Contextual Abstractions

Critique of the Status Quo

Scala's implicits are its most distinguished feature. They are *the* fundamental way to abstract over context. They represent a unified paradigm with a great variety of use cases, among them: implementing type classes, establishing context, dependency injection, expressing capabilities, computing new types and proving relationships between them.

Following Haskell, Scala was the second popular language to have some form of implicits. Other languages have followed suit. E.g Rust's traits or Swift's protocol extensions. Design proposals are also on the table for Kotlin as compile time dependency resolution, for C# as Shapes and Extensions or for F# as Traits. Implicits are also a common feature of theorem provers such as Coq or Agda.

Even though these designs use widely different terminology, they are all variants of the core idea of *term inference*. Given a type, the compiler synthesizes a "canonical" term that has that type. Scala embodies the idea in a purer form than most other languages: An implicit parameter directly leads to an inferred argument term that could also be written down explicitly. By contrast, type class based designs are less direct since they hide term inference behind some form of type classification and do not offer the option of writing the inferred quantities (typically, dictionaries) explicitly.

Given that term inference is where the industry is heading, and given that Scala has it in a very pure form, how come implicits are not more popular? In fact, it's fair to say that implicits are at the same time Scala's most distinguished and most controversial feature. I believe this is due to a number of aspects that together make implicits harder to learn than necessary and also make it harder to prevent abuses.

Particular criticisms are:

1. Being very powerful, implicits are easily over-used and mis-used. This

observation holds in almost all cases when we talk about *implicit conversions*, which, even though conceptually different, share the same syntax with other implicit definitions. For instance, regarding the two definitions

```
implicit def i1(implicit x: T): C[T] = ...
implicit def i2(x: T): C[T] = ...
```

the first of these is a conditional implicit *value*, the second an implicit *conversion*. Conditional implicit values are a cornerstone for expressing type classes, whereas most applications of implicit conversions have turned out to be of dubious value. The problem is that many newcomers to the language start with defining implicit conversions since they are easy to understand and seem powerful and convenient. Scala 3 will put under a language flag both definitions and applications of "undisciplined" implicit conversions between types defined elsewhere. This is a useful step to push back against overuse of implicit conversions. But the problem remains that syntactically, conversions and values just look too similar for comfort.

2. Another widespread abuse is over-reliance on implicit imports. This often leads to inscrutable type errors that go away with the right import incantation, leaving a feeling of frustration. Conversely, it is hard to see what implicits a program uses since implicits can hide anywhere in a long list of imports.

3. The syntax of implicit definitions is too minimal. It consists of a single modifier, `implicit`, that can be attached to a large number of language constructs. A problem with this for newcomers is that it conveys mechanism instead of intent. For instance, a type class instance is an implicit object or val if unconditional and an implicit def with implicit parameters referring to some class if conditional. This describes precisely what the implicit definitions translate to -- just drop the `implicit` modifier, and that's it! But the cues that define intent are rather indirect and can be easily misread, as demonstrated by the definitions of `i1` and `i2` above.

4. The syntax of implicit parameters also has shortcomings. While implicit *parameters* are designated specifically, arguments are not. Passing an argument to an implicit parameter looks like a regular application `f(arg)`. This is problematic because it means there can be confusion regarding what parameter gets instantiated in a call. For instance, in

```
def currentMap(implicit ctx: Context): Map[String, Int]
```

one cannot write `currentMap("abc")` since the string `"abc"` is taken as explicit

argument to the implicit `ctx` parameter. One has to write `currentMap.apply("abc")` instead, which is awkward and irregular. For the same reason, a method definition can only have one implicit parameter section and it must always come last. This restriction not only reduces orthogonality, but also prevents some useful program constructs, such as a method with a regular parameter whose type depends on an implicit value. Finally, it's also a bit annoying that implicit parameters must have a name, even though in many cases that name is never referenced.

5. Implicits pose challenges for tooling. The set of available implicits depends on context, so command completion has to take context into account. This is feasible in an IDE but tools like Scaladoc that are based on static web pages can only provide an approximation. Another problem is that failed implicit searches often give very unspecific error messages, in particular if some deeply recursive implicit search has failed. Note that the Scala 3 compiler has already made a lot of progress in the error diagnostics area. If a recursive search fails some levels down, it shows what was constructed and what is missing. Also, it suggests imports that can bring missing implicits in scope.

None of the shortcomings is fatal, after all implicits are very widely used, and many libraries and applications rely on them. But together, they make code using implicits a lot more cumbersome and less clear than it could be.

Historically, many of these shortcomings come from the way implicits were gradually "discovered" in Scala. Scala originally had only implicit conversions with the intended use case of "extending" a class or trait after it was defined, i.e. what is expressed by implicit classes in later versions of Scala. Implicit parameters and instance definitions came later in 2006 and we picked similar syntax since it seemed convenient. For the same reason, no effort was made to distinguish implicit imports or arguments from normal ones.

Existing Scala programmers by and large have gotten used to the status quo and see little need for change. But for newcomers this status quo presents a big hurdle. I believe if we want to overcome that hurdle, we should take a step back and allow ourselves to consider a radically new design.

The New Design

The following pages introduce a redesign of contextual abstractions in Scala. They introduce four fundamental changes:

1. Given Instances are a new way to define basic terms that can be synthesized. They replace implicit definitions. The core principle of the proposal is that,

   rather than mixing the `implicit` modifier with a large number of features, we have a single way to define terms that can be synthesized for types.

2. Using Clauses are a new syntax for implicit *parameters* and their *arguments*. It unambiguously aligns parameters and arguments, solving a number of language warts. It also allows us to have several `using` clauses in a definition.

3. "Given" Imports are a new class of import selectors that specifically import givens and nothing else.

4. Implicit Conversions are now expressed as given instances of a standard `Conversion` class. All other forms of implicit conversions will be phased out.

This section also contains pages describing other language features that are related to context abstraction. These are:

- Context Bounds, which carry over unchanged.
- Extension Methods replace implicit classes in a way that integrates better with type classes.
- Implementing Type Classes demonstrates how some common type classes can be implemented using the new constructs.
- Type Class Derivation introduces constructs to automatically derive type class instances for ADTs.
- Multiversal Equality introduces a special type class to support type safe equality.
- Context Functions provide a way to abstract over context parameters.
- By-Name Context Parameters are an essential tool to define recursive synthesized values without looping.
- Relationship with Scala 2 Implicits discusses the relationship between old-style implicits and new-style givens and how to migrate from one to the other.

Overall, the new design achieves a better separation of term inference from the rest of the language: There is a single way to define givens instead of a multitude of forms all taking an `implicit` modifier. There is a single way to introduce implicit parameters and arguments instead of conflating implicit with normal arguments. There is a separate way to import givens that does not allow them to hide in a sea of normal imports. And there is a single way to define an implicit conversion which is clearly marked as such and does not require special syntax.

This design thus avoids feature interactions and makes the language more consistent

and orthogonal. It will make implicits easier to learn and harder to abuse. It will greatly improve the clarity of the 95% of Scala programs that use implicits. It has thus the potential to fulfil the promise of term inference in a principled way that is also accessible and friendly.

Could we achieve the same goals by tweaking existing implicits? After having tried for a long time, I believe now that this is impossible.

- First, some of the problems are clearly syntactic and require different syntax to solve them.
- Second, there is the problem how to migrate. We cannot change the rules in mid-flight. At some stage of language evolution we need to accommodate both the new and the old rules. With a syntax change, this is easy: Introduce the new syntax with new rules, support the old syntax for a while to facilitate cross compilation, deprecate and phase out the old syntax at some later time. Keeping the same syntax does not offer this path, and in fact does not seem to offer any viable path for evolution
- Third, even if we would somehow succeed with migration, we still have the problem how to teach this. We cannot make existing tutorials go away. Almost all existing tutorials start with implicit conversions, which will go away; they use normal imports, which will go away, and they explain calls to methods with implicit parameters by expanding them to plain applications, which will also go away. This means that we'd have to add modifications and qualifications to all existing literature and courseware, likely causing more confusion with beginners instead of less. By contrast, with a new syntax there is a clear criterion: Any book or courseware that mentions `implicit` is outdated and should be updated.

Table of Contents

- Context Functions
- Implicit Conversions
- By-Name Context Parameters
- Relationship with Scala 2 Implicits

Transla... Given I...