The Limitations of Type Classes as Subtyped Implicits (Short Paper)

Adelbert Chang
adelbertc@gmail.com

Abstract

Type classes enable a powerful form of ad-hoc polymorphism which provide solutions to many programming design problems. Inspired by this, Scala programmers have striven to emulate them in the design of libraries like Scalaz and Cats.

The natural encoding of type classes combines subtyping and implicits, both central features of Scala. However, this encoding has limitations. If the type class hierarchy branches, seemingly valid programs can hit implicit resolution failures. These failures must then be solved by explicitly passing the implicit arguments which is cumbersome and negates the advantages of type classes.

In this paper we describe instances of this problem and show that they are not merely theoretical but often arise in practice. We also discuss and compare the space of solutions to this problem in Scala today and in the future.

CCS Concepts • **Software and its engineering → Language features**; *Polymorphism*;

Keywords Scala, implicits, type classes, subtyping

ACM Reference Format:
Adelbert Chang. 2017. The Limitations of Type Classes as Subtyped Implicits (Short Paper). In *Proceedings of 8th ACM SIGPLAN International Scala Symposium (SCALA'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3136000.3136006

1 Introduction

Type classes provide a powerful form of ad-hoc polymorphism and are an essential tool in functional programming [Wadler and Blott 1989]. Instead of overloading functions on types, functions are parameterized over type classes. *Instances* of these type classes are defined for appropriate types and functions parameters are instantiated to these types at the call site. When the call site asks for a specific instantiation the type class resolver automatically searches through the dictionary of instances to ensure the appropriate instances are defined.

Scala programmers have sought to emulate type classes to leverage this kind of ad-hoc polymorphism. The natural encoding of type classes uses implicits for instance definition and resolution and subtyping for specifying type class relationships.

As a running example consider the (stubbed) encoding of the `Functor` and `Monad` type classes. Each type class becomes a trait, and relationships between type classes become subtype relationships. For example, every `Monad` gives rise to a `Functor`, so `Monad[F] extends Functor[F]`.

```
trait Functor[F[_]] { }
trait Monad[F[_]] extends Functor[F] { }
```

It is also possible to write functions abstracting over these type classes.

```
def needFunctor[F[_]: Functor]: Unit = ()
```

We can define instances of these type classes by constructing implicit values of the appropriate type, such as `Monad[Option]`.

```
implicit val monadOption: Monad[Option] = ...
```

`needFunctor` can then be called when `F` is instantiated to `Option`. This type checks because implicit resolution considers subtyping during its search, and since `Monad[Option]` subtypes `Functor[Option]` the search succeeds.

For that same reason, we can call functions parametric over a supertype from functions parametric over a subtype. For instance:

```
def needMonad[F[_]: Monad]: Unit =
  needFunctor[F]
```

Now consider the addition of another type class `Traverse`, any instance of which also implies an instance of `Functor`.

```
trait Traverse[F[_]] extends Functor[F] { }
```

We then wish to write a function parametric over both `Traverse` and `Monad` - perhaps we want to monadically traverse some data. Since we still have a `Monad[F]` in scope we expect the call to succeed as before.

```
def tAndM[F[_]: Traverse: Monad]: Unit =
  needFunctor[F]
```

This fails with the error "`ambiguous implicit values`." When the implicit resolver runs, it again looks for an instance of `Functor[F]`. However this time there are two instances, one through `Traverse` and another through `Monad`, but it

SCALA'17, October 22–23, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5529-2/17/10…$15.00
https://doi.org/10.1145/3136000.3136006

is because there are two instances that resolution fails. As we will see in section 3, in the context of type classes the underlying instance through either of these should be semantically equivalent. Therefore, either choice would be valid. Unfortunately in the more general system of implicits the resolver does not have this information, so instead of arbitrarily picking one it fails.

For simple cases like these the problem can be solved by explicitly passing in the implicit parameter, relieving the implicit resolver of the need to run at all.

```
def tAndMEx[F[_]: Traverse: Monad]: Unit =
  needFunctor[F](implicitly[Traverse[F]])
```

However, this just replaces the implicit resolver with the programmer which can quickly become cumbersome with more type classes and more complex hierarchies. In section 2 we'll see that for cases like syntax enrichments and for-comprehensions no similar solution exists; users simply lose functionality.

Variants of this problem can manifest whenever there are usages of branching type classes. For the rest of this paper we will explore and motivate these instances, and look into possible solutions to the problem.

- In section 2 we show use cases of type classes that hit variations on the problems above.
- We discuss properties of type classes in other languages that preclude them from having these problems in section 3.
- In section 4 we explore proposed solutions for Scala today and in the future.
- Existing languages like OCaml have also experimented with retrofitting implicits and type classes. We consider these in section 5.

2 Motivation

The problems above can manifest whenever a function is parametric over two or more type classes which share a super class. We now examine variations of such functions and observe the problems in these instances.

2.1 Branching Type Class Hierarchies

Ambiguous implicits can be avoided if we restrict ourselves to type classes that never branch. This is clearly an unreasonable restriction as there are many examples of branching yet useful type class hierarchies.

For example, the `Alternative` type class also extends `Functor`, adding the ability to "choose" between two given `F[A]`s. This kind of abstraction is very useful in the context of parsing [Haskell.org 2017b] where an input string can be one of many instances of a single type, as is the case when parsing into a sealed family of classes.

An even more common and powerful use case is "MTL-style programming[1]" [Jones 1995; Kmett 2017]. Types such as `Function1` and `Either` are often referred to as "effects" since they encode in their type classes instances behavior outside of computing a pure value. For instance, `Function1`'s instances treat it as a value computed from a read-only context and `Either`'s instances treat it as a possibly failed value. Programs will often work with a variety of such effects and different functions will return different combinations of effects. Composing these functions then becomes cumbersome since the programmer needs to shim between effect types - consider combining an `A => Option[B]` with an `Either[E, List[A]]`. MTL-style solves this by encoding effects as type classes and working parametrically with these classes. Composing effects then becomes accumulating constraints. For instance, the following shows the type class analogues for `Function1` and `Either`, as well as effectful functions using these classes.

```
trait MonadReader[F[_], Ctx] extends Monad[F] { }
trait MonadError[F[_], Err] extends Monad[F] { }

def readerOnly[F[_]](implicit
  F: MonadReader[F, Int]): Unit = ()

def errorOnly[F[_]](implicit
  F: MonadError[F, Throwable]): Unit = ()

def rAndE[F[_]](implicit
  F0: MonadReader[F, Int],
  F1: MonadError[F, Throwable]): Unit = {
  readerOnly[F]
  errorOnly[F]
}
```

However once we are using multiple effects it is quite easy to hit ambiguous implicits resolution. Consider what happens when `rAndE` calls a function parametric over `Monad`.

2.2 Syntax Enrichments

Libraries that provide type classes like Scalaz [Scalaz 2017a] and Cats [Typelevel 2017a] often provide syntax enrichments as well. For example, syntax for `Functor` may add a `map` method to any value of type `F[A]` where there is a `Functor[F]`. This syntax is provided with implicit classes and implicit (type class) constraints, shown below (using the Cats project for completeness).

```
import cats.Functor

// Reproduced from Cats
implicit class FunctorOps[F[_]: Functor, A]
  (fa: F[A]) {
    def map[B](f: A => B): F[B] =
      implicitly[Functor[F]].map(fa)(f)
```

[1] "MTL" refers to the Haskell monad transformer library .

```
}

def functorOnly[F[_]: Functor, A]
  (fa: F[A]): F[Unit] = fa.map(_ => ())
```

Because the enrichment requires a `Functor[F]` instance, the enrichment cannot be used when ambiguity arises.

```
import cats.{Traverse, Monad}

def noSyntax[F[_]: Traverse: Monad, A]
  (fa: F[A]): F[Unit] = fa.map(_ => ())
```

Usage of syntax in situations like `noSyntax` not only fails to compile but also gives the very misleading error message "`value map is not a member of type parameter F[A]`." Unlike function calls, this variation cannot be solved by passing in an argument explicitly. With ambigous implicits, syntax is lost. Users are forced to use method calls directly on an instance (again requiring the user to manually resolve) which is cumbersome and results in code that is difficult to read.

```
def syntax[F[_]: Traverse: Monad, A]
  (fa: F[A]): F[Unit] =
  implicitly[Traverse[F]].map(fa)(_ => ())
```

These failures are exacerbated with for-comprehensions which are commonly used when working with monadic effects. Consider the following for-comprehension:

```
for {
  a <- fa
  b <- fb
  c <- f(a, b)
} yield g(a, b, c)
```

This desugars to the less readable chain of `flatMap` and `map` calls.

```
fa.flatMap { a =>
  fb.flatMap { b =>
    f(a, b).map { c => g(a, b, c) }
  }
}
```

These `flatMap` and `map` methods are often provided as polymorphic syntax enrichments for monads and functors, allowing them to be used in a generic context. However, if this syntax is used in a context with ambiguous `Functor` or `Monad`, the enrichments are lost as before and users must resort to calling these methods directly on the instance.

```
val instance = implicitly[Monad[F]]

instance.flatMap(fa) { a =>
  instance.flatMap(fb) { b =>
    instance.map(f(a, b)) { c => g(a, b, c) }
  }
}
```

2.3 Chained Type Class Instances

Earlier we saw that implicit ambiguity in method calls can be resolved by passing in the argument explicitly. This solution gets trickier for more sophisticated usages as many type class instances are defined in a "backward-chaining" fashion. For instance the `Option` monad transformer `OptionT` requires a `Functor[F]` to define a `Functor[OptionT[F, ?]]`[2].

```
case class OptionT[F[_], A](value: F[Option[A]])

implicit def optionTFunctor[F[_]: Functor]:
  Functor[OptionT[F, ?]] = ...
```

A call to a function that needs to be resolved to `OptionT[F, ?]` in a context where there is an ambiguous `Functor[F]` needs to explicitly pass the argument to the instance, take the result, and pass it to the function. This approach is not sustainable as chained instances can get arbitrarily complicated with more constraints and more chaining.

3 Type Class Coherency

Having shown that type classes as subtyped implicits is insufficient, we look to see how languages with first-class support for type classes solve these problems.

In languages like Haskell [Haskell.org 2017a] and Rust[3] [Rust-lang.org 2017] type classes are a first-class feature with semantics different than that of Scala's implicits. One critical difference is the restriction that type class resolution must be coherent. Coherency refers to the general type system property that "every different valid typing derivation for a program leads to a resulting program that has the same dynamic semantics [Peyton Jones et al. 1997]." This means for a given superclass (e.g. `Functor[F]`), the resolution through any of its subclasses (e.g. `Traverse[F]` or `Monad[F]`) must return a semantically equal instance.

Both Haskell and Rust achieve coherency by restricting where type class instances can be defined. To define an instance of a type class `TC` for a type `A`, the instance must either be defined with the code for `TC` or with the code for `A`. Doing so has the effect that a given (type class, type) pair is globally unique, guaranteeting coherency even in dynamically linked programs.

Implicits in Scala are much more general than type classes as there are no restrictions on where or how many implicits can be defined. Consequently, the implicit resolver has no promise of coherency when picking implicits as it is possible for different parameters of the same supertype to have different semantics as subtypes.

[2]The ? syntax is enabled through the kind-projector compiler plugin available at https://github.com/non/kind-projector.
[3]Rust's type class-like mechanism is called "traits."

4 Potential Scala Solutions

Short of implementing a separate type class system, a solution to the ambiguous implicits problem in Scala must communicate coherency information to the implicit resolver. At the time of writing two such solutions have been proposed, one which uses an alternative encoding of type classes and another which actually modifies Scala to be aware of type classes.

4.1 The Scato Encoding

One way of communicating coherency information is to just assume it and solve the problem of guiding the implicit resolver down a particular path[4]. Scato [Cochard 2017] and Scalaz 8 [Scalaz 2017b] take this approach by replacing subtyping with implicit conversions. Implicit conversions, unlike subtyping, can be prioritized which when done carefully can guide the resolver down a particular path. For instance, the following is an example of how `Functor`, `Monad`, and `Traverse` might be written.

```
trait Functor[F[_]] { }
trait Monad[F[_]] { def functor: Functor[F] }
trait Traverse[F[_]] { def functor: Functor[F] }

trait Conversions1 {
  implicit def m2F[F[_]: Monad]: Functor[F] =
    implicitly[Monad[F]].functor
}

trait Conversions0 extends Conversions1 {
  implicit def t2F[F[_]: Traverse]: Functor[F] =
    implicitly[Traverse[F]].functor
}

object Prelude extends Conversions0
import Prelude._

def resolves[F[_]: Traverse: Monad] =
  implicitly[Functor[F]]
```

In Scala implicits in subclasses have higher priority than implicits in superclasses, so in `resolves` the `t2F` conversion is picked and implicit resolution succeeds.

4.2 Making Scala Type Class-aware

The Dotty[5] [LAMP@EPFL 2017a] team have proposed another solution which changes Scala itself, but allows for a working version of the subtyped implicits encoding. The solution introduces a new marker trait which would be used to distinguish between type classes and regular implicits, allowing the implicit resolver to change its behavior depending on which one it encounters. Type classes would then be checked for coherency, enabling the implicit resolver to pick a path in cases that would have otherwise been considered ambiguous. This marker trait also works nicely with the subtyped implicits encoding since the mechanism itself is checked through subtyping.

Such a change has implications regarding parametricity to ensure it is impossible to detect which path was taken to an instance of a superclass. Details of this proposal are still under consideration so we defer to the issue ticket [LAMP@EPFL 2017b] for more details.

4.3 Discussion

The proposals above solve the problem along different axes and on different timelines.

The Scato encoding, while unsafe, solves the problem in Scala today and is being used in projects like Scalaz 8 and cats-mtl [Typelevel 2017b]. However, it is a relatively young encoding with few experience reports. One concern is its syntactic overhead. Structuring implicit conversions in a way that ensures all conversions are defined yet avoids implicit ambiguities is difficult, especially when compared to the subtyping approach. This concern could be addressed with code generation tools such as macros.

Another concern in Scato is the performance cost of using implicit conversions instead of subtyping. Each superclass resolution through a subclass invokes a function call to convert the latter to the former, whereas with subtypes no such call is needed. Under the right conditions some of these function calls could be inlined either statically with an annotation or by the JIT compiler. A more thorough investigation is needed to measure the practical implications of using these implicit conversions.

The Dotty proposal describes a safer solution that keeps the concise, natural encoding of type classes in Scala. However the proposal is centered around Dotty, a research compiler for a future Scala. While the Dotty language is very close to Scala and a solution implemented for Dotty could be implemented for Scala, the proposal is still under experimentation.

5 Related Work

Prior research in this space has been in retrofitting type classes and implicits in existing languages.

In Dreyer et al. [2007] the authors describe an ML-like language which uses ML modules in a type class-like fashion. To solve potential problems in coherency, restrictions are placed on where type classes can be resolved. The result is a system which guarantees coherency within a module boundary, but may still be incoherent across modules.

White et al. [2014] discusses an extension of OCaml [OCaml.org 2017] which adds an implicits system similar to

[4] In practice this assumption while unsafe, tends to work, having been made for years in many type class-based projects.

[5] Dotty is a research compiler for experimenting with ideas for the future of Scala.

that of Scala's. Section 3.3 of the paper describes a "diamond problem," which is similar to the problems described in this paper. The solution given is similar to the solution being discussed for Dotty. Unfortunately, only an informal description of the system is given so it is hard to tell how modular implicits behave with the situations presented above.

6 Conclusion

We have shown that the natural type class encoding for Scala is inadequate for general usage, specifically with instance resolution in branching hiearchies. This shortcoming greatly impedes the ergonomics of function calls, syntax enrichments, for-comprehensions, and chained instances. If Scala wants to better support type classes, the current state of subtyping and implicits is not enough. We believe that type classes are a powerful tool useful for doing functional programming both in the small and in the large, and deserve a solution in a future Scala.

Acknowledgments

The authors wish to thank Lars Hupel for recommending the development of this paper from its original blog post form and for assisting with the submission process. We would also like to thank Jared Roesch, our shepherd Ilya Sergey, and the anonymous reviewers for their helpful comments.

References

Aloïs Cochard. 2017. aloiscochard/scato: An exploration of purely functional library design in Scala. (2017). https://github.com/aloiscochard/scato

Derek Dreyer, Robert Harper, and Manuel M.T. Chakravarty. 2007. Modular Type Classes. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Nice, France. https://people.mpi-sws.org/~dreyer/papers/mtc/main-short.pdf

Haskell.org. 2017a. Haskell Language. (2017). https://www.haskell.org/

Haskell.org. 2017b. Haskell/Alternative and MonadPlus. (2017). https://wiki.haskell.org/Typeclassopedia#Failure_and_choice:_Alternative.2C_MonadPlus.2C_ArrowPlus

Mark Jones. 1995. Functional Programming with Overloading and Higher-Order Polymorphism. *Springer-Verlag Lecture Notes in Computer Science* 925 (1995).

Edward Kmett. 2017. mtl: Monad classes, using functional dependencies. (2017). https://hackage.haskell.org/package/mtl

LAMP@EPFL. 2017a. Dotty. (2017). http://dotty.epfl.ch

LAMP@EPFL. 2017b. lampepfl/dotty #2047: Allow Typeclasses to Declare Themselves Coherent. (2017). https://github.com/lampepfl/dotty/issues/2047

OCaml.org. 2017. OCaml - OCaml. (2017). http://ocaml.org/

Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: an exploration of the design space. In *Haskell workshop*. Amsterdam. https://www.microsoft.com/en-us/research/publication/type-classes-an-exploration-of-the-design-space/

Rust-lang.org. 2017. The Rust Programming Language. (2017). https://www.rust-lang.org/en-US/

Scalaz. 2017a. scalaz/scalaz: An extension to the core Scala library for functional programming. (2017). https://github.com/scalaz/scalaz

Scalaz. 2017b. scalaz/scalaz series/8.0.x: An extension to the core Scala library for functional programming. (2017). https://github.com/scalaz/scalaz/tree/series/8.0.x

Typelevel. 2017a. typelevel/cats: Lightweight, modular, and extensible library for functional programming. (2017). https://github.com/typelevel/cats

Typelevel. 2017b. typelevel/cats-mtl: cats transformer type classes. (2017). https://github.com/typelevel/cats-mtl

Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Austin, Texas. http://homepages.inf.ed.ac.uk/wadler/topics/type-classes.html#class

Leo White, Frédéric Bour, and Jeremy Yallop. 2014. Modular implicits. In *ML Family/OCaml Users and Developers*. Gothenburg, Sweden. http://www.lpw25.net/ml2014.pdf