most of the projects that appear on Scaladex are intended for reuse.

Table 2. Project categories

Category	Projects	Code size	GitHub stars	Commits
Small apps.	3.3K	1M (mean=0.3K)	28K (mean=8)	139K (mean=41)
Large apps.	1.3K	5M (mean=4.0K)	74K (mean=57)	425K (mean=325)
Libraries	2.6K	6M (mean=2.4K)	285K (mean=108)	712K (mean=271)
Tests	5.4K	5M (mean=1.1K)	-	-

Figure 6 shows all projects, the size of the dots reflects number of stars, the color their category (large/small apps or libraries), the x-axis indicates the number of lines of code (excluding 5.9M lines of tests) in log scale, the y-axis gives the number of commits to the project in log scale. Solid lines indicate the separation between small and large applications. Dotted lines indicate means.

The corpus was obtained from publicly available projects listed in the GHTorrent database [Gousios 2013] and Scaladex. The data was downloaded between January and March 2019. We started with 65,177 non-empty, non-fork projects, which together contained 121.4M lines of code. We filtered out projects that were not compatible with our analysis pipeline (e.g., projects using early versions of Scala) and removed duplicates. 43K use SBT as their build system (other popular build systems are Maven with 5.1K projects and Graddle with 1.5K). From the SBT projects, 23.6K use SBT version 0.13.5+ or 1.0.0+ that is required by our analysis. We thus discarded about half of the downloaded code.

For duplicates, the problem is that even without GitHub forks, the corpus still contained unofficial forks, *i.e.*, copies of source code. For example, there were 102 copies of spark. Since spark is the largest Scala project (over 100K LOC), keeping them would significantly skew the subsequent analysis as 37.6% of the entire data set would be identical. In general, getting rid of duplicate projects is difficult task as one needs to determine the origins of individual files. We use that following criteria to retain a project: (1) it must have more than one commit, (2) it must be active for at least 2 months, (3) it must be in Scaladex or have less than 75% of file-level duplication or more than 5 stars on GitHub, and (4) it must be in Scaladex or have less than 80% duplication or more than 500 stars on GitHub. These rules were tuned to discard as many duplicates as possible while keeping originals. While large numbers of GitHub stars do not necessarily mean that a project widely-used, originals tend to have higher star counts than copies. The actual thresholds were chosen experimentally to make sure we keep all the bigger (> 50K LOC) popular Scala projects without any duplicates. We excluded 12,550 projects (33.1M lines of code). While this is over half of the source code from the compatible SBT projects, we lost fewer than 2.8% stars.

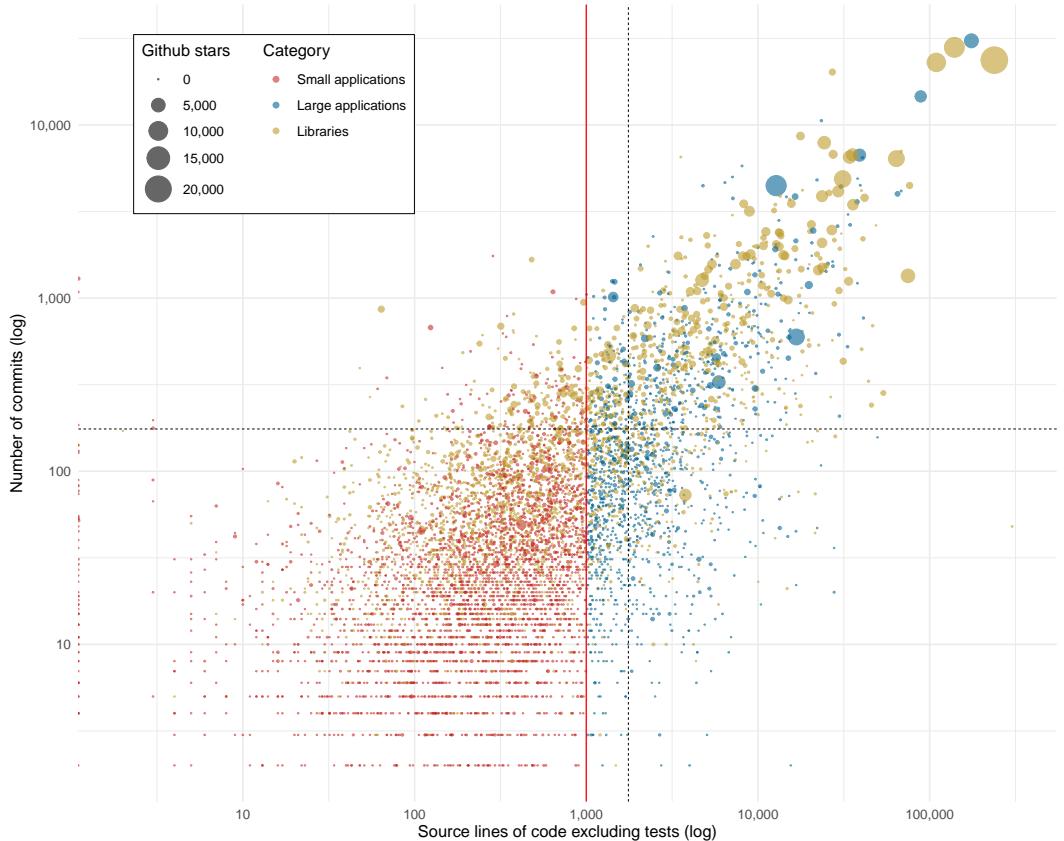


Fig. 6. Corpus overview

From the resulting 11,057 projects, we were able to successfully compile 7,326 projects. 3,731 projects failed to build. We follow the standard procedure of building SBT projects. If a project required additional steps, we marked it as failed. The following are the main sources of failures:

- *Missing dependencies* (2.1K). Most missed dependencies were for scalajs (964), a Scala-to-JavaScript compiler with a version that was likely removed because of security vulnerabilities. The next most frequent issue was due to snapshot versions (263) that were no longer available. The remainder were libraries that were taken down or that reside in non-standard repositories. Following common practice, we use a local proxy that resolves dependencies. No additional resolvers were configured. The proxy downloaded 204K artifacts (110GB).
- *Compilation error* (873). Some commits do not compile, and others fail to compile due our restriction on Scala versions. SCALAMETA requires Scala 2.11.9+ or 2.12.4+. Some projects are sensitive even down to the path version number. Some of these version upgrades might have also caused the missing dependencies in case the required artifact was built for a particular Scala version.
- *Broken build* (189). The SBT could not even start due to errors in the build.sbt.
- *Empty build* (156). Running SBT did not produce class files, leaving the projects *empty*. This happens when the build has some non-standard structure.

Finally, in the analysis, we discarded 46 projects (1.1% of the code) because some of their referenced declarations were not resolvable (the SCALAMETA symbol table did not return any path entry) and inconsistencies in SEMANTICDB. Table 3 lists some of the top rated projects that were included in the final corpus, including number of stars, lines of code, number of commits, level of duplication, Scala version and whether it is listed in Scaladex.

Table 3. Top 40 open source projects

Project	GitHub stars	Code size	Commits	Duplication	Scala version	Scaladex
apache/spark	21,067	238,062	23,668	0.4	2.12.8	Y
apache/predictionio	11,696	12,764	4,461	0	2.11.12	N
scala/scala	11,386	139,300	28,062	0.9	2.12.5	Y
akka/akka	9,666	109,359	22,966	0.001	2.12.8	Y
gitbucket/gitbucket	7,612	31,144	4,874	0	2.12.8	Y
twitter/finagle	7,003	63,976	6,386	0.01	2.12.7	Y
yahoo/kafka-manager	6,958	16,733	596	0.5	2.11.8	N
ornicar/lila	5,218	175,054	30,617	0.01	2.11.12	N
rtyley/bfg-repo-cleaner	5,014	1,351	465	0	2.12.4	Y
linkerd/linkerd	4,910	74,775	1,344	0.003	2.12.1	Y
fpinscala/fpinscala	4,244	5,914	327	1	2.12.1	N
haifengl/smile	4,242	4,731	1,271	0	2.12.6	Y
gatling/gatling	4,151	24,322	7,900	0	2.12.8	Y
scalaz/scalaz	4,079	34,146	6,523	0	2.12.8	Y
mesosphere/marathon	3,823	39,097	6,694	0.03	2.12.7	N
sbt/sbt	3,782	35,574	6,726	0.4	2.12.8	Y
twitter/diffy	3,375	3,778	73	0	2.11.7	Y
lampepfl/dotty	3,278	88,680	14,616	0.3	2.12.8	N
twitter/scalding	3,113	29,346	4,133	0	2.11.12	Y
typelevel/cats	3,093	23,607	3,878	0.009	2.12.7	Y
scalanlp/breeze	2,816	35,747	3,461	0.002	2.12.1	Y
scalatra/scalatra	2,382	8,914	3,174	0.3	2.12.8	Y
netflix/atlas	2,288	22,474	1,450	0	2.12.8	Y
spark-jobserver/spark-jobserver	2,286	7,403	1,571	0.3	2.11.8	Y
twitter/util	2,243	26,927	2,472	0.2	2.12.7	Y
slick/slick	2,188	23,622	2,084	0	2.11.12	Y
laurilehmijoki/s3_website	2,178	1,435	1,014	0	2.11.7	N
twitter/summingbird	2,011	9,057	1,790	0.3	2.11.12	Y
MojoJolo/texteaser	1,942	420	49	0	2.11.2	N
twitter/finatra	1,888	14,071	1,772	0.001	2.12.6	Y
twitter/algebird	1,836	23,676	1,502	0	2.11.12	Y
scala-exercises/scala-exercises	1,775	5,398	1,570	0	2.11.11	Y
circe/circe	1,633	8,140	1,749	0.006	2.12.8	Y
datastax/spark-cassandra-connector	1,569	11,120	2,418	0.2	2.11.12	Y
rickynils/scalacheck	1,480	4,038	1,091	0	2.12.6	Y
monix/monix	1,466	33,749	1,251	0	2.12.8	Y
http4s/http4s	1,459	27,412	6,765	0.003	2.12.7	Y
sangria-graphql/sangria	1,442	14,999	975	0.2	2.12.7	Y
spotify/scio	1,439	20,477	2,659	0.002	2.12.8	Y
coursier/coursier	1,417	13,313	1,984	0	2.12.8	Y

5 ANALYZING IMPLICITS USAGE

This section presents the results of our analysis and paints a picture of the usage of implicits in our corpus of Scala programs. We follow the structure of Section 2 and give quantitative data on the various patterns and idioms we presented including details about how identified them. We further discuss the impact of implicits on code comprehension and compilation time.

Identifying implicits requires performing a number of queries on the data files produced by our pipeline. Doing this also turned out to be necessary to remove duplication due to compilation artifacts. These come from projects compiled for multiple platforms and projects compiled for multiple major versions of Scala. While the main compilation target for Scala projects is Java byte-code (7,075 projects), JavaScript and native code are also potential targets. To prevent double counting, we make sure that shared code is not duplicated. Since Scala 2.11 and 2.12 are not binary compatible, libraries supporting both branches cross compile to both versions. We take care to compile only to one version.

In the remainder of this paper, when we refer to the “Scala library,” “Scala standard library,” or sometimes just to “Scala” we mean code defined in `org.scala-lang:scala-library` artifact.

Overview of Results. Out of the 7,280 analyzed projects, 7,148 (98.2%) have at least one implicit call site. From over 29.6M call sites in the corpus (explicit and implicit combined), 8.1M are call sites involving implicits. Most of these calls are related to the use of implicit parameters (60.3%). Figure 7 shows for each category a distribution of implicit call site ratios. The box is the 25th/75th percentiles and the line inside the box represents the median with the added jitters showing the actual distribution. For applications and libraries, the median is similar. It is smaller ~17.1%. In the case of test code, it is more than double, 38%. There tend to be more implicit call sites in tests than in the rest of the code. That is not surprising because the most popular testing frameworks heavily rely on implicits. Across the project categories the median is 23.4% (shown by the dashed line)—*i.e., one out of every four call sites involves implicits.*

Figure 8 shows the distribution of the declarations that are being called from the implicit call sites. There is a big difference between the test and non-test category. In the case of the both applications and libraries, most implicits used come from the standard library, followed by their external dependencies. The main sources of implicits in Scala are collections, concurrency and reflection packages together with the omnipresent `scala.Predef` object.

The collections are used by 80.3% projects (from 96.6% in large apps to 44.6% in tests). Most of the collection transforming operations such as `map`, use a builder factory passed as an implicit parameter `CanBuildFrom`. 38.3% of all implicit call sites involving methods that use this implicit parameter appear in libraries. Implicit parameters are used for reflection. Instances of `Manifest`, `ClassTag` or `TypeTag` classes can be requested from the compiler to be passed as implicit arguments, allowing one to

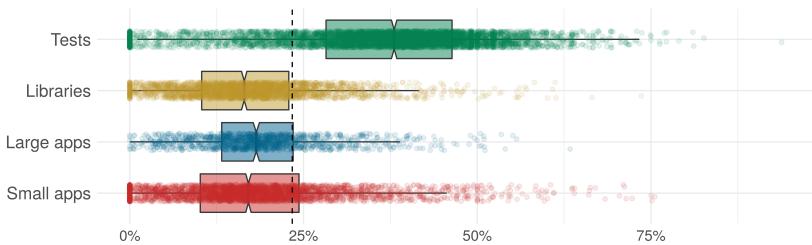


Fig. 7. Ratio of implicit call

get in-depth information about the type parameters of a method at runtime, circumventing the limitation of Java's type erasure. This is used a lot in large applications (90.2%). Less in libraries (61.7%) and small projects (56%) or tests (56.4%). Over half of all the large projects (58.5%) and third of libraries (32.1%) employ some concurrency routines from the Scala standard library. `scala.Predef` defines basic conversion like `String` to `StringOps` (extending the functionality of Java strings) or an arrow association, allowing one to use `a->b` to create a tuple of `(a,b)`. These are used by almost all the projects regardless of category.

Excluding the Scala standard library and testing frameworks, the rest of the implicits in the case of application and libraries come from a number of different external dependencies. There are some well known and projects with rich set of implicit usage such as the Lightbend/Typesafe stack with Play (a web-application framework, used in 5% of implicit call sites), Slick (object-relational mapping, 2.6%) or akka (an actor framework, 2.3%). These libraries define domain-specific languages which, in order to fit well in the host language yet to appear to introduce different syntactic forms, heavily rely on implicits. Next to a more flexible syntax (as compared to Java or C#), implicits are the main feature for embedding DSLs.

In the case of tests, the vast majority of implicits comes from project dependencies, which are dominated by one of the popular testing frameworks. These frameworks define DSLs in one form or another, striving to provide an API that reads like English sentences. For example a simple test:

```
"Monte Carlo method" should "estimate pi" in { MCarloPi(tries=100).estimate === 3.14 +- 0.01 }
```

contains six implicit call sites. Four are implicit conversions adding methods `should` to `String`, in to `ResultOfStringPassedToVerb` (the resulting type of calling the `should` method), `==` and `+-` to `Double`. Three of them additionally take implicit parameters for pretty-printing, source position (generated by a macro), test registration, and floating point operations. The implicit macro generating the source position is actually the single most used implicit parameter in the corpus with 912.7K

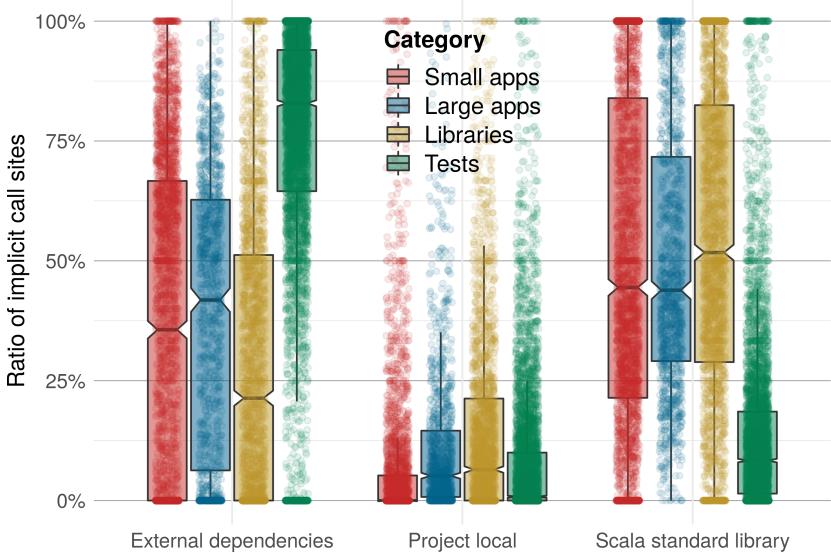


Fig. 8. Origin of parameter declarations

instances. Excluding the test frameworks, the ratio of implicit locations become very close to that of the main code, with collections and `scala.Predef` dominating the distribution.

5.1 Implicit Conversion

We recognize conversions by finding signatures that are either: (1) an `implicit def` with one non-implicit parameter (and 0+ implicit parameters) and a `non-Unit` return, or (2) an `implicit val`, `var` or `object` that extends a function type `T=>R` such that `R` is not `Unit`. Note, that `implicit class` declarations are already de-sugared into a class and a corresponding `implicit def`.

Table 4 summarizes conversions across the four categories of projects; X ($Y\%$ $Z\%$) are such that X is the number of occurrences, $Y\%$ is the ratio of X across all categories and Z is a ratio of projects identified in the given category. As expected, the majority of implicit conversions (80%) are defined in libraries (52% of libraries define at least one conversion) while most use is in the tests (61% of all implicit conversion call sites).

Table 4. Conversions

	Small Apps	Large Apps	Libraries	Tests
Declarations	2K (04% 22%)	7K (13% 58%)	49K (80% 52%)	2K (03% 11%)
Call sites	89K (04% 88%)	384K (15% 99%)	514K (20% 94%)	1M (61% 95%)

Table 5 lists the projects declaring and using the most conversions; each project's GitHub name is followed by its star rating, lines of code, and the number of occurrences. It is interesting to observe that the projects that define the most conversions are not necessarily the ones which use the most, as usage is likely correlated to project size.

Table 5. Top conversions

Project	Declarations	Project	Callsites
<code>shadaj/slinky</code> (265, 46K)	34K	<code>exoego/aws-sdk-scalajs-facade</code> (3, 302K)	130K
<code>pbaun/rere</code> (4, 14K)	446	<code>scalatest/scalatest</code> (782, 76K)	116K
<code>etorreborre/specs2</code> (642, 26K)	440	<code>apache/spark</code> (21K, 238K)	60K
<code>sisioh/aws4s</code> (7, 15K)	402	<code>akka/akka</code> (10K, 109K)	30K
<code>CommBank/grimlock</code> (29, 22K)	385	<code>gapt/gapt</code> (48, 68K)	22K
<code>scala/scala</code> (11K, 139K)	346	<code>ornicar/lila</code> (5K, 175K)	17K
<code>scalatest/scalatest</code> (782, 76K)	343	<code>psforever/PSF-LoginServer</code> (28, 41K)	15K
<code>scalan/special</code> (2, 33K)	336	<code>broadinstitute/cromwell</code> (384, 65K)	15K
<code>scalaz/scalaz</code> (4K, 34K)	301	<code>hmrc/tai-frontend</code> (0, 31K)	14K
<code>lift/framework</code> (1K, 42K)	280	<code>getquill/quill</code> (1K, 11K)	14K

Conversions are used in 96.8% of all projects (7,050). There are 2.5M implicit conversions or 31.5% of all implicit call sites. This is understandable as it is hard to write code that does not, somehow, trigger one of the many conversions defined in the standard library. In fact, for application code 47.4% of implicit conversions have definitions originating in the standard library. Most conversions, 61.1% to be exact, happen in tests; for those, 59.4% of them have definitions that originate from one of the two popular testing frameworks (`scalatest` or `specs2`). If we exclude the standard library and testing frameworks, most conversions are defined in imported code, only about 18.8% are calls to conversions with definitions local to their project.

In terms of conversion declarations, 41.1% of projects (2,991) provide 61,995 conversions (16.7% of all declarations) with a median of 3 per project and a s.dev of 615.5. As expected, testing frameworks have many declarations (343 in `scalatest`, 440 in `specs2`). We note that `slinky` defines over 33.6K

conversions (almost all programmatically generated). The reason is that this project aims at allowing one to writing React code (a JavaScript library for building user interfaces) in Scala in a similar manner as to that of JavaScript. This project is hardly used, we could find only 2 clients (with 3.6K LOC) that used 8 `slinky` conversions.

The most used conversion is `ArrowAssoc` as it enables users to create tuples with an arrow (e.g., `1 -> 2`). The next most popular is `augmentString`, a conversion that allows users to use index sequence methods on `String` objects. On average, projects targeting JavaScript use 2.5 times more often implicit conversions than JVM projects. Most of these conversions come from libraries that simplify front-end web development with DSLs for recurring tasks such as DOM construction and navigation. Only 1.1K (0.3%) of the implicit conversions were defined with functional types (*i.e.*, using `implicit val`, `var` or `object`); this is good as implicit values that are also conversions can be the source of problems.

5.2 Implicit Parameters

We record all method and constructor declarations with implicit parameter list. Table 6 summarizes parameters across the four categories of projects; X ($Y\%$ $Z\%$) are such that X is the number of occurrences, $Y\%$ is the ratio of X over all categories and Z is a ratio of projects in the given category.

Table 6. Parameters

	Small Apps	Large Apps	Libraries	Tests
Declarations	8K (06% 35%)	50K (32% 73%)	87K (55% 68%)	11K (07% 23%)
Call sites	134K (04% 89%)	749K (20% 99%)	691K (19% 94%)	2M (58% 95%)

Table 7 lists the projects declaring and using the most implicit parameters; each project's GitHub name is followed by its star rating, lines of code, and the number of occurrences. As with conversion, the projects that define the most implicits are not necessarily the ones with most calls.

Table 7. Top implicit parameters

Project	Declarations	Project	Callsites
lampepf/lotty (3K, 89K)	4K	scalatest/scalatest (782, 76K)	242K
scalaz/scalaz (4K, 34K)	4K	apache/spark (21K, 238K)	59K
typelevel/cats (3K, 24K)	3K	typelevel/cats (3K, 24K)	53K
robertofischer/hackrank (0, 50K)	2K	CommBank/grimlock (29, 22K)	52K
scalatest/scalatest (782, 76K)	2K	exoego/aws-sdk-scalajs-facade (3, 302K)	49K
sirthias/parboiled2 (604, 6K)	1K	akka/akka (10K, 109K)	43K
laserdisc-io/laserdisc (23, 7K)	1K	monix/monix (1K, 34K)	40K
slamdata/quasar (742, 27K)	1K	scalaz/scalaz (4K, 34K)	39K
etorreborre/specs2 (642, 26K)	984	slamdata/quasar (742, 27K)	31K
EHRI/ehri-frontend (10, 68K)	981	lampepf/lotty (3K, 89K)	29K

Calls sites with implicit parameters are frequent, they account for 46.2% (3.7M) of all Scala call sites. As shown in Table 6, tests account for 58% of these calls. Small applications have a lower proportion, most likely because they account for relatively few lines of code.

In terms of declarations, 78.2% of projects (5.7K) have over 370.7K implicit parameter declarations. The remaining projects do not declare any. The majority, 89.6% (332.2K), of declarations are public. Over half of the declarations come from 200 projects which often implement DSL-like APIs. This also happens internally in applications. For example, `ornicar/lila`, an open source chess server, is one of the largest and most popular apps in the corpus. It uses implicits for a small database management DSL.

5.3 Idioms and Patterns

In this subsection, we look at popular implicit idioms and answer the question how frequently are these idioms used. For each, we describe the heuristic used to recognize the pattern and give a table with the 10 top most projects in terms of declarations as well as in use in terms of call sites. Each of the table has the same structure: each project's GitHub name is followed by its star rating, lines of code, and the number of occurrences for declarations and call sites.

Table 8 gives a summary of the declaration and uses of the various idioms and patterns split by our code categories; X ($Y\%$ $Z\%$) are such that X is the number of occurrences, $Y\%$ is the ratio of X over all categories and Z is a ratio of projects in the given category.

Table 8. Idioms and patterns

Pattern	Small Apps	Large Apps	Libraries	Tests
Late Trait Implementation	278 (08% 04%)	968 (28% 15%)	2.1K (59% 14%)	177 (05% 01%)
Extension Methods	1.7K (09% 17%)	5.1K (28% 48%)	10.5K (57% 45%)	1.2K (06% 08%)
Type Classess	4.3K (05% 19%)	17.2K (21% 49%)	54.2K (67% 53%)	5.8K (07% 15%)
Extension Syntax Methos	1.3K (06% 09%)	4.3K (20% 28%)	13.9K (66% 31%)	1.6K (08% 06%)
Type Proofs	110 (06% 01%)	320 (18% 05%)	1.3K (73% 06%)	39 (02% 00%)
Context	5K (06% 25%)	34.9K (41% 62%)	39.2K (46% 50%)	5.7K (07% 14%)
Unrelated Conversions	672 (02% 07%)	2.3K (06% 26%)	38.1K (92% 20%)	441 (01% 03%)
Bidirectional Conversion	197 (17% 01%)	321 (28% 06%)	556 (49% 03%)	61 (05% 00%)

(a) Declarations

Pattern	Small Apps	Large Apps	Libraries	Tests
Late Trait Implementation	21.4K (07% 54%)	67.3K (22% 84%)	97.8K (31% 54%)	125.4K (40% 47%)
Extension Methods	40.9K (03% 68%)	207.7K (13% 95%)	250.7K (15% 82%)	1.1M (69% 90%)
Type Classess	99.4K (05% 86%)	502.2K (23% 99%)	544K (25% 92%)	1.1M (48% 88%)
Extension Syntax Methos	42.7K (03% 55%)	213.5K (16% 89%)	227.5K (17% 61%)	881K (65% 75%)
Type Proofs	1.7K (03% 19%)	10.6K (19% 61%)	14.9K (27% 44%)	28.8K (51% 19%)
Context	35.9K (02% 60%)	239.2K (14% 87%)	154.6K (09% 61%)	1.3M (75% 84%)
Unrelated Conversions	29.7K (07% 72%)	107.4K (25% 96%)	112.9K (26% 78%)	178.1K (42% 57%)
Bidirectional Conversion	1.9K (06% 13%)	7.9K (25% 42%)	8.8K (28% 26%)	13.2K (41% 13%)

(b) Call sites

5.3.1 Late Trait Implementation. Late traits are recognized by looking for `implicit def T=>R` where R is a Scala trait or JAVA interface. Technically, the same effect can be achieved with an `implicit class` extending a trait, but in all cases the implicit class adds additional methods, and thus is disqualified. As Table 8 shows there are only a few declarations of this pattern, mostly in libraries. Table 9 gives the top 10 projects using late traits.

Most conversions, 79.8%, are used between types defined in the same project. Conditional implementation account for 16.4% of this pattern. 19.7% convert JAVA types (from 176 different libraries). Focusing on the JDK, 53 conversions are related to I/O, 50 are from JAVA primitives and 27 involve time and date types. There are 990 conversions from Scala primitives with `String` (217) and `Int` (77) being the most often converted from.

5.3.2 Extension Methods. In general extension methods can be defined using both `implicit class` and `implicit def`. While the former is preferred, the latter is still being used. Since an `implicit def` can be also used for late trait implementation or to simply relating two types, we only consider

Table 9. Top late traits

Project	Declarations	Project	Callsites
lift/framework (1K, 42K)	152	exoego/aws-sdk-scalajs-facade (3, 302K)	49K
lampepf/dotty (3K, 89K)	106	scalatest/scalatest (782, 76K)	9K
etorreborre/specs2 (642, 26K)	94	akka/akka (10K, 109K)	6K
scala/scala (11K, 139K)	82	CommBank/grimlock (29, 22K)	4K
CommBank/grimlock (29, 22K)	81	hmrc/tai (1, 13K)	3K
scalatest/scalatest (782, 76K)	74	broadinstitute/cromwell (384, 65K)	3K
l-space/l-space (3, 17K)	68	maif/izanami (91, 19K)	2K
anskarl/auxlib (1, 1K)	63	etorreborre/specs2 (642, 26K)	2K
anskarl/LoMRF (58, 13K)	63	mattppap/mathematica-parser (24, 476)	2K
squreyl/squreyl (521, 9K)	49	playframework/play-json (193, 5K)	2K

`implicit def` with a return type that is neither a Scala trait nor a JAVA interface and that is defined in the same file as the conversion target because extension methods are usually collocated in either the same compilation unit or in the source file. We found 12,150 implicit classes, 65.3% of all extension methods. Table 8 shows that extension methods are widely used, they are defined across the corpus and in particular in large applications and libraries. Their use is widespread as well. The top 10 projects using extension methods appear in Table 10.

Table 10. Top extension methods

Project	Declarations	Project	Callsites
pbaun/rere (4, 14K)	428	scalatest/scalatest (782, 76K)	87K
etorreborre/specs2 (642, 26K)	295	exoego/aws-sdk-scalajs-facade (3, 302K)	46K
scalaz/scalaz (4K, 34K)	281	apache/spark (21K, 238K)	24K
scalan/special (2, 33K)	248	akka/akka (10K, 109K)	22K
lampepf/dotty (3K, 89K)	214	hmrc/tai-frontend (0, 31K)	14K
ritschwumm/scutil (6, 12K)	214	getquill/quill (1K, 11K)	13K
typelevel/cats (3K, 24K)	171	hmrc/tai (1, 13K)	13K
lift/framework (1K, 42K)	168	monix/monix (1K, 34K)	12K
broadinstitute/cromwell (384, 65K)	166	broadinstitute/cromwell (384, 65K)	10K
monsantoco/aws2scala (19, 10K)	134	hmrc/iht-frontend (1, 49K)	10K

There are 1.9K conditional extensions (10.2%). From these, 1.6K are related to type classes and 323 to contexts. 1.7K instances extends JAVA types (9.3%) across 676 libraries. Similarly to late traits, the JAVA I/O (224), date and time (200) and JAVA primitives (59) are the most often extended. Extension methods are also used to extends Scala primitives (3.7K), again `String` and `Int` being the most popular (1,169 and 452 respectively). This is understandable as these are the basic types for building embedded DSL.

5.3.3 Type Classes. We recognize type classes from their instances that are injected by a compiler as implicit arguments. What differentiate them from an implicit argument is the presence of type arguments linked to type parameters available in the parent context. This is what distinguishes a type class and a context. For example, the following do not match:

```
def f(x: Int)(implicit y: A[Int])      def f[T](x: T)(implicit y: T)
```

while the following do:

```
def f[T](x: T)(implicit y: A[T])      implicit class C[T](x: T)(implicit y: A[T])
```

We match implicit parameters with at least one type argument referencing a type parameter. Table 8 shows that type classes are the most widely declared pattern. Both libraries and large application

use it frequently. They are also the most frequent call sites. The top 10 projects using type classes are in Table 11.

Table 11. Top type classes

Project	Declarations	Project	Callsites
scalaz/scalaz (4K, 34K)	4K	scalatest/scalatest (782, 76K)	96K
typelevel/cats (3K, 24K)	3K	exoego/aws-sdk-scalajs-facade (3, 302K)	49K
robertofischer/hackerrank (0, 50K)	2K	typelevel/cats (3K, 24K)	48K
sirthias/parboiled2 (604, 6K)	1K	apache/spark (21K, 238K)	46K
slamdata/quasar (742, 27K)	1K	CommBank/grimlock (29, 22K)	43K
laserdisc-io/laserdisc (23, 7K)	1K	scalaz/scalaz (4K, 34K)	38K
scalatest/scalatest (782, 76K)	947	slamdata/quasar (742, 27K)	30K
twitter/algebird (2K, 24K)	899	laserdisc-io/laserdisc (23, 7K)	18K
scalanlp/breeze (3K, 36K)	887	scalaprops/scalaprops (226, 6K)	17K
nrinaudo/kantan.csv (244, 5K)	832	nrinaudo/kantan.csv (244, 5K)	16K

Type classes are involved in 30% of the implicit calls which use over 11K type classes. Type classes are dominated by the standard library (42%). As expected, most come from the collection framework, `scala.Predef` and the `math` library. Next are testing libraries (15%) followed by some of the most popular frameworks and libraries including `Typelevel cats` and `scalaz` that provide basic abstractions for functional programming, including a number of common type classes. These two libraries are used by almost 40% in the corpus.

5.3.4 Extension Syntax Methods. From extension methods we select instances that define implicit parameters that match out type class definition from Section 5.3.3. Summary is in Table 12.

Table 12. Top extension syntax methods

Project	Declarations	Project	Callsites
pbaun/rere (4, 14K)	428	scalatest/scalatest (782, 76K)	87K
etorreborre/specs2 (642, 26K)	295	exoego/aws-sdk-scalajs-facade (3, 302K)	46K
scalaz/scalaz (4K, 34K)	281	apache/spark (21K, 238K)	24K
scalan/special (2, 33K)	248	akka/akka (10K, 109K)	22K
lampepl/dotty (3K, 89K)	214	hmrc/tai-frontend (0, 31K)	14K
ritschwumm/scutul (6, 12K)	214	getquill/quill (1K, 11K)	13K
typelevel/cats (3K, 24K)	171	hmrc/tai (1, 13K)	13K
lift/framework (1K, 42K)	168	monix/monix (1K, 34K)	12K
broadinstitute/cromwell (384, 65K)	166	broadinstitute/cromwell (384, 65K)	10K
monsantoco/aws2scala (19, 10K)	134	hmrc/iht-frontend (1, 49K)	10K

We found 18.6K of syntax methods instances in 2.5K projects. Most of them are defining operations of generic algebraic data types.

5.3.5 Type Proofs. We recognize this pattern by select `implicit def` that take generalized type constraints, such as equality (`=:=`), subset (`<:<`) and application (`=>`) as implicit type parameters. Summary is in Table 13.

This revealed a very few projects (270) besides Scala itself and related projects (the new Scala 3 compiler). They define 1.6K methods taking type proofs as implicit parameters. Most of them are small applications which seem to be projects experimenting with type level programming. There are however interesting use cases. Manually inspecting the bigger projects we found common use cases, both are related to enforcing certain API restrictions at compile time. In one case (`scalajs-react`—another project bringing React application development into Scala), it is used to ensure that a given

Table 13. Top type proofs

Project	Declarations	Project	Callsites
scalatest/scalatest (782, 76K)	167	CommBank/grimlock (29, 22K)	5K
scalikejdbc/scalikejdbc (982, 13K)	91	akka/akka (10K, 109K)	3K
scalanlp/breeze (3K, 36K)	67	typelevel/cats (3K, 24K)	2K
mpollmeier/gremlin-scala (412, 2K)	54	outworkers/phantom (1K, 12K)	2K
playframework/play-json (193, 5K)	45	scalatest/scalatest (782, 76K)	2K
xuwei-k/applybuilder (7, 767)	42	sisioh/aws4s (7, 15K)	2K
japgolly/test-state (108, 6K)	39	laserdisc-io/laserdisc (23, 7K)	1K
NICTA/scoobi (487, 13K)	34	apache/spark (21K, 238K)	623
scoundrel-tech/scoundrel (0, 10K)	34	tixxit/framian (118, 7K)	546
twitter/scalding (3K, 29K)	34	scoundrel-tech/scoundrel (0, 10K)	541

method is called only once. Another instance (`finagle`, an RPC system) creates a type-safe builder pattern that throws a compile-time error in the case the constructed object is missing required field. In both cases authors used `@implicitNotFound` annotation to provide customized error message.

5.3.6 Context. Whether or not an implicit argument is an instance of the context pattern is hard to quantify, since it depends on intent. We recognize them by selecting implicit call sites that are neither labeled as a type class application nor as a type proof. Summary is in Table 14.

Table 14. Top context

Project	Declarations	Project	Callsites
lampepf/dotty (3K, 89K)	4K	scalatest/scalatest (782, 76K)	201K
scalatest/scalatest (782, 76K)	1K	apache/spark (21K, 238K)	38K
sirthias/parboiled2 (604, 6K)	1K	akka/akka (10K, 109K)	28K
EHRI/ehri-frontend (10, 68K)	779	monix/monix (1K, 34K)	27K
ornicar/lila (5K, 175K)	774	lampepf/dotty (3K, 89K)	26K
ponkotuy/MyFleetGirls (86, 26K)	717	CommBank/grimlock (29, 22K)	18K
Sciss/SoundProcesses (23, 13K)	715	hmrc/iht-frontend (1, 49K)	18K
sciss/fscape-next (6, 27K)	696	hmrc/tai-frontend (0, 31K)	17K
ruimo/store (5, 38K)	688	gapt/gapt (48, 68K)	16K
sciss/patterns (1, 8K)	620	twitter/finagle (7K, 64K)	13K

As expected, contexts are used heavily in projects such as Scala compiler (`dotty` is the new Scala compiler), `spark` or `akka`, *i.e.*, projects that are centered around some main context which is being passed around in number of methods. JAVA types are also used as context parameters. Together 50 types from JDK are used in 1.8K methods across 179 projects. The top used one is `SQLConnection` followed by interfaces from `java.io`. Scala primitive types are used in 1,044 methods in 159. Function types are also used as contexts (645 methods in 154 projects), providing a convenient way to define application counters, implicit filters and other default data processors.

5.3.7 Anti-pattern: Conversions. Unrelated conversions are public, top-level definitions defined outside of either from or to compilation units. We recognize them by selecting implicit conversions that are not block-local, or private, or protected and are not defined in the same compilation unit as the source or the target type. Summary is in Table 15.

There are 41.5K of unrelated conversions spanning across 1.2K projects (16.2%). Most of them (33.6K) belong to the already mentioned `slinky` projects bringing React apps development to Scala. They are used in 6.1K (83.9%) projects. If we change the query to only regard the same artifact then it drops too 1.9K conversions in 619 projects. There are some indication that unrelated conversions

Table 15. Top unrelated conversions

Project	Declarations	Project	Callsites
shadaj/slinky (265, 46K)	34K	apache/spark (21K, 238K)	21K
sisioh/aws4s (7, 15K)	402	gapt/gapt (48, 68K)	11K
CommBank/grimlock (29, 22K)	299	scalatest/scalatest (782, 76K)	8K
scala/scala (11K, 139K)	166	akka/akka (10K, 109K)	7K
etorreborre/specs2 (642, 26K)	130	CommBank/grimlock (29, 22K)	6K
squeryl/squeryl (521, 9K)	128	ornicar/lila (5K, 175K)	6K
scoundrel-tech/scoundrel (0, 10K)	113	ilya-klyuchnikov/tapl-scala (126, 13K)	3K
scala/scala-java8-compat (353, 4K)	112	scoundrel-tech/scoundrel (0, 10K)	3K
typelevel/cats (3K, 24K)	110	broadinstitute/cromwell (384, 65K)	2K
lift/framework (1K, 42K)	101	mattpap/mathematica-parser (24, 476)	2K

might be deprecated in the upcoming revision of the Scala language¹². The numbers here show, that these conversions are being defined, but they are usually in the scope of the same library. From the unrelated conversions, 1.6K from 552 projects involves Scala primitive types. They are present in all categories, but majority comes from libraries where they are used as building blocks for DSLs. Only a very few (81 in 47 projects) convert just between primitive types.

For the conversions that go both ways, we consider all pairs of such conversions that are defined in the same artifact and thus could be easily imported in the same scope.

We have identified 1.1K such conversions defined in 209 (2.9%) projects and used across 1.9K (26.5%) projects. As expected, this has matched all the Scala-Java collection conversions defined in the `scala.collection` package. They are used in 728 (10%) projects. This is significantly less than the recommended alternative using explicit `asJava` or `asScala` decorators that are being used by 22.4% of projects. 244 projects mix both approaches.

From a manual inspection of some of the other popular bi-directional conversion, we find that it is used in libraries that provide both Java and Scala API (e.g., `spark` or `akka`) allowing one to freely mix Java and Scala version of the classes. Some libraries use them to provide easier syntax for its domain objects (e.g., using a tuple to represent a cell coordinate), or lifting types from/to `scala.Option`. Another distinct category are conversions between many of the different date and time representations in both Java and Scala. We found only 129 bi-directional conversions involving primitive types, out of which 13 are only between primitives.

5.4 Complexity

One question we wanted to address was the amount of work performed by the Scala compiler. This is motivated by the need for the programmer to reverse engineer the compiler’s work to understand how to fix their code when an error is related to implicits. In terms of code size, if one were to sum up the length of the symbols inserted by the compiler at the various call sites that use implicit arguments, this would amount to 55M characters or about 3.5x the size of the entire Scala project. Figure 9 shows the distribution of injected implicit arguments into methods. We limit the graph to 10 injected arguments, but in practice there is a long tail. The measurements are obtained by inspecting each call site where implicit resolution is involved and counting arguments injected directly to the target function as well as arguments injected to nested calls needed for the implicit derivation. While the distribution has a long tail, going all the way to 5,695, the median is 1. At the extreme, the `xdotai/typeless` project is exploring type-level programming and has one call site that includes 5,695 nested implicit calls and value injection. Expressed in length of the injected code, that call site has the compiler inject 56.2K characters. Figure 10 shows the distribution of

¹² cf. <https://github.com/lampepfl/dotty/pull/2060>

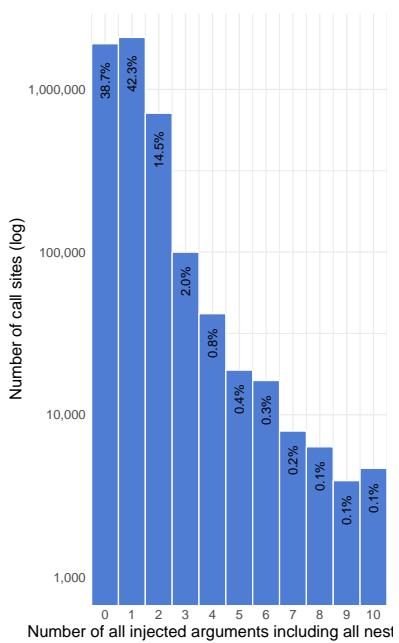


Fig. 9. Injected arguments

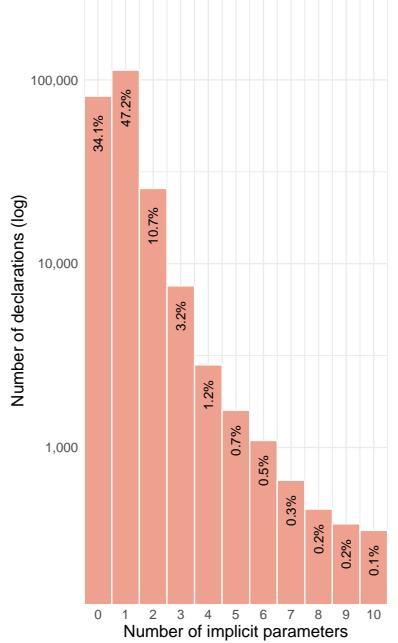


Fig. 10. Implicit parameters

the number of implicit parameter declarations. The data suggests that programmers are likely to encounter functions with one or two implicits rather frequently. And they are likely to deal with functions with four or more implicits several times per project.

To help navigate this complexity, the Scala plugin for IntelliJ IDEA has a feature that can show implicit hints, including implicit resolution in the code editor. This effectively reveals the injected code making it an indispensable tool for debugging. However, turning the implicit hints on severely hinders the editor performance creating a significant lag when working with implicits-heavy files. The second problem with this is that the IntelliJ Scala compiler is not the same as scalac and implicit resolution often disagrees between compiler implementations (e.g., IntelliJ does not consider implicit shadowing in lexical context). Another way to mitigate some of the complexity related to errors occurring during resolution is to customize the error message emitted when an implicit type is not found. Scala provides the `@implicitNotFound(message)` annotation to this end, where `message` can be parameterized with the names of type parameters that the type defines. In the corpus, we have found it defined 1.2K times in 436 projects, and used in 110.9K call sites.

5.5 Overheads

Another question we are interested to investigate is the effect of implicits on compile time. We have demonstrated that on a synthetic example, resolution can significantly impact type-checking performance. There are 1,969 (8.4M LOC) using Scala 2.12.4+ for which we can get compile time statistics using the `-Ystatistics:typer` compiler flag. Furthermore 488 projects (2.8M LOC) use the shapeless library which is the most common approach to guide the type class derivation [Cantero 2018]. The result of measuring compilation speed between these two sets of projects is shown in Figure 11. More precisely, the figure shows data for projects that have more than 1,000 lines of

code (for smaller projects compilation times may be dominated by startup costs). The x-axis shows the density of implicit call sites (their ratio per line of code, ranging between 0 and almost 2). The y-axis shows compilation speed measured in lines per second. For this figure we capture the entire compilation time of each project, including I/O. Higher is better on this graph. Colors distinguish projects that use type classes (red) from those who do not (blue). The lines indicate an estimate of the conditional mean function (loess). If implicits were not influencing compilation time, one would expect both lines to be roughly flat and at the same level. What we see instead confirms our hypothesis, the cost of compilation increases with the density of implicits and the use of type classes further reduce compilation speed.

Another manifestation can be found in the scalatest testing framework. It defines a `Prettifier` for pretty printing which looks like a perfect candidate for a type class, yet the authors have decided to use it as a context parameter instead. The reason given for that is performance: *"Prettifier is not parameterized ... because assertions would then need to look up Prettifiers implicitly by type. This would slow compilation."* In the corpus there are over 563.6K calls to methods using the `Prettifier` context. Resolving all of them implicitly using the implicit type class derivation machinery could indeed induce a slowdown across 2.5K projects.

5.6 Threats to Validity

We report on two source of threats to validity. One threat to *external validity* is linked to selection of code that was analyzed. We analyzed 15% of the Scala code publicly-available on GitHub. Our findings only generalize to industry if the code we analyzed is representative of industrial use of implicits. It is possible, for instance, that some companies enforce coding guidelines that impact the usage of implicits. We have no evidence that this is the case, but cannot rule it out. In terms

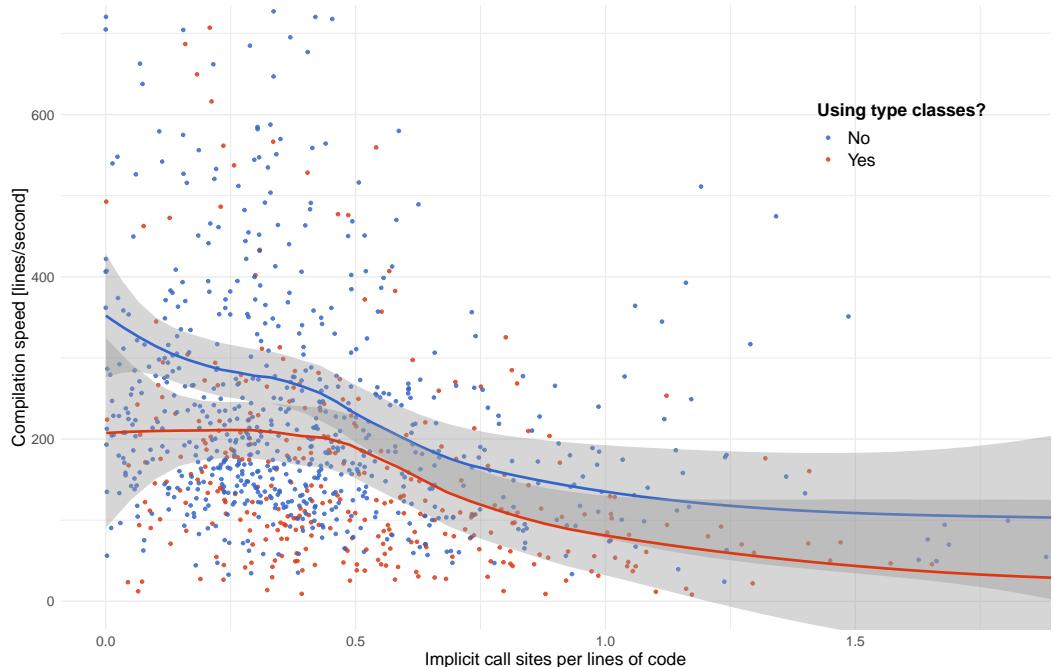


Fig. 11. Compilation slowdown

of threats to *internal validity* we consider our data analysis pipeline. It has several sources of inaccuracies. We rely on SCALAMETA to gather synthetic call sites. SCALAMETA restricts us to two Scala versions and it only generates metadata for about half of the selected projects. We are also aware that for 3% of implicit uses symbols could not be resolved.

6 RELATED WORK

The design of implicits as it appears in Scala is but one point in a larger space. While alternative designs are out of the scope of this work, we mention some important related work. Oliveira et al. [2010] established the connection between Haskell’s type classes and Scala implicits with multiple examples. Oliveira et al. [2012] formalized the key ideas of implicits in a core calculus. Rouvoet [2016] expanded the Oliveira et al. work and proved soundness and partial completeness independent of termination. Schrijvers et al. [2019] present an improved variant of the implicit calculus. One key property of this work is the notion of *coherence* (which is attributed to Reynolds [1991]). Coherence requires a program to have a single meaning, i.e. it precludes any semantic ambiguity. Scala eschews coherence in favor of expressivity by allowing overlapping implicits. Schrijvers et al. propose a design that recovers coherence.

There have been efforts to study how Scala is used by practitioners. Tasharofi et al. [2013] looked at how often and why Scala developers mix the actor model with other models of concurrency. They analyzed only GitHub 16 projects at the compiled byte-code level with a custom tool. The choice of byte-code had some drawbacks. For example, their analysis could not detect indirect method invocations and thus they had to supplemented it with manual inspection. The same corpus is used by Koster [2015] to analyze different synchronization mechanisms used in Scala code. Despite using the same projects, he analyzed 80% more lines of code as the projects were updated to their latest commit. The increase was mostly due to spark that grew from 12K to 104K lines of code. Unlike the previous study, he opted for source code analysis based on string matching. De Bleser et al. [2019] analyzed the tests of 164 Scala projects (1.7M LOC) for a diffusion of test smells. They used a similar way of assembling a corpus. While they started with 72K projects, but only managed to compile 2.9K projects. They discard projects with less than 1K LOC or without scalatest unit tests. For analysis, they also used semantic data from the SEMANTICDB.

Pradel and Sen [2015] analyzed the use of implicit type conversions in JavaScript. They use dynamic analysis running hundreds of programs including the common JavaScript benchmarks and popular real-world websites. In JavaScript, implicit type conversion is basically a type coercion. Despite that the coercion rules are well formalized, they are fairly complex and confuse even seasoned JavaScript developers. Unlike in Scala that has static type system, JavaScript uses implicit type conversion extensively (it is present in over 80% of the studied programs), yet the study finds that over 98% of the conversion is what the authors consider as harmless.

7 CONCLUSIONS

Implicits are a cornerstone of the Scala programming language. There is hardly any API without them as they enable elegant architectural design. They allow one to remove a lot of boilerplate by leveraging the compiler’s knowledge about the code. However, they can be also easily misused and if taken too far seriously hurt the readability of a code. Implicits are driven by type declarations. Thus, while, implicits are *used* transparently, with no indication in the program text, their application is guided by clear and precise rules. Our data shows that programmers have embraced them, with 98.2% of the projects we analyzed using them, and 78.2% of projects defining at least one implicit declaration. We also observed the prevalence of the idioms described, as most projects use them in some form. For implicit conversions, our results indicate that 96.8% of projects make use of them at some point, with the most popular conversions coming from the standard library and testing libraries. From the idioms we presented in this paper, type classes and extension methods are used extensively. Regarding conversions, most convert to and from types within the scope of the project. However, there is a number of conversions defined on unrelated types. While deprecating this form of conversion has been discussed, doing so would break 1.2K projects (16.2%) in our corpus.

Observations for language designers. We have seen many source of complexities related to the notion of coherence. Future designs of implicits should strongly consider adopting some limits to expressivity in order to improve code comprehension. A related point is to avoid relying on names of implicits during their resolution as this leads to subtle errors. Better tool support and static analysis could help diagnose performance problems and could help code comprehension, but it is crucial that IDEs and the Scala compiler agree on how resolution is to be performed.

Observations for library designers. Over-engineered libraries are hard to understand. It is worth considering the costs and benefits of adding, for example, type classes to an API. Asking questions such as “Is retroactive extension an important use case?” or “How much boilerplate can actually be avoided?” may help target the right use-cases for implicits. Often the key design issue is whether good defaults can be provided. When they cannot, the benefits of implicits decrease significantly. A good library design is one that lets regular users benefit without forcing them fully understand the cleverness that the library designer employed. Finally, we leave designers with the following unsolicited advice: Do not use unrelated implicits! Do not use conversions that go both ways! Do not use conversions that might change semantics!

Acknowledgments. Borja Lorente Escobar implemented an early version of the pipeline presented in this paper, we thank him for his contributions. We thank the reviewers for constructive comments that helped us improve the presentation. We thank Ólafur Páll Geirsson for his help with SEMANTICDB and SCALAMETA. We thank the members of the PRL lab in Boston and Prague for additional comments and encouragements. This work received funding from the Office of Naval Research (ONR) award 503353, from the European Research Council under the European Union’s Horizon 2020 research and innovation program (grant agreement 695412), the NSF (awards 1544542, and 1617892), and the Czech Ministry of Education, Youth and Sports (grant agreement CZ.02.1.010.00.015_0030000421).

REFERENCES

- Eugene Burmako. 2017. Unification of Compile-Time and Runtime Metaprogramming in Scala. (2017). <https://doi.org/10.5075/epfl-thesis-7159>
- Jorge Vicente Cantero. 2018. Speeding Up Compilation Time with scalac-profiling. <https://bit.ly/32gwTwP>
- Jonas De Blieser, Dario Di Nucci, and Coen De Roover. 2019. Assessing Diffusion and Perception of Test Smells in Scala Projects. In *International Conference on Mining Software Repositories (MSR)*. <https://doi.org/10.1109/MSR.2019.00072>
- Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Working Conference on Mining Software Repositories (MSR)*. <https://doi.org/2487085.2487132>
- Li Haoyi. 2016. Implicit Design Patterns in Scala. <https://web.archive.org/web/20180326160306/http://www.lihaoyi.com/post/ImplicitDesignPatternsinScala.html>.
- Joeri De Koster. 2015. *Domains: Language Abstractions for Controlling Shared Mutable State in Actor Systems*. Ph.D. Dissertation. Vrije Universiteit Brussel, Belgium.
- R Lämmel and K Ostermann. 2006. Software extension and integration with type classes. *Conference on Generative Programming and Component Engineering (GPCE)*. <https://doi.org/10.1145/1173706.1173732>
- Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. 2000. Implicit Parameters: Dynamic Scoping with Static Types. In *Symposium on Principles of Programming Languages (POPL)*. <https://doi.org/10.1145/325694.325708>
- Lightbend. 2018. Scala Developer Survey. <https://bit.ly/2Uk56sB>.
- Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *Proc. ACM Program. Lang.* 1, OOPSLA. <https://doi.org/10.1145/3133908>
- Chris Marshall. 2009. Is the Scala 2.8 collections library a case of the longest suicide note in history? <https://stackoverflow.com/questions/1722726/is-the-scala-2-8-collections-library-a-case-of-the-longest-suicide-note-in-hist>.
- Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. 2013. Instant Pickles: Generating Object-oriented Pickler Combinators for Fast and Extensible Serialization. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. <https://doi.org/10.1145/2509136.2509547>
- Heather Miller, Philipp Haller, and Martin Odersky. 2014. Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/978-3-662-44202-9_13
- Martin Odersky. 2017. What to leave implicit. <https://www.youtube.com/watch?v=Oij5V7LQJsA>. In *ScalaDays Chicago*.
- Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2017. Simplicity: foundations and applications of implicit function types. *PACMPL 2*, POPL. <https://doi.org/10.1145/3158130>
- Martin Odersky and Adriaan Moors. 2009. Fighting bit rot with types (experience report: Scala collections). In *Foundations of Software Technology and Theoretical Computer Science (FST TCS)*. <https://doi.org/10.4230/LIPIcs.FSTTCS.2009.2338>
- Bruno Oliveira C. d. S., Adriaan Moors, and Martin Odersky. 2010. Type classes as objects and implicits. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. <https://doi.org/10.1145/1869459.1869489>
- Bruno Oliveira C. d. S., Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. 2012. The implicit calculus: a new foundation for generic programming. In *Conference on Programming Language Design and Implementation (PLDI)*. <https://doi.org/10.1145/2254064.2254070>
- Andrew Phillips and Nermín Serifovic. 2014. *Scala Puzzlers*. Artima Inc.
- Michael Pradel and Koushik Sen. 2015. The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*. <https://doi.org/10.4230/LIPIcs.ECOOP.2015.519>
- John C. Reynolds. 1991. The coherence of languages with intersection types. *Theoretical Aspects of Computer Software* https://doi.org/10.1007/3-540-54415-1_70
- Arjen Rouvoet. 2016. *Programs for Free: Towards the Formalization of Implicit Resolution in Scala*. Master's thesis. TU Delft.
- Miles Sabin. 2019. Shapeless. <https://github.com/milessabin/shapeless>.
- Tom Schrijvers, Bruno C. d. S. Oliveira, Philip Wadler, and Koar Marntirosian. 2019. Cochis: Stable and Coherent Implicits. *Journal of Functional Programming* <https://doi.org/10.1017/s0956796818000242>
- Joshua D Suereth. 2013. Implicit Classes. <https://web.archive.org/web/20170922191333/https://docs.scala-lang.org/overviews/core/implicit-classes.html>.
- Ole Tange et al. 2011. Gnu parallel—the command-line power tool. *The USENIX Magazine* 36, 1.
- Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. 2013. Why Do Scala Developers Mix the Actor Model with other Concurrency Models?. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/978-3-642-39038-8_13
- Eric Torreborre. 2017. Achieving 3.2x Faster Scala Compile Time. <https://jobs.zalando.com/tech/blog/achieving-3.2x-faster-scala-compile-time/>
- Philip Wadler and Stephen Blott. 1989. How to Make ad-hoc Polymorphism Less ad-hoc. In *Symposium on Principles of Programming Languages (POPL)*. <https://doi.org/10.1145/75277.75283>