

Simplicity

Foundations and Applications of Implicit Function Types

MARTIN ODERSKY, AGGELOS BIBOUDIS, FENGYUN LIU, OLIVIER BLANVILLAIN,
and HEATHER MILLER, École Polytechnique Fédérale de Lausanne

Understanding a program entails understanding its context; dependencies, configurations and even implementations are all forms of contexts. Modern programming languages and theorem provers offer an array of constructs to define contexts, implicitly. Scala offers *implicit parameters* which are used pervasively, but which cannot be abstracted over.

This paper describes a generalization of implicit parameters to *implicit function types*, a powerful way to abstract over the context in which some piece of code is run. We provide a formalization based on bidirectional type-checking that closely follows the semantics implemented by the Scala compiler.

To demonstrate their range of abstraction capabilities, we present several applications that make use of implicit function types. We show how to encode the builder pattern, tagless interpreters, reader and free monads and we assess the performance of the monadic structures presented.

ACM Reference format:

Martin Odersky, Aggelos Biboudis, Fengyun Liu, Olivier Blanvillain, and Heather Miller. 2017. Simplicity. *PACM Progr. Lang.* 1, 1, Article 1 (June 2017), 29 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A central issue with programming is how to express dependencies. A piece of program *text* can be understood only in some *context* on which it depends. Most imperative programs express context as global state. This technique is non-modular and dangerous due to the pervasiveness of actions affecting that state. The object-oriented programming field has invented several more fine-grained dependency injection mechanisms, either in the language itself (e.g. the *cake pattern* (Odersky and Zenger 2005)) or external to it (e.g., Guice (Vanbrabant 2008), Spring (Johnson 2002), MacWire (Warski 2013)).

Functional programming has a more straightforward answer: Dependencies are simply expressed as parameters. A function that relies on some piece of data needs to be passed that data as a parameter. This is pleasingly simple but sometimes it's too much of a good thing. Programs can quickly become riddled with long parameter lists. A number of programming techniques have been invented to combat that problem. Currying and point-free style, or the reader monad (Jones 1995), are some examples. Nevertheless, it's fair to say that the flexible composition of components with many functional dependencies remains challenging.

A straightforward way of dealing with the problem of having too many parameters is to make some of them *implicit*. Arguments to implicit parameters are synthesized according to the type of the parameter. Implicit parameters were first invented in Haskell (Lewis et al. 2000). They are widely used in Scala, where they model type classes (Oliveira et al. 2010; Wadler and Blott 1989), configurations, capabilities, type constraints (Emir et al. 2006) and many other forms of contextual abstractions. In fact, we found that 94% of the 120 most popular Scala projects hosted on GitHub

make use of implicits in some way, with implicit parameters being the most commonly preferred (full details in Appendix A). These widespread usage patterns indicate that real world applications prefer to pass context *implicitly*. Implicit parameters have also been introduced to Agda (Devriese and Piessens 2011) and to OCaml (White et al. 2015).

Implicit parameters in Scala avoid the tedium of having to pass many parameters to many functions. But they do not eliminate all repetition as they still have to be declared in every function that uses them. For instance, the *Dotty* compiler (Odersky 2017) for Scala contains over 2600 occurrences of the parameter clause (`implicit ctx: Context`). The next version of Scala as implemented by *Dotty* supports a concept that can get rid of this repetition. *Implicit function types* are first-class types representing implicit functions. Just as implicit parameters synthesize function applications to implicit values, implicit function types synthesize implicit lambda abstractions. It turns out that implicit function types are surprisingly powerful in that they can abstract over many different kinds of context dependencies.

Given the widespread use and power of implicits in Scala, it is important to have a precise formalization that explains their semantics. The implicit calculus (Oliveira et al. 2012; Schrijvers et al. 2017) is motivated by examples from both Scala’s implicits and Haskell’s type classes. It comes close to expressing Scala’s implemented semantics, but there are also some differences. In particular, the implicit calculus demands queries of implicit values to be written explicitly, whereas in Scala implicit values are passed automatically as arguments to implicit functions. Implicit function types also appear under the name *instance arguments* in Agda (Devriese and Piessens 2011; Norell et al. 2017), but their formalization and implementation is quite dependent on the rest of Agda’s type elaboration.

In this paper, we explore foundations for implicits as they are found in Scala. We develop SI, a calculus that expresses implicit parameters and implicit function types in a version of System F with implicit type application. The typing rules also provide a mapping from SI to full System F. The rules as given are notably simpler than previous work and correspond closely to the semantics of implicits in Scala. In particular, the calculus keeps Scala’s technique that implicit resolution amounts to choosing an identifier among a finite number of candidates. Once this choice is made, the rest is regular type checking and inference, no separate calculus or algorithm for implicit resolution is needed.

The paper substantiates the following two claims:

- that implicit parameters and implicit function types are powerful and widely applicable constructs for abstracting over context;
- that they admit a simple and elegant formalization.

Organization. First, we introduce a motivating example for implicit function types (Section 2). Next, we introduce a formalization of implicit function types based on System F (Section 3). Then, we navigate, through various different applications, a range of new coding patterns available to programmers (Section 5). And finally, we finish with a discussion of related work (Section 6).

2 OVERVIEW: FROM IMPLICIT PARAMETERS TO IMPLICIT FUNCTION TYPES

Implicit parameters offer a convenient way to write code without the need to pass all arguments explicitly.

On every call to methods with implicit parameters, the compiler looks for an implicit definition in scope to satisfy the call.

So, instead of passing a parameter explicitly:

```
val number: Int = 1
def add(x: Int)(y: Int) = x + y
add(2, number)
```

we can mark a set of parameters as *implicit* (one parameter in this example) and let the compiler retrieve the missing argument for us. In the following example `add` is a method with one implicit parameter and `number` is an implicit definition.

```
implicit val number: Int = 1
def add(x: Int)(implicit y: Int) = x + y
add(2)
```

This discovery process performed by the compiler is called *implicit resolution*. There are two basic groups of rules applied by the resolution algorithm: 1) it looks for implicits in the current scope and 2) in all associated types (e.g., companion objects and outer objects in the presence of nested types). In the previous example, the implicit definition is defined in the current scope and since it is an `Int`, the compiler resolves the method call by passing `number` automatically.

Implicits can be used to implement the type class-design (Wadler and Blott 1989) in Scala. We give an example of implementation of the ordered type class. This example consists of three parts: first, `Ord[T]`, which is a regular trait with a single method, `compare`. Second, the generic function `comp`, which compares two arguments and accepts an implicit argument, providing an *implicit evidence* that these two values can be compared. Third, the implicit definition `intOrd`, which provides an *instance* of the `Ord` trait for integers.

```
trait Ord[T] {
  def compare(a: T, b: T): Boolean
}

def comp[T](x: T, y: T)(implicit ev: Ord[T]): Boolean =
  ev.compare(x, y)

implicit def intOrd: Ord[Int] = new Ord[Int] {
  def compare(a: Int, b: Int): Boolean = a < b
}
comp(1, 2)
```

2.1 Introducing Implicit Function Types

We introduce *implicit function types*; types for implicit function values. These types extend the support for first class functions to implicit arguments. Just like the normal function type syntax desugars `A => B` to `scala.Function1[A, B]`, the implicit function type `implicit A => B` desugars to `scala.ImplicitFunction1[A, B]`:

```
trait ImplicitFunction1[-T0, +R] extends Function1[T0, R] {
  override def apply(implicit x: T0): R
}
```

The `comp` function can thus be rewritten as follows:

```
def comp[T](x: T, y: T): implicit Ord[T] => Boolean =
  implicitly[Ord[T]].compare(x, y)
```

Note that in this definition of `comp` omits naming implicit argument. The compiler will desugar body of this definition into `{ implicit ev: Ord[T] => ... }`.

`implicitly` is defined in the standard library as the implicit identity function:

```
def implicitly[T](implicit x: T) = x
```

Implicit function types can also be used in argument position. For example, we can define an ascending function which injects the `intOrd` implicit in an expression:

```
def ascending[T](expr: implicit Ord[Int] => T): T = expr(intOrd)

ascending {
  comp(1, 2) && comp(3, 4)
}
```

3 FORMALIZATION

Implicits give considerable power to program designers. It's therefore important to have a clear understanding of their elaboration. To this purpose we develop a small calculus SI that can express key elements of high-level Scala programs. The calculus can be thought of as an extension of System F (Girard 1972; Reynolds 1974) with implicit instantiations of both type arguments and implicit parameters. Its semantics is given by a translation to classical System F.

3.1 Syntax

$s, t =$	Term	$R =$	Restricted type
x	variable	X	type variable
y	implicit variable	$T \rightarrow T$	function type
$\lambda x.t$	function		
$t t$	application	$S, T =$	Full type
$?$	implicit query	R	restricted type
let $x : T = t$ in t	explicit let	$T ? \rightarrow T$	implicit function type
let $? : T = t$ in t	implicit let	$\forall X.T$	polymorphic function type

Fig. 1. SI Syntax

The syntax of the calculus is presented in Figure 1. At the term level, we have both explicit variables (x) and implicit variables (y). This separation saves us the effort to maintain two different environments in the typing rules. Consequently, only implicit variables are available for implicit resolution. We assume that programmers only use explicit variables in the source code. Implicit variables are used internally in type checking and semantic translation.

One peculiarity of this syntax is that lambda abstractions are written curry-style, without a type for the abstracted variable. Types are introduced separately in a let-expression. There are two variants of the let-binding: explicit and implicit. The latter introduce implicit values.

Another peculiarity is that there are no syntax constructs for implicit functions, nor for passing type or implicit value arguments. All these syntactic elements are to be inferred. The syntax is hence quite familiar to a Haskell programmer (say), in that types are given separately for let-bound variables, closures do not carry types themselves and all type arguments are inferred.

In the calculus, implicit parameters get resolved and applied automatically. The query operator (denoted as $?$) is used to access implicit parameters of implicit functions (which will be synthesized during the semantic translation). Programmers can also use the query operator to trigger implicit resolution explicitly – this corresponds then to the use of `implicitly` in Scala (Aside: People familiar with Scala might raise the issue that `implicitly` usually comes with a type argument such as in `implicitly[Ord[Int]]` whereas $?$ is written without one. But in fact ‘implicitly’ is a normal polymorphic function, so its type argument can be inferred just as for any other function. So, `implicitly`, written alone, corresponds exactly to our use of $?$.)

For types, in addition to the normal function type, we have implicit function types ($T \text{ ?} \rightarrow T$). The types are divided into restricted types and non-restricted types. Restricted types consist of normal function types and type variables. Non-restricted types represent polymorphic and implicit functions; they come with elimination rules that are not syntax directed.

3.2 Type System

The type system of SI is presented in its entirety in Figure 2. SI is based on *bidirectional type checking* (Pierce and Turner 2000). Bidirectional type-checking is a technique that incorporates both a type inference and a type checking flow for typing terms. In the traditional formulation, two directions exist: a term t can be inferred to have a type T on the one direction and a term t can be checked to have a type T on the other. Bidirectionality has been used in a wide range of scenarios from implementations of systems for dependent types (Coquand 1996; Xi and Pfenning 1999) to local type inference (Pierce and Turner 2000) and higher-rank types (Peyton Jones et al. 2007).

In SI the judgement form $\Gamma \vdash t \triangleright T$ means that the term t gets a *synthesized* type T under the context Γ , while $\Gamma \vdash t \triangleleft T$ means that the term t is *checked* to be compatible with the expected type T under the context Γ .

Rule (VAR) is standard. Its implicit analogue (QUERY) can be thought as implicit resolution for the query $?$. The variable y in the precondition of (QUERY) is chosen arbitrarily – any implicit variable in the environment could qualify. We discuss in Section 3.6 how to address this source of non-determinism.

The typing rule ($\rightarrow I$) is standard for a curry-style system. The rule is a checking rule, which means that the type annotation for the lambda parameter can be inferred from the context. The typing rule ($\rightarrow E$) is also standard. Note that in the typing of the argument t_2 , we use the checking judgement, as the type S is already known from the type $S \rightarrow T$ of the function t_1 . The typing rule ($\text{?} \rightarrow I$) type checks an expression by assuming its expansion to an implicit function, but only if the expected type of the expression is of an implicit function type. This last point is the reason why we do not have a syntax for implicit functions: if we did, this rule would have been applied endlessly! While it’s a simple check in the compiler to stop applying this rule if the term is already an implicit function, avoiding to specify such a rule makes the calculus more clear.

The typing rule ($\text{?} \rightarrow E$) is where automatic resolution of implicits happens. This rule says that if a term t is of the implicit function type $S \text{ ?} \rightarrow T$ and implicit resolution can synthesize a term of the type S , then we can type the term with the type T . The resolution of the implicit parameter could

$\frac{x : T \in \Gamma}{\Gamma \vdash x : \triangleright T} \quad (\text{VAR})$	$\frac{y : T \in \Gamma}{\Gamma \vdash ? : \triangleright T} \quad (\text{QUERY})$
$\frac{\Gamma, x : S \vdash t \triangleleft : T}{\Gamma \vdash \lambda x. t \triangleleft : S \rightarrow T} \quad (\rightarrow \text{I})$	$\frac{\Gamma \vdash t_1 : \triangleright S \rightarrow T \quad \Gamma \vdash t_2 \triangleleft : S}{\Gamma \vdash t_1 t_2 : \triangleright T} \quad (\rightarrow \text{E})$
$\frac{\Gamma, y : S \vdash t \triangleleft : T}{\Gamma \vdash t \triangleleft : S ? \rightarrow T} \quad (? \rightarrow \text{I})$	$\frac{\Gamma \vdash t : \triangleright S ? \rightarrow T \quad \Gamma \vdash ? \triangleleft : S}{\Gamma \vdash t : \triangleright T} \quad (? \rightarrow \text{E})$
$\frac{\Gamma, X \vdash t \triangleleft : T}{\Gamma \vdash t \triangleleft : \forall X. T} \quad (\forall \text{I})$	$\frac{\Gamma \vdash t : \triangleright \forall X. T}{\Gamma \vdash t : \triangleright [X := S]T} \quad (\forall \text{E})$
$\frac{\Gamma \vdash t \triangleleft : T \quad \Gamma, x : T \vdash s : \triangleright R}{\Gamma \vdash \mathbf{let} \ x : T = t \ \mathbf{in} \ s : \triangleright R} \quad (\text{LET-EX})$	$\frac{\Gamma \vdash t \triangleleft : T \quad \Gamma, y : T \vdash s : \triangleright R}{\Gamma \vdash \mathbf{let} \ ? : T = t \ \mathbf{in} \ s : \triangleright R} \quad (\text{LET-IM})$
$\frac{\Gamma \vdash t : \triangleright R}{\Gamma \vdash t \triangleleft : R} \quad (\text{STITCH})$	

Fig. 2. SI Typing Rules

result in a term of implicit function type on which this rule is applied again. This is how recursive resolution works in the system.

The typing rules $(\forall \text{I})$ and $(\forall \text{E})$ introduce and eliminate polymorphic types. Neither of those has a term form associated with it, so introduction and elimination of polymorphic types is again implicit. Being checking rules, $(\forall \text{I})$ instances are inserted deterministically depending on the expected type. For $(\forall \text{E})$, we rely on type inference to determine the argument type S .

The typing rule (LET-EX) type checks explicit let-bindings, while the rule (LET-IM) type checks implicit let-bindings. Let-bindings are the only place where programmers give type annotations.

The rule (STITCH) says that in order to check that a term t has type R , it suffices to show that the synthesized type for t is R . This is the important switch that lets us change from type checking to type synthesis. We allow only restricted types to go through this rule, so that all implicit function types will go through the rule $(? \rightarrow \text{I})$, and universal types will go through the rule $(\forall \text{E})$ before the rule is applied. Conversely, the introduction rules $(? \rightarrow \text{I})$ and $(\forall \text{I})$ have to be applied in checking mode to the conclusion of a (STITCH) .

Notice that the two rules that handle implicit function types have different directions. The rule $(? \rightarrow \text{E})$ is a synthesis rule. It essentially expresses that implicit function types are eliminated eagerly, as soon as they arise. On the other hand, the rule $(? \rightarrow \text{I})$ is a checking rule. It says that implicit function types are introduced only if the expected type specifies it. Together with the rule (STITCH) , these rules determine the following strategy in the compiler for type checking a term t :

- (1) If the expected type is an implicit function type $T ? \rightarrow T'$, create an implicit closure by entering in the environment an anonymous implicit value and proceed type checking t with T' as expected type.

- (2) If the expected type is a restricted type R , type check t . If that succeeds with type T , post-process T in step (3).
- (3) If the type of t is an implicit function type $T \rightarrow T'$, perform an implicit search for T . If a unique term t' is found that matches T , continue by type checking $t t'$.

3.3 Type Checking Examples

To illustrate how the type system works, we walk through two code examples. Both are in Scala syntax for readability. The first example is as follows:

```
val f: implicit Char => Boolean = ...
implicit val n: Int = 3
implicit val g: implicit Int => Char = ...

f : Boolean
```

In the code above, interpret the `implicit S => T` according to the rules: $S \rightarrow T$. The implicit variable definition `implicit val n: T = ...` is equivalent to `let ? : T = ...`, and `f : Boolean` is equivalent to `let x : Boolean = f in x`.

$$\frac{\frac{f \in \Gamma}{\Gamma \vdash f :> \text{Char} \rightarrow \text{Boolean}} \text{(VAR)} \quad \frac{\frac{g \in \Gamma}{\Gamma \vdash ? :> \text{Int} \rightarrow \text{Char}} \text{(QUERY)} \quad \frac{\frac{n \in \Gamma}{\Gamma \vdash ? :> \text{Int}} \text{(QUERY)} \quad \frac{\Gamma \vdash ? \Leftarrow: \text{Int}}{\Gamma \vdash ? \Leftarrow: \text{Int}} \text{(STITCH)}}{\Gamma \vdash ? \Leftarrow: \text{Char}} \text{(? } \rightarrow \text{ E)}}{\Gamma \vdash ? :> \text{Char}} \text{(STITCH)} \quad \frac{\Gamma \vdash ? :> \text{Char}}{\Gamma \vdash ? \Leftarrow: \text{Char}} \text{(STITCH)} \quad \frac{\Gamma \vdash f :> \text{Boolean}}{f, n, g \vdash f \Leftarrow: \text{Boolean}} \text{(STITCH)}$$

Fig. 3. Type Derivation for `f : Boolean`

The type derivation tree for type checking `f` with the expected type `Boolean` is given in Figure 3. For readability, we omit types of variables in the environment and abstract them by Γ when its meaning is clear from the context. As we observe, automatic implicit resolution happens with the rule $(? \rightarrow E)$. The rule (Query) is used twice to pick the right implicits `g` and `n` from the environment.

Here's is another example which covers the rule $(? \rightarrow I)$:

```
val h: implicit (implicit Int => Char) => Boolean = ...
implicit val n: Int = 3
implicit val g: implicit Int => Char = ...

h: Boolean
```

The type derivation for type checking `h` with the expected type `Boolean` is given in Figure 4. An essential difference compared to the previous example is that in the resolution of the implicit parameter for `h` (with the type `Int → Char`), we cannot apply the rule (STITCH) as it only works for restricted types, not implicit function types. The only possible choice here is to use the rule $(? \rightarrow I)$, which does type checking by assuming expansion of expressions into implicit functions. The rule (Query) is also used twice to pick the right implicits `g` and `y` from the environment. Note that the rule gives priority to the inner-most implicit in the environment, thus `y` is chosen instead of `n`.

$$\frac{\frac{\frac{h \in \Gamma}{\Gamma \vdash h : \triangleright (\text{Int } ? \rightarrow \text{Char}) ? \rightarrow \text{Boolean}} \text{(VAR)}}{\Gamma \vdash h : \triangleright \text{Boolean}} \text{(STITCH)} \quad \frac{\frac{g \in \Gamma'}{\Gamma' \vdash ? : \triangleright \text{Int } ? \rightarrow \text{Char}} \text{(QUERY)} \quad \frac{\frac{y \in \Gamma'}{\Gamma' \vdash ? : \triangleright \text{Int}} \text{(QUERY)} \quad \frac{\Gamma' \vdash ? \triangleleft : \text{Int}}{\Gamma' \vdash ? \triangleleft : \text{Int}} \text{(STITCH)}}{\Gamma, y : \text{Int } \vdash ? \triangleleft : \text{Char}} \text{(? \rightarrow I)}}{\Gamma \vdash ? \triangleleft : \text{Int } ? \rightarrow \text{Char}} \text{(? \rightarrow E)}}{\Gamma \vdash h : \triangleright \text{Boolean}} \text{(STITCH)}$$

Fig. 4. Type Derivation for $h : \text{Boolean}$

3.4 Translation to System F

We introduce a type-preserving translation from SI to System F. The syntax, typing rules and semantics of System F are standard, so we omit them here.

The translation of types is given below:

$$\begin{aligned}
(S ? \rightarrow T)^* &= S^* \rightarrow T^* \\
(S \rightarrow T)^* &= S^* \rightarrow T^* \\
(\forall X.T)^* &= \forall X.T^* \\
T^* &= T, \text{ otherwise}
\end{aligned}$$

We use the following judgment form to mean that a well-typed term t in SI will be translated into a term t' in System F:

$$\Gamma \vdash t : T \rightsquigarrow t'$$

The translation rules are presented in Figure 5. Note that we do not translate implicit variables and assume that the target language treats implicit variables and explicit variables the same way.

THEOREM 3.1 (TYPE-PRESERVING TRANSLATION). *Let t be a SI term of type T , and t' be an System F term. If $\emptyset \vdash t : T \rightsquigarrow t'$, then $\emptyset \vdash t' : T^*$.*

PROOF. Straight-forward induction on the typing rules. □

3.5 Examples

To demonstrate the expressiveness of the calculus, we present two examples that apart from using syntax equivalent to the notation of the calculus they are also verified by the Dotty compiler. The implicit function type $S ? \rightarrow T$ is written `implicit S => T` in Scala. Instead of the query operator $?$ we write `implicitly` and instead of the `let` constructs of the calculus we use `defs`. Since Scala does not allow `defs` to be anonymous, we use anonymized names such as `__1` and `__2`, for implicit definitions instead. These names are used nowhere else in the program, so their precise spelling is not important.

Example: Ordering. This example defines a typeclass for orderings with instances on `Int` and `List`. It models *higher-order implicits*, i.e. implicits that depend on other implicits. In the example, the implicit for the type `Ord[List[T]]` depends on an implicit instance of `Ord[T]`.

```

object Orderings {
  trait Ord[T] { def less: T => T => Boolean }

  implicit def __1: Ord[Int] = new Ord[Int] {
    def less: Int => Int => Boolean = x => y => x < y
  }

```


$\frac{x : T \in \Gamma}{\Gamma \vdash x : \triangleright T \rightsquigarrow x} \quad (\text{VAR-Ex})$	$\frac{y : T \in \Gamma}{\Gamma \vdash ? : \triangleright T \rightsquigarrow y} \quad (\text{QUERY})$
$\frac{\Gamma, x : S \vdash t \triangleleft : T \rightsquigarrow u}{\Gamma \vdash \lambda x. t \triangleleft : S \rightarrow T \rightsquigarrow \lambda x : S^*. u} \quad (\rightarrow I)$	$\frac{\Gamma \vdash t_1 : \triangleright S \rightarrow T \rightsquigarrow u \quad \Gamma \vdash t_2 \triangleleft : S \rightsquigarrow u'}{\Gamma \vdash t_1 t_2 : \triangleright T \rightsquigarrow u u'} \quad (\rightarrow E)$
$\frac{y \text{ fresh} \quad \Gamma, y : S \vdash t \triangleleft : T \rightsquigarrow u}{\Gamma \vdash t \triangleleft : S ? \rightarrow T \rightsquigarrow \lambda y : S^*. u} \quad (? \rightarrow I)$	$\frac{\Gamma \vdash t : \triangleright S ? \rightarrow T \rightsquigarrow u \quad \Gamma \vdash ? \triangleleft : S \rightsquigarrow u'}{\Gamma \vdash t : \triangleright T \rightsquigarrow u u'} \quad (? \rightarrow E)$
$\frac{\Gamma, X \vdash t \triangleleft : T \rightsquigarrow u}{\Gamma \vdash t \triangleleft : \forall X. T \rightsquigarrow \Delta X. u} \quad (\forall I)$	$\frac{\Gamma \vdash t : \triangleright \forall X. T \rightsquigarrow u}{\Gamma \vdash t : \triangleright [X := S] T \rightsquigarrow u [S^*]} \quad (\forall E)$
$\frac{\Gamma \vdash t \triangleleft : T \rightsquigarrow u \quad \Gamma, x : T \vdash s : \triangleright R \rightsquigarrow u'}{\Gamma \vdash \mathbf{let} x : T = t \mathbf{in} s : \triangleright R \rightsquigarrow (\lambda x : T^*. u') u} \quad (\text{LET-Ex})$	$\frac{\Gamma \vdash t \triangleleft : T \rightsquigarrow u \quad y \text{ fresh} \quad \Gamma, y : T \vdash s : \triangleright R \rightsquigarrow u'}{\Gamma \vdash \mathbf{let} ? : T = t \mathbf{in} s : \triangleright R \rightsquigarrow (\lambda y : T^*. u') u} \quad (\text{LET-Im})$
$\frac{\Gamma \vdash t : \triangleright R \rightsquigarrow u}{\Gamma \vdash t \triangleleft : R \rightsquigarrow u} \quad (\text{STITCH})$	

Fig. 5. Type-directed translation from SI to System F

```

}

implicit def __2[T]: implicit Ord[T] => Ord[List[T]] = new Ord[List[T]] {
  def less: List[T] => List[T] => Boolean =
    xs => ys =>
      if ys.isEmpty then false
      else if xs.isEmpty then true
      else if xs.head == ys.head then less(xs.tail)(ys.tail)
      else isLess(xs.head)(ys.head)
}

def isLess[T]: T => T => implicit Ord[T] => Boolean =
  x => y => implicitly[Ord[T]].less(x)(y)
}
import Orderings._

isLess(Nil)(List(1, 2, 3))
isLess(List(List(1)))(List(List(1)))

```

Example: Propagation of Session Context. This example models a *conference management system*. In this system, users are not allowed to see the scores or rankings of their own papers. Thus, all operations, like getting the score of a paper or rankings of papers, depends on the identity of the current user. Passing current user (or session) explicitly as parameters of the operations would be very verbose. Implicit function types help here. By minor changes to the type signatures of methods, the compiler will propagate the session context automatically.

```

case class Person(name: String)
case class Paper(title: String, authors: List[Person], body: String)

class ConfManagement(papers: List[Paper], realScore: Map[Paper, Int]) {
  type Session[T] = implicit Person => T
  def currentUser: Session[Person] = implicitly
  def hasConflict(p: Person, ps: List[Person]) = ps contains p

  def score: Paper => Session[Int] = paper =>
    if hasConflict(currentUser, paper.authors) then -1
    else realScore(paper)

  def viewRankings: Session[List[Paper]] =
    papers.sortBy(score(_))
}

```

The following code demonstrates a simple setup where the assumed logged-in user is Bob, who has also submitted a paper in the system. By running this program we observe that Bob is unable to see the score of his paper (-1 instead of 4 which is the actual value).

```

val bob = Person("Bob")
val eve = Person("Eve")
val p1 = Paper("Bob's paper", List(bob), "...")
val p2 = Paper("Eve's paper", List(eve), "...")
val cm = new ConfManagement(List(p1, p2), Map(p1 -> 4, p2 -> 3))

implicit def __1: Person = bob

cm.score(p1)    // -1
cm.score(p2)    // 3

```

3.6 Disambiguation

The calculus presented in Figure 2 is ambiguous in two aspects. First, rule (\forall E) does not specify the type argument syntactically, and different type arguments can lead to different derivations. This is similar to the situation in the Hindley-Milner (Damas and Milner 1982) system where we also assume that the type argument is inferred by an algorithm. The second source of ambiguity is specific to implicits: rule (QUERY) does not specify how to pick an implicit variable from the environment. By the rule itself, any implicit variable in the environment qualifies, thus there are several possible derivations. The implicit ambiguity is problematic because it affects semantics. Choices of implicit values can affect the outcome of a program, which means ambiguous implicit resolutions causes a loss of *coherence*.

We can regain coherence through a disambiguation scheme that always selects in a deterministic way a specific implicit value among several possible candidates. A simple way to do so is to look

at nesting: always select the innermost implicit value that matches a required type R . The Scala compiler follows this strategy but augments it with other criteria. Section 4 gives more details.

In the rest of this section we formalize this strategy using a definition of *well-scopedness* and show that it leads to unique type derivations if we restrict ourselves to the monomorphic fragment of SI without \forall types (i.e. if we exclude the other source of ambiguity). In the rest of this section, we will assume a calculus the subset of SI without rules $(\forall E)$ and $(\forall I)$.

The proofs are significant because they show that argument type selection and implicit variable selection are the only sources of ambiguity in SI. In particular, no ambiguities can arise from different possible arrangements of implicit introduction and elimination rules. This property is enforced by the directional nature of the typing rules and the particular role of (STITCH) , which only works on restricted types. It makes SI different from the treatment of the unrestricted implicit calculus (Oliveira et al. 2012) and also from the treatment of implicit parameters in Haskell, where implicit introduction and elimination are less well controlled and therefore can give rise to surprising behavior (GHC 2015).

Definition 3.2. Extend a measure of type size $\text{size}(T)$ to judgements as follows:

$$\text{size}(\Gamma \vdash t \triangleleft: T) = \text{size}(T) \quad (1)$$

$$\text{size}(\Gamma \vdash t \triangleright: T) = -\text{size}(T) \quad (2)$$

LEMMA 3.3. *Let J be a judgement over the term t . Then for any non-syntax-directed inference rule that uses J as its leftmost precondition and that has t again as the term part of its conclusion: $\text{size}(J) < \text{size}(J')$.*

PROOF. By inspection of the typing rules □

Definition 3.4. The *resolved implicit variable* $\text{res}(\mathcal{D})$ in a derivation tree \mathcal{D} ending in a judgement $\Gamma \vdash ? \triangleleft: T$ or $\Gamma \vdash ? \triangleright: T$ is defined as follows.

- (1) If \mathcal{D} consists of an application of a (QUERY) instance

$$\frac{\Gamma \vdash y \triangleright: T}{\Gamma \vdash ? \triangleright: T}$$

, the variable y .

- (2) If \mathcal{D} ends with an application of a $(? \rightarrow E)$ instance

$$\frac{\Gamma \vdash ? \triangleright: S \rightarrow T \quad \Gamma \vdash ? \triangleleft: S}{\Gamma \vdash ? \triangleright: T}$$

, the resolved implicit variable of the derivation tree of $\Gamma \vdash ? \triangleright: S \rightarrow T$.

- (3) If \mathcal{D} ends in an application of a (STITCH) instance

$$\frac{\Gamma \vdash ? \triangleright: R}{\Gamma \vdash ? \triangleleft: R}$$

, the resolved implicit variable of the derivation tree of $\Gamma \vdash ? \triangleright: R$.

- (4) If \mathcal{D} consists of an application of a $(? \rightarrow I)$ instance

$$\frac{\Gamma, y : S \vdash ? \triangleleft: T}{\Gamma \vdash ? \triangleleft: S \rightarrow T}$$

, the resolved implicit variable of the derivation tree of $\Gamma, y : S \vdash ? \triangleleft : T$.

Definition 3.5. A typing derivation \mathcal{D} is *well-scoped*, if for every subgoal $\Gamma \vdash ? \triangleleft : T$ or $\Gamma \vdash ? \triangleright : T$ in \mathcal{D} , and every other derivation \mathcal{D}' of the same subgoal, $\text{res}(\mathcal{D}) = \text{res}(\mathcal{D}')$ or $\text{res}(\mathcal{D}')$ is defined in Γ to the left of $\text{res}(\mathcal{D})$.

Well-scoped typing derivations can be produced by a semi-algorithm that, when faced with a query, always picks the rightmost eligible implicit (which corresponds to the innermost implicit definition as seen from the point where the query is made (it's not an algorithm because implicit search might diverge)). We can show that the following holds for the monomorphic system without polymorphic function types:

PROPOSITION 3.6. *Given Γ and t , there is at most one well-scoped typing derivation that ends in $\Gamma \vdash t \triangleright : R$, for some restricted type R .*

PROPOSITION 3.7. *Given Γ, t and T , there is at most one well-scoped typing derivation that ends in $\Gamma \vdash t \triangleleft : T$.*

PROOF. We prove the following three conditions which imply Proposition 3.6 and proposition 3.7: Given Γ, t , and a well-scoped derivation \mathcal{D} ,

- (1) If \mathcal{D} ends in $\Gamma \vdash t \triangleright : R$, then it is the only well-scoped derivation with this property.
- (2) If \mathcal{D} ends in $\Gamma \vdash t \triangleright : T$, and \mathcal{D} does not end in a rule ($? \rightarrow E$), then it is the only well-scoped derivation with these two properties.
- (3) If \mathcal{D} ends in $\Gamma \vdash t \triangleleft : T$ then it is the only well-scoped derivation with this property.

Condition (1) implies Proposition 3.6 and condition (3) implies Proposition 3.7. We prove all three conditions together by an induction on the derivation tree \mathcal{D} .

If the derivation consists of a single application of (VAR), then (1) and (2) are immediate consequences, and (3) holds vacuously because its precondition does not apply.

Assume now the derivation consists of a single application of (QUERY).

$$\frac{\Gamma \vdash y \triangleright : T}{\Gamma \vdash ? \triangleright : T}$$

We only need to prove (1) and (2), since (3) holds vacuously. Consider another derivation \mathcal{D}' which ends in $\Gamma \vdash ? \triangleright : T$. Since \mathcal{D} and \mathcal{D}' are both well-scoped, they must both resolve to the same implicit variable, y . Therefore, \mathcal{D}' also starts with the same application of (QUERY). By lemma 3.3, \mathcal{D}' cannot have any other inference rules that take the (QUERY) rule as a precondition and that end in the same conclusion as (QUERY).

Assume now the derivation tree \mathcal{D} ends in a synthesis rule different from (VAR) and (QUERY), with $\Gamma \vdash t : T$ as conclusion. Let \mathcal{D}_0 be the smallest sub-derivation of \mathcal{D} that can be extended to full \mathcal{D} by applying zero or more instances of rule ($? \rightarrow E$), with \mathcal{D}_0 and its successors being in each case the left-most precondition of what follows. Assume \mathcal{D}' ends in $\Gamma \vdash t : T'$. We distinguish according to the form of t . By inspection of the typing rules there are only three cases, one where t is an application $t_1 t_2$, the other two there t is an explicit let **let** $x : T = t_1$ **in** t_2 or implicit let **let** $? : T = t_1$ **in** t_2 . Assume the first case. Then the only applicable rule is ($\rightarrow E$).

$$\frac{\Gamma \vdash t_1 \triangleright : S \rightarrow T' \quad \Gamma \vdash t_2 \triangleleft : S}{\Gamma \vdash t_1 t_2 \triangleright : T'}$$

By the inductive hypothesis the typing derivations of $\Gamma \vdash t_1 \multimap S \rightarrow T'$ and $\Gamma \vdash t_2 \triangleleft: S$ are both unique. Hence, any other derivation \mathcal{D}'_0 of $\Gamma \vdash t_1 t_2 \multimap T'$ must contain \mathcal{D}_0 as a sub-derivation. The same holds for \mathcal{D}_0 , which together with Lemma 3.3 proves (2). The other two cases are analogous.

To show (1), consider the type T of the conclusion of \mathcal{D}_0 . Its general form is $S_1 \multimap \dots \multimap S_n \multimap R$, for some types S_1, \dots, S_n and restricted type R . We show unicity of derivations by induction on n . If $n = 0$, then by Lemma 3.3 $\mathcal{D} = \mathcal{D}_0$ and therefore (2) implies (1). If $n > 0$ then the only rule applicable is $(\multimap E)$.

$$\frac{\Gamma \vdash t \multimap S_1 \multimap (S_2 \multimap \dots \multimap S_n \multimap R) \quad \Gamma \vdash ? \triangleleft: S_1}{\Gamma \vdash t \multimap S_2 \multimap \dots \multimap S_n \multimap R}$$

By the induction hypothesis, the derivation of $\Gamma \vdash t \multimap S_1 \multimap (S_2 \multimap \dots \multimap S_n \multimap R)$ is unique. Therefore, the same holds for \mathcal{D} .

Finally, assume that derivation tree \mathcal{D} ends in a checking judgement $\Gamma \vdash t \triangleleft: T$. If T is of the form $S \multimap T'$ the only applicable rule that ends in this judgement is $(\multimap I)$. By induction, the derivation for the premise of this rule is unique, which implies with Lemma 3.3 that \mathcal{D} is unique. If T is a restricted type R , the applicable rules are $(\rightarrow I)$ and (STITCH) . We further distinguish according to the term t . If t is of the form $\lambda x.t'$, the last rule must be $(\rightarrow I)$, since (STITCH) has a synthesis judgement as its premise and there is no rule that can typecheck a lambda abstraction in synthesis mode. By the inductive hypothesis, the premise of the $(\rightarrow I)$ application has a unique derivation, and therefore \mathcal{D} is also unique. If t is not a lambda abstraction, the only applicable rule is (STITCH) . By the inductive hypothesis the premise of (STITCH) has a unique derivation, and therefore \mathcal{D} is also unique. This concludes the proof. \square

The propositions do not hold anymore once we add polymorphic function types because type instantiation in the rule $(\forall E)$ is also non-deterministic, and interacts in interesting ways with implicit search. Dealing with this will require a formalization of local type inference and how it is influenced by implicit search.

3.7 Type Checking

We now present a semi-algorithm for type checking programs in the monomorphic fragment of SI. It is given by four mutually recursive functions:

$$\begin{array}{lll} \text{synth} & \llbracket \Gamma \vdash t \rrbracket & : (T, t) \mid \text{error} \\ \text{check} & \llbracket \Gamma \vdash t \triangleleft: T \rrbracket & : t' \mid \text{error} \\ \text{elim} & \llbracket \Gamma \vdash t \multimap T \rrbracket & : (R, t) \mid \text{error} \\ \text{query} & \llbracket \Gamma \vdash t \rrbracket \Gamma' & : t' \mid \text{error} \end{array}$$

The definitions of these functions are given in Figure 6. *synth* elaborates synthesis rules in Figure 5 and *check* elaborates checking rules. However, there is no *synth* rule that corresponds to (QUERY) . Instead, queries $?$ that fill in function arguments are handled by a combination of *query* and *elim*. *query* tries an implicit candidate in the environment, going left to right, and *elim* instantiates implicit function types.

PROPOSITION 3.8. (*Soundness*)

- (1) If $\text{synth} \llbracket \Gamma \vdash t \rrbracket = (T, t')$ then $\Gamma \vdash t \multimap T \rightsquigarrow t'$.
- (2) If $\text{check} \llbracket \Gamma \vdash t \triangleleft: T \rrbracket = t'$ then $\Gamma \vdash t \triangleleft: T \rightsquigarrow t'$

The proof of Proposition 3.8 is by induction on the number of steps taken by the algorithm. One can show that each step corresponds to a typing rule in Figure 5.

$synth[\Gamma \vdash x]$	$=$	$elim[\Gamma \vdash x \text{ :> } \Gamma(x)]$	
$synth[\Gamma \vdash t_1 t_2]$	$=$	$(T, t'_1 t'_2)$	if $(S \rightarrow T, t'_1) = synth[\Gamma \vdash t_1]$ $t'_2 = check[\Gamma \vdash t_2 \triangleleft: S]$
$synth[\Gamma \vdash \mathbf{let } x : S = s \mathbf{ in } t]$	$=$	$(T, (\lambda x : S^*. t') s')$	otherwise if $s' = check[\Gamma \vdash s \triangleleft: S]$ $(T, t') = synth[\Gamma, x : S \vdash t]$
$synth[\Gamma \vdash \mathbf{let } ? : S = s \mathbf{ in } t]$	$=$	$(T, (\lambda y : S^*. t') s')$	otherwise if $s' = check[\Gamma \vdash s \triangleleft: S]$ $(T, t') = synth[\Gamma, y : S \vdash t]$ y fresh
		$error$	otherwise
$elim[\Gamma \vdash t \text{ :> } R]$	$=$	(R, t)	
$elim[\Gamma \vdash t \text{ :> } S \text{ ?}\rightarrow T]$	$=$	(R, t')	if $s' = query[\Gamma \vdash S] \Gamma$ $(R, t') = elim[\Gamma \vdash t s' \text{ :> } T]$
		$error$	otherwise
$query[\Gamma \vdash T] ()$	$=$	$error$	
$query[\Gamma \vdash T] (\Gamma', x : S)$	$=$	$query[\Gamma \vdash T] \Gamma'$	
$query[\Gamma \vdash T] (\Gamma', y : S)$	$=$	t	if $(T, t) = synth[\Gamma \vdash y]$
		$query[\Gamma \vdash T] \Gamma'$	otherwise
$check[\Gamma \vdash \lambda x. t \triangleleft: S \rightarrow T]$	$=$	$\lambda x : S^*. t'$	if $t' = check[\Gamma, x : S \vdash t \triangleleft: T]$
		$error$	otherwise
$check[\Gamma \vdash t \triangleleft: S \text{ ?}\rightarrow T]$	$=$	$\lambda y : S^*. t'$	if $t' = check[\Gamma, y : S \vdash t \triangleleft: T]$ y fresh
		$error$	otherwise
$check[\Gamma \vdash t \triangleleft: R]$	$=$	t'	if $(R, t') = synth[\Gamma \vdash t]$
		$error$	otherwise

Fig. 6. Type checking algorithm for SI

The definitions given in Figure 6 constitute a semi-algorithm, which might diverge on some programs (we explain in Section 4 how we deal with this in practice). Hence, we cannot establish completeness. Nevertheless, we believe the following holds:

CONJECTURE 3.9. (*Semi-Completeness*)

- (1) *If $\Gamma \vdash t \text{ :> } T \rightsquigarrow t'$ by a well-scoped derivation then the application $synth[\Gamma \vdash t]$ yields (T, t') or it diverges.*
- (2) *If $\Gamma \vdash t \triangleleft: T \rightsquigarrow t'$ by a well-scoped derivation then the application $check[\Gamma \vdash t \triangleleft: T]$ yields t' or it diverges.*

3.8 Comparison with the Implicit Calculus

Compared to the implicit calculus (Oliveira et al. 2012; Schrijvers et al. 2017) there are two essential differences.

First, SI models the automatic application of implicit functions whereas the implicit calculus requires an explicit query operator like $?T$ to trigger implicit resolution for the type T .

Second, SI unifies type checking and implicit resolution whereas the implicit calculus uses two different sets of rules for these concerns. This makes SI significantly smaller than the implicit calculus. We show in Section 4 that the Scala compiler also follows SI’s strategy of using normal type checking of an implicit candidate for implicit search.

The two points are connected. It’s *because* SI can model automatic function application (both for types and for implicit values) that it can do without a separate set of rules for implicit resolution. To illustrate this, consider the typing derivation that finds an ordering over `List[Int]`, given the code from Section 3.5.

$$\frac{\frac{\frac{_2 : (\forall T. \text{Ord}[T] \rightarrow \text{Ord}[\text{List}[T]]) \in \Gamma}{\Gamma \vdash ? \triangleright \forall T. \text{Ord}[T] \rightarrow \text{Ord}[\text{List}[T]] \rightsquigarrow _2} \text{(QUERY)}}{\Gamma \vdash ? \triangleright \text{Ord}[Int] \rightarrow \text{Ord}[\text{List}[Int]] \rightsquigarrow _2[\text{Int}]} \text{(VE)}}{\Gamma \vdash ? \triangleright \text{Ord}[\text{List}[Int]] \rightsquigarrow _2[\text{Int}] \text{(}_1)} \text{(STITCH)}}{\frac{\frac{_1 : \text{Ord}[Int] \in \Gamma \rightsquigarrow _1}{\Gamma \vdash ? \triangleright \text{Ord}[Int] \rightsquigarrow _1} \text{(QUERY)}}{\Gamma \vdash ? \triangleleft : \text{Ord}[Int] \rightsquigarrow _1} \text{(STITCH)}}{\Gamma \vdash ? \triangleleft : \text{Ord}[\text{List}[Int]] \rightsquigarrow _2[\text{Int}] \text{(}_1)} \text{(}\rightarrow \text{E)}} \text{(STITCH)}$$

In this derivation, each application of (QUERY) selects a simple variable ($_1$ and $_2$), but the end result is the more complex term $_2[\text{Int}](_1)$. If the type system demanded an explicit placeholder for implicit function arguments, a similar construction would not have been feasible, since we would have to “guess” a priori where implicit arguments to implicit functions are needed. Hence, we’d need a separate mechanism of implicit resolution.

On the other hand, a design with separate type checking and resolution judgements has the advantage that type instantiation can be made explicit. In the `Ord` instantiation, we “pulled a rabbit out of our hat” in the application of (VE) where the instance type was guessed to be `Int`. So to arrive at a type checking algorithm we have to complement the system as given with rules that specify how to infer type arguments. Scala uses a variant of colored local type inference (Odersky et al. 2001) for this.

Both SI and the basic implicit calculus are ambiguous. They both under-specify what implicit will be chosen in the face of ambiguities. The Cochis calculus (Schrijvers et al. 2017) elaborates the implicit resolution rules to ensure determinacy, but the details are involved and it’s not clear to what degree they correspond to the disambiguation scheme used in Scala and reported in Section 4. The well-scopedness condition of Section 3.6 also ensures unique type derivations, but only for the monomorphic fragment of SI. To extend it to full SI we’d have to complement the system with rules for the inference of type arguments. This is beyond the scope of the current paper, though.

4 IMPLEMENTATION

Implicit function types have been implemented in *dotc*, the reference compiler for future versions of the Scala language. This section explains the relationship between this implementation and the formalization presented in Section 3.

Scala uses colored local type inference (Odersky et al. 2001), which is a refinement of the bidirectional type checking rules in Figure 2. In essence, every typing rule is a hybrid between checking and synthesis: It checks the term against a given *outline type* and synthesizes the final type of the term. Outline types can have holes in them. Synthesized types follow the shape of the corresponding outline types, while at the same time filling in the holes. This system coincides in the following algorithmic aspects with Figure 2 and Figure 5

- If the expected type of a term is an implicit function type, an implicit closure is unconditionally generated, as prescribed by rule $[? \rightarrow I]$ in Figure 5,
- If the synthesized type of a term is an implicit function type, the term is immediately applied to implicit arguments, as prescribed by rule $[? \rightarrow E]$.

The implicit search algorithm used by *dotc* is a refinement of the algorithm implied by a Figure 2 if the well-scopedness condition is added. In both cases, we choose one of a finite number of implicit candidate variables as the implicit argument so that type checking as a whole succeeds. The differences between the calculus and the full language lie in the question which values are considered candidates and how to choose one if there are several possible alternatives.

4.1 Implicit Search Candidates

In the calculus, the candidates are all implicit variables bound in the environment. In the full language, this is refined as follows: A reference to an implicit value is a candidate for an implicit search of type T if (1) it can be expressed as a simple identifier or (2) it refers to a value of the *implicit scope* of T .

Condition (1) subsumes the condition of the calculus, since the variables bound in the environment are precisely those expressions which can be referred to by a simple identifier. But in the full language there are also other variables which can be referred to that way: A variable might have been made available through an import, or it might have been inherited from a base class.

Condition (2) does not have a counter part in the calculus. This condition also makes available as candidates any implicit values “that are defined with” the type T for which an implicit value is searched. The *implicit scope* of a type T are all implicit definitions in the companion object of T itself, as well as in the companion objects of any part of T , and the companion objects of any base type of T . The purpose of condition (2) is to make implicit values available in a more robust way that does not require an import. The definition of implicit scope is intentionally kept rather large, so that any implicit definition that bears some relationship to the searched-for type is considered.

The full language first searches implicit candidates that are visible in the current scope, i.e. that match condition (1). If none are found it falls back to searching candidates in the implicit scope according to condition (2).

4.2 Disambiguation

The calculus and the full language also differ in how an implicit value is chosen among several candidates. The rules in Figure 2 are silent about this issue, rendering the calculus ambiguous. Adding the well-scopedness condition means that the rightmost (i.e. innermost) matching definition is chosen as is described in Figure 6. The full-language also uses nesting to disambiguate but combines this with specificity. Given two candidate implicit references, it plays a tournament with two rounds.

One round awards a point for candidate c_1 over candidate c_2 if c_1 is introduced (made visible) in a scope more deeply nested than the scope where candidate c_2 is introduced. This round is only applicable to implicit searches in the current scope according to condition (1). For searches in the implicit scope according to condition (2), a candidate c_1 wins instead over c_2 if its associated class is a subclass of the associated class of c_2 . Here, the associated class of an implicit value c is the class where c is defined, or, if c is defined in an object, the object’s companion class.

The other round awards a point for c_1 over c_2 if c_1 ’s type is strictly more specific than c_2 ’s type. This means that c_1 ’s type can be instantiated through widening or polymorphic parameter instantiation to be c_2 ’s type.

If one candidate gets more points in these rounds than the other, it is chosen as implicit instance. In the case of a draw the compiler rejects the program because it is ambiguous.

4.3 Divergence

Since implicit searches can be recursive, it is possible that an implicit search does not terminate. The compiler uses the following strategy to detect divergent searches: It keeps track of all unresolved query types in a stack. For each query type it records its top-level type constructor. If a recursive query is entered which matches the type constructor of a type on the stack, the query is allowed to proceed only if its type arguments are strictly smaller than the arguments for the matching query on the stack. If that condition does not hold, the implicit search fails with a divergence error.

4.4 Optimizations

The basic search algorithm specifies that all implicit values in the current scope or in the implicit scope of the expected types are tried as candidates to resolve a query. Implemented naïvely, this would be very inefficient. The Scala compiler employs two main strategies to speed up the search.

The first strategy eliminates from consideration all implicit variables that cannot possibly match the expected type. To do this, the compiler computes a *fingerprint type*, where all implicit parameters are dropped and all type variables are replaced by a wildcard type that matches any other type. Only implicit values with fingerprints matching the expected type are further considered. This pre-selection is usually beneficial because a fingerprint match is much cheaper than a full type elaboration of an implicit value, which might in turn trigger recursive implicit searches. What's more, the sets of types fingerprint-matching a query type are also cached, so computation is saved if the same query is asked several times.

The second strategy tries to truncate the search space once a first match has been found. If there are several candidates, the compiler normally would need to elaborate the types of all of them in order to determine a best match or raise an ambiguity error. But we can avoid some of these elaborations by checking beforehand whether a candidate could possibly affect the outcome of an implicit search. If the candidate value that was already found would be strictly preferable to the alternative value according to the disambiguation criteria, there is no point in elaborating the type of the alternative value. This scheme is rendered more effective by keeping track of results of previous implicit searches and trying values that were selected most often in previous queries first.

Using these techniques, implicit search becomes reasonably efficient. We have observed implicit search typically takes 30% of the total running time of *dotc*. Of course this depends on the number and size of generated implicit terms, which can be arbitrarily large, so the 30% number is not a hard limit but an indication of average behavior on “typical” programs.

4.5 Language differences

SI has no syntax for the explicit application of an implicit function. Implicit functions are always applied automatically by looking for an implicit value that matches the argument type of the function.

In current Scala this is different in that implicit functions can be applied explicitly - the language uses the normal functional application syntax for this. So the following would typecheck:

```
def f: implicit Context => Map[Int, Int]
val ctx: Context
f(ctx)(2)
```

In the last line of this program, the first parameter `ctx` is passed as an argument to the implicit function whereas the second parameter is passed as the argument to the `apply` function of the result `map` (which is inserted automatically).

While convenient, this convention can also be quite limiting and surprising. For instance, we could *not* have written `f(2)` in the last line of the program, since the compiler is incapable of detecting whether we mean to pass an implicit argument or an explicit one following it. In the particular case above, the programmer can write `f.apply(2)`, which makes it clear that the implicit parameter is to be inferred. But this is roundabout and awkward.

This design choice is by now regarded as a mistake. It would be possible to simply do without explicit applications of implicit functions, just as SI does. If explicit applications were eliminated, the last line of the program above could then be expressed like this:

```
{ implicit val c = ctx; f(2) }
```

For syntactic convenience, the next major version of Scala will have a construct for applying implicit functions explicitly, but it will be different from normal function application. The idea is to introduce a new method named `explicitly` on the function type. `explicitly` converts an implicit function to an explicit one. So if `f` has the implicit function type `implicit Context => Map[Int, Int]` as above, then `f.explicitly` has the normal function type `Context => Map[Int, Int]` and `f.explicitly(ctx)(2)` passes `ctx` as explicit argument to the `Context` parameter of that function. Furthermore, we have the following equivalence, for any unary implicit function `f`:

```
f = f.explicitly(implicitly)
```

5 EXPRESSIVENESS & PERFORMANCE

Implicit function types can be a powerful way to abstract over implicits. In this section, we demonstrate that implicit function types not only promote expressivity, but that they can additionally improve performance over solutions without implicit function types.

To that aim, we show that implicit function types can be used in the expression of many different applications. We introduce four different applications that are made clearer and more concise through the use of implicit function types. The last two of which we show improve performance over existing implementations without implicit function types. In Section 5.1, we present an expressive DSL using the builder pattern, followed by an encoding of tagless interpreters (Carette et al. 2009) demonstrating how we can abstract over the number of implicit parameters in Section 5.2. In Section 5.3, we show how we encode the reader monad with an isomorphic representation using implicit functions, and assess its performance alongside of popular implementations in the Scala ecosystem. Finally, in Section 5.4, we introduce a new implementation of the free monad, and assess its performance as well.

5.1 Builder Pattern: An Expressive DSL

We demonstrate how implicit function types can be used to build declarative API using the type-safe builders pattern (Kotlin 2014) without any boilerplate at the use-site. Let's assume we'd like to define a small DSL to create tables that look like the following:

```
table {
  row {
```

```

    cell("top left")
    cell("top right")
  }
  row {
    cell("bottom left")
    cell("bottom right")
  }
}

```

In this example table, row and cell are function calls to a factory method. Every call creates a new node (resembling an AST), and registers this newly created node to its parent. Implicit functions are used to propagate references of parent nodes to newly created nodes. For instance, the table method takes an `implicit Table => Unit` as argument:

```

class Table {
  val rows = new ArrayBuffer[Row]
  def add(r: Row): Unit = rows += r
  override def toString = rows.mkString("Table(", ", ", ", ")")
}
def table(init: implicit Table => Unit): Table = {
  implicit val t = new Table
  init
  t
}

```

The Definitions of row and cell are analogous. After desugaring and implicit resolution, the table construction above is translated into the following:

```

table { $t: Table =>
  row { $r: Row =>
    cell("top left")($r)
    cell("top right")($r)
  }($t)
  row { $r: Row =>
    cell("bottom left")($r)
    cell("bottom right")($r)
  }($t)
}

```

5.2 Tagless Interpreters: Abstracting Over Multiple Constraints

Implicit function types enable a very useful way to abstract over the number of implicit parameters introduced in some scope. We demonstrate a usage of this abstraction through an implementation of the tagless interpreter pattern (Carette et al. 2009) (or object algebra (Oliveira and Cook 2012) as popularized in the OOP domain) for simple arithmetic expressions.

Tagless interpreters make it possible to add both new syntactic elements and interpretations without breaking the existing hierarchy, thus serving as a solution to the Expression Problem (Wadler 1998).

Below, we show the tagless encoding of a toy language for simple arithmetic expressions with only two constructs; `lit` and `add`. (Though we would ultimately like to add more constructs!)

```

trait Exp[T] {
  def lit(i: Int): T
  def add(l: T, r: T): T
}
object ExpSyntax {
  def lit[T](i: Int) (implicit e: Exp[T]): T = e.lit(i)
  def add[T](l: T, r: T)(implicit e: Exp[T]): T = e.add(l, r)
}

```

To define an expression on integers using these two constructs, we first need to implement an interpreter to evaluate integer expressions as follows:

```

implicit val evalExp: Exp[Int] = new Exp[Int] {
  def lit(i: Int): Int = i
  def add(l: Int, r: Int): Int = l + r
}

```

With that, we can now define an expression in the Exp language. Using what we have so far, we can define $8 + (1 + 2)$ as follows:

```

def tf1[T](implicit e: Exp[T]): T = add(lit(8), add(lit(1), lit(2)))

```

Allowing one to interpret tf1 as follows:

```

val evaluated: Int = tf1()
println(evaluated) // 11

```

To extend the Exp language to be able to handle multiplication, we define a new trait Mult with the new multiplication operation we'd like to add:

```

trait Mult[T] {
  def mul(l: T, r: T): T
}
object MultSyntax {
  def mul[T](l: T, r: T)(implicit e: Mult[T]): T = e.mul(l, r)
}

```

Ultimately, we'd like to define an expression to evaluate multiplication and additions in the same expression. We can define $7 + (1 * 2)$ as follows:

```

def tfm1[T](implicit e: Exp[T], m: Mult[T]): T = add(lit(7), mul(lit(1), lit(2)))

```

Interpreting tfm1 requires two instances of interpreters; it requires two implicit parameters, one of type Exp[T] and another of type Mult[T]. We can use evalExp for addition, and we need another for multiplication, which can be defined as follows:

```

implicit val evalMult: Mult[Int] = new Mult[Int] {
  def mul(l: Int, r: Int): Int = l * r
}

```

As we observe, by increasing the number of interpreters, we increase the number implicit parameters. If we continue extending our toy language in this fashion, we have no way to abstract over these new extensions as they pile up in parameter lists.

Implicit function types enable us to easily abstract over implicit parameters using type aliases:

```
type ExtExp[T] = implicit (Exp[T], Mult[T]) => T
```

This enables us to define `tfm1` much more concisely, allowing us to omit the two implicit parameters we had to write above, by using the `ExtExp[T]` type alias:

```
def tfm1[T]: ExtExp[T] = add(lit(7), mul(lit(1), lit(2)))
```

5.3 Reader Monad: Use Contextual Abstraction

The reader monad represents a computation with the ability to read from an environment. It is defined in term of two operations, `ask`, to retrieve the environment, and `local`, to modify the environment for sub-computations (monad instance omitted):

```
trait Reader[R, A] {
  def ask: R
  def local[A](f: R => R)(a: A): A
}
```

Reader expressions are formed using the `for`-comprehensions (equivalent to the `do`-notation in Haskell):

```
val expr1: Reader[Env, Int] = ...
val expr2: Reader[Env, Int] = ...
val expr3: Reader[Env, Int] =
  for {
    e1 <- expr1
    e2 <- expr2
  } yield e1 + e2
```

Implicit function types can be used as a concise alternative to `Reader`:

```
type ReaderIFT[T] = implicit Env => T
```

Values of type `ReaderIFT[T]` automatically obtain an `Env` from the implicit context, and propagate this value to all sub-expressions in the right-hand side of their definition:

```
val expr1: ReaderIFT[Int] = ...
val expr2: ReaderIFT[Int] = ...
val expr3: ReaderIFT[Int] = expr1 + expr2
```

`Env` values can be obtained using `implicitly[Env]` in the body of `ReaderIFT` expression, which corresponds to the `ask` operation on `Reader`. Analogously, a new `Env` can be defined for all sub-expressions via by defining a local implicit of that type.

This pattern is very common in large-scale applications. For instance, in web programming, the majority of functions take a context argument to propagate information about the request that is currently being processed. Implicits provide a simple and concise way to transmit this information across such applications.

5.3.1 Performance Evaluation. We compare the implicit function type encoding of the reader monad with implementations from two widely-used Scala libraries: Scalaz 7.2 (Scalaz 2017) and Cats 0.9 (Cats 2017). Scalaz and Cats are general-purpose, functional programming libraries that provide definitions of standard type classes and implementations of commonly used functional data structures, including the reader monad.

We benchmark¹ the aforementioned implementations using the monad reader laws:²

ask-idempotent $\text{flatMap}(\text{ask})(x \Rightarrow \text{ask}) \equiv \text{ask}$
local-ask $\forall f : \text{local}(f)(\text{ask}) \equiv \text{map}(\text{ask})(f)$
local-flatmap $\forall r, f, g : \text{local}(g)(\text{flatMap}(r)(f)) \equiv \text{flatMap}(\text{local}(g)(r))(x \Rightarrow \text{local}(g)(f(x)))$
local-pure $\forall f, a : \text{local}(f)(\text{pure}(a)) \equiv \text{pure}(a)$

Four micro-benchmarks comprise our evaluation set; each one represents a reader monad law instantiated with a String environment and identity functions.

Results. Figure 7a shows a 4x to 9x improvement of throughput for the implicit function type implementation. This large gap is a consequence of removing a layer of abstraction. Indeed, the implicit function type encoding is equivalent to manually propagating the environment as function arguments, whereas both Cats and Scalaz store reader computations in an intermediate data structure and use type classes for operations. These performance improvements are expected, given that Scala compilers do not inline type classes.

5.4 Free Monads

Data types à la carte (Swierstra 2008) popularized the free monad pattern, an abstraction to decouple monadic expressions from their semantics. Typical Scala implementations (Bjarnason 2012) of this idea use a generalized algebraic data type (GADT) to encapsulate the monadic structure, called Free:

```
sealed trait Free[A[_], T]
case class Pure[A[_], T](a: T) extends Free[A, T]
case class Suspend[A[_], T](a: A[T]) extends Free[A, T]
case class FlatMapped[A[_], B, C](c: Free[A, C], f: C => Free[A, B]) extends Free[A, B]
```

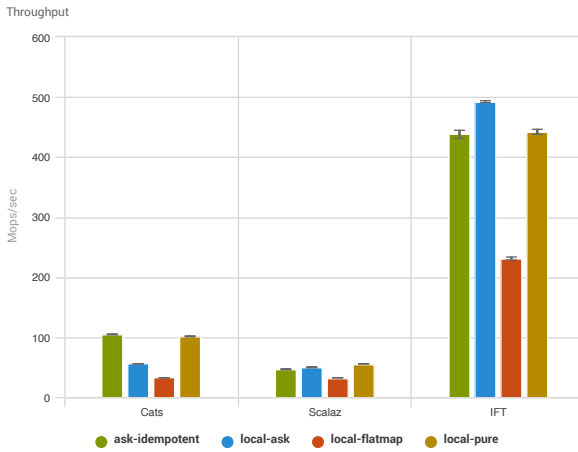
Interpreters for Free expressions are defined as natural transformations, that is, instances of the Natural trait:

```
trait Natural[F[_], G[_]] {
  def apply[A](fa: F[A]): G[A]
}
```

¹The measurements obtained are the average of 10 runs executed on an i7-7700K Processor CPU running Oracle JVM 1.8.0 on Debian 9.0 with binaries produced by scalac 2.12.2. We use the Java Microbenchmark Harness (JMH) (Shipilev et al. 2014) tool with default settings: each run averages throughput of execution over 1 second period for 20 warm-up iteration and 20 measurements iteration.

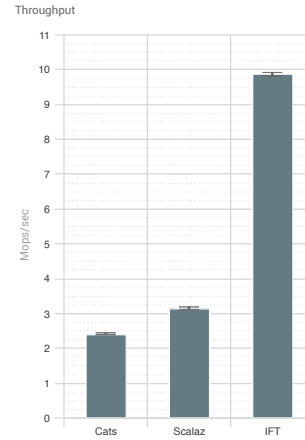
²<https://github.com/typelevel/cats/blob/v0.9.0/laws/src/main/scala/cats/laws/MonadReaderLaws.scala>

Reader monad implementations



(a) Reader Monad

Free monad implementations



(b) Free Monad

Fig. 7. JVM micro-benchmarks measuring throughput of various implementations in Mops/sec. Average of 10 runs, error bars for 99.9% confidence intervals.

Interpretation of Free expressions is done through a foldMap function:

```
def foldMap[A[_], T, M[_]](e: Free[A, T])(n: Natural[A, M])(implicit m: Monad[M]): M[T] = ...
```

Users of free monads typically depend on a library that provides the definition for Free, foldMap, and a monad instance for Free. These definitions are non-trivial and duplication is required to support other free structures such as the free applicative functor.

5.4.1 An Implicit Function Type Encoding. Implicit function types open the door to an alternative design of free monad that is simpler to use, more efficient and doesn't require any library infrastructure.

The new design defines a FreeIFT type alias with two curried implicit function types, mirroring the signature of foldMap:

```
type FreeIFT[A[_], M[_], T] = implicit Natural[A, M] => implicit Monad[M] => M[T]
```

In this new encoding of free monads, expressions are function type parametric in the monad used for interpretation. For instance, a free expression with made Put and Get operations (subtypes of KVStore) can be defined as follows:

```
type KVStoreIFT[M[_], T] = FreeIFT[KVStore, M, T]

def iftExpr[M[_]]: KVStoreIFT[M, Option[Int]] =
  for {
    _ <- lift(Put("foo", 2))
    _ <- lift(Put("bar", 5))
    n <- lift(Get("foo"))
  } yield n
```

Where the `lift` method is used to apply an implicit natural transformation:

```
def lift[F[_], M[_], A](fa: F[A])(implicit F: Natural[F, M]): M[A] = F(fa)
```

Interpreters are, as in the traditional encoding, natural transformations. However, the interpretation doesn't require a library defined `foldMap` function. Instead, it is a simple function application of `iftExpr` to an interpreter:

```
def iftInterpreter = new Natural[KVStore, Future] {
  def apply[T](fa: KVStore[T]): Future[T] = ...
}
val iftOutput: Future[Option[Int]] = iftExpr[Future](iftInterpreter)
```

5.4.2 Comparing Encodings. The main benefit of the implicit function type encoding of free monad is that it doesn't require any library support. From a user perspective, there is also less boilerplate involved (the `put` and `get` function become unnecessary).

The two encodings can be shown to be equivalent by defining a bijection between representations. The conversion to the *À la carte encoding* is done via interpretation from `A` to `Free[A, ?]`:³

```
def initialize[A[_]] = new Natural[A, [T] => Free[A, T]] {
  def apply[T](a: A[T]): Free[A, T] = Suspend[A, T](a)
}
```

Conversion from the *À la carte encoding* also goes through an interpretation, from `A` to new `Expr` trait, which captures the polymorphism in expressions by the implicit function type encoding⁴:

```
trait Expr[A[_], T] {
  def e[M[_]]: FreeIFT[A, M, T]
}

def finalize[A[_]] = new Natural[A, [T] => Expr[A, T]] {
  def apply[T](a: A[T]): Expr[A, T] = new Expr[A, T] {
    def e[M[_]]: FreeIFT[A, M, T] = a.lift
  }
}
```

5.4.3 Performance Evaluation. The benchmark simulates a state monad with an expression that counts to 10 by successively reading from the state, incrementing by 1, and writing back to the state. Interpreters are implemented by mutating a local variable.

Results. This new approach shows significant improvements in term of runtime performance. As opposed to the traditional encoding of free monad, the implicit function type encoding does not allocate any intermediate structure to capture the monadic structure. Instead, the interpretation flows directly through the definition.

Figure 7b shows the throughput for creating and interpreting of a simple expression using different free monad implementations. The Scalaz implementation closely follows the pattern

³In this example `[T] => Expr[A, T]` uses the syntax for type lambdas introduced by the Dotty compiler.

⁴Language support for polymorphic functions would remove the need for a `Expr` trait.

described in the Data Types à la Carte paper (Swierstra 2008). The Cats implementation is a slight simplification over Scalaz', in that it does not require a functor instance. The implicit function type encoding described in this section has the best performance among all implementations.

Overall, Scalaz' infrastructure about free monads is over 500LOC. Cats' is on the same order of magnitude, with about 250LOC. This new encoding is able to implement equivalent functionalities with a single type alias. We expect the pattern of factoring out type class constraints to be applicable to a large number of use cases. For example, this technique can be used to build free counterparts of other type classes such as applicative functors and co-monads.

6 RELATED WORK

The Haskell programming language supports a version of implicit parameters (Lewis et al. 2000) through the `ImplicitParams` language extension. Implicit parameters in Haskell are orthogonal to the built-in type class mechanism, but are used in a similar manner as constraints on functions.

Even though they come under the same name, implicit parameters in Scala and Haskell are different in several aspects. Implicit parameters in Scala trade explicit types for implicit terms. If the type makes it clear that an implicit parameter is in scope, that parameter can be used implicitly as an argument to another function. By contrast, implicit parameters in Haskell are synthesized using an explicit query such as `?cmp` which also specifies the name of the searched parameter.

In Haskell, implicit parameter constraints are then propagated automatically to the calling context in the inferred type of a function. This automatic propagation makes it look like implicit parameters support a form of dynamic scoping as was explained in the original paper (Lewis et al. 2000), even though O. Kiselyov showed the correspondence is not exact.⁵

By contrast, Scala always demands the function's type to be given explicitly, and, if no implicit parameter is mentioned, resolves any implicit queries at the point of definition instead of propagating them to the callsite. This is much more predictable, at the price of a heavier notation for function definitions. As we have shown here, the notation overhead can be reduced substantially using type aliases of implicit function types.

These differences between the implicit parameter rules in Haskell and Scala might seem relatively minor, but they have a large impact on typical use cases and overall usability.

Agda instance arguments (Devriese and Piessens 2011) are closely related to Haskell's type class constraints and were inspired by both Scala's implicits and Agda's existing implicit arguments. With implicit function types, we lift the limitations that are described in Devriese et. al (Devriese and Piessens 2011, Section 1.5) for Scala. Additionally, we present a self-contained high-level formalization, whereas Devriese et. al present a more algorithmic description on how to modify Agda's existing implicit search mechanism. In the aforementioned paper instance arguments do not support implicit definitions of functions that take implicit arguments. However, Agda supports them with the use of the keyword `instance` (Norell et al. 2017).

Modular implicits (White et al. 2015) propose an extension to the OCaml language for ad-hoc polymorphism using modules as the types of implicit parameters. Contrarily to Scala, implicits in OCaml always raise an ambiguity error in the presence of multiple implicit modules. Modular implicits support functions with implicit arguments and during elaboration they translate them into first-class functors. However they do not provide dedicated support for the equivalent of implicit function types.

Coq supports notation overloading and implicit program construction through canonical structures (Gonthier et al. 2011) and type classes (Sozeau and Oury 2008). Canonical structure instances

⁵O. Kiselyov discussing Haskell's 'implicit parameters' are not dynamically scoped—<https://web.archive.org/web/20170708010400/http://okmij.org/ftp/Computation/dynamic-binding.html#implicit-parameter-neq-dynvar>

are record types that are used to solve equations, during type-checking, involving implicit arguments. However, overlapping instances are not supported in Coq, and since they are required for numerous scenarios involving canonical structure instances, they need to be restored with design patterns. This is necessary for lemma overloading capabilities as described in Gonthier et al. (Gonthier et al. 2011). On the contrary, the capability to backtrack during type inference, enables dependently-typed logic programming.

6.1 Implicit Calculi

The Implicit Calculus (Oliveira et al. 2012) provides a general formalization of implicit programming. It supports partial resolution and higher-order rules. In the calculus, it introduces *rule types* which are similar to our implicit function types.

Cochis (Schrijvers et al. 2017) is a recent calculus that tries to combine the strength of Haskell typeclasses and Scala implicits, i.e. the combination of ease of reasoning and flexibility. Cochis supports local implicits and meanwhile guarantees coherency. Coherency in Cochis means that substitution of equals doesn't change the semantics of programs, which is a reasonable property of pure functional programs.

According to the Cochis paper, Scala implicits are incoherent. However, this claim is contestable, as it depends on the time when the substitution happens. If we take equivalent expressions, directly from source code “substituting equals by equals”, then indeed the semantics might change. But that's neither how the compiler reasons about the code nor what the programmer expects. Otherwise, by doing so, we can turn a well-typed program into an ill-typed program. If we consider “substitution of equals by equals” happening after type checking, as it's done in the compiler, of course Scala implicits are coherent, and that's what programmers expect.

Both of the two calculi mentioned above depend on an explicit type query $?T$ to trigger implicit resolution for an instance of a type T . It is noticeable that explicit type query loses the essential appeal of implicit programming. In this regard, our calculus is closer to Scala implicits.

The calculi mentioned above, describe more powerful implicit resolution algorithms. However, the implementation of Scala implicits restrains the power of implicit resolution. Weird or complex resolutions harm maintainability of the code, and may give rise to tricky bugs in programs. Consequently we keep implicit resolution simple and intuitive to programmers. For this reason, our calculus better models implicits in practical programming languages.

The two papers mentioned above are more oriented towards resolution algorithms and the connection between logic and resolution. In this paper we provide a calculus that takes the first step on contextual abstraction in a practical programming language, based on implicit function types.

Finally, A. Rouvoet (Rouvoet 2016) presents a formal development of implicits, based on Oliveira et al. (Oliveira et al. 2012). The author provides a provably complete, syntax-directed resolution algorithm that either finds the solution or diverges. The author also studies a family of termination conditions that may be added to ensure the termination of the resolution. Although SI is not syntax-directed, it provides a well-scopedness condition that ensures unique type derivations, but only for the monomorphic fragment of SI.

7 CONCLUSION

94% of the 120 most popular Scala projects hosted on GitHub take full advantage of implicits. Their widespread usage patterns indicate that real world applications prefer to pass context *implicitly*. We propose *implicit function types* as a simple and powerful language feature for dealing with contexts in programming and we present a formal development of implicits, as a whole, in the system SI.

The applications we present, highlight emerging encodings, that promote expressivity without sacrificing performance.

REFERENCES

- Runar Bjarnason. 2012. Stackless Scala With Free Monads. <https://web.archive.org/web/20170107134129/http://blog.higher-order.com/assets/trampolines.pdf>. (April 2012).
- Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.* 19, 5 (Sept. 2009), 509–543. DOI : <http://dx.doi.org/10.1017/S0956796809007205>
- Cats. 2017. Cats: Lightweight, modular, and extensible library for functional programming. <https://web.archive.org/web/20160219022626/https://github.com/typelevel/cats>. (2017).
- Thierry Coquand. 1996. An algorithm for type-checking dependent types. *Science of Computer Programming* 26, 1 (1996), 167 – 177.
- Luis Damas and Robin Milner. 1982. Principal Type-schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*. ACM, New York, NY, USA, 207–212.
- Dominique Devriese and Frank Piessens. 2011. On the Bright Side of Type Classes: Instance Arguments in Agda. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 143–155.
- Burak Emir, Andrew Kennedy, Claudio Russo, and Dachuan Yu. 2006. Variance and Generalized Constraints for C# Generics. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*. Springer-Verlag, Berlin, Heidelberg, 279–303.
- GHC. 2015. Glasgow Haskell Compiler Users Guide. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html. (2015).
- Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. Dissertation. PhD thesis, Université Paris VII.
- Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. 2011. How to Make Ad Hoc Proof Automation Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 163–175.
- Rod Johnson. 2002. *Expert One-on-One J2EE Design and Development*. Wrox Press Ltd., Birmingham, UK, UK.
- Mark P. Jones. 1995. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Springer-Verlag, London, UK, UK, 97–136.
- Kotlin. 2014. Type-Safe Groovy Style Builders in Kotlin. <https://web.archive.org/web/20170607150651/https://kotlinlang.org/docs/reference/type-safe-builders.html>. (2014).
- Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. 2000. Implicit Parameters: Dynamic Scoping with Static Types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*. ACM, New York, NY, USA, 108–118.
- Ulf Norell and others. 2017. Agda Documentation. <https://web.archive.org/web/20170618112351/https://media.readthedocs.org/pdf/agda/v2.5.2/agda.pdf>. (March 2017).
- Martin Odersky. 2017. Dotty Compiler: A Next Generation Compiler for Scala. <https://web.archive.org/web/20170325001401/http://dotty.epfl.ch/>. (2017).
- Martin Odersky and Matthias Zenger. 2005. Scalable Component Abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 41–57.
- Martin Odersky, Matthias Zenger, and Christoph Zenger. 2001. Colored Local Type Inference. In *Proc. ACM Symposium on Principles of Programming Languages*. 41–53.
- Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses: Practical Extensibility with Object Algebras. In *Proc. of the 26th European Conference on Object-Oriented Programming*. ECOOP '12, Vol. 7313. Springer Berlin Heidelberg, 2–27.
- Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type Classes As Objects and Implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 341–360. DOI : <http://dx.doi.org/10.1145/1869459.1869489>
- Bruno C. d. S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. 2012. The Implicit Calculus: A New Foundation for Generic Programming. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 35–44.

- Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical Type Inference for Arbitrary-rank Types. *J. Funct. Program.* 17, 1 (Jan. 2007), 1–82.
- Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44.
- John Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium*. Springer, 408–425.
- Arjen Rouvoet. 2016. *Programs for Free: Towards the Formalization of Implicit Resolution in Scala*. Master’s thesis. TU Delft.
- Scalaz. 2017. Scalaz: An extension to the core Scala library for functional programming. <https://web.archive.org/web/20170305041357/https://github.com/scalaz/scalaz/>. (2017).
- Tom Schrijvers, Bruno C d S Oliveira, and Philip Wadler. 2017. *Cochis: Deterministic and Coherent Implicits*. Technical Report CW705. KU Leuven.
- Aleksey Shipilev and others. 2014. OpenJDK Code Tools: jmh. <https://web.archive.org/web/20161211145153/http://openjdk.java.net/projects/code-tools/jmh/>. (2014).
- Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLS ’08)*. Springer-Verlag, Berlin, Heidelberg, 278–293.
- Wouter Swierstra. 2008. Data Types à La Carte. *J. Funct. Program.* 18, 4 (July 2008), 423–436.
- Robbie Vanbrabant. 2008. *Google Guice: Agile Lightweight Dependency Injection Framework (Firstpress)*. APress.
- Philip Wadler. 1998. The Expression Problem. <https://web.archive.org/web/20170322142231/http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt>. (Dec. 1998).
- P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’89)*. ACM, New York, NY, USA, 60–76.
- Adam Warski. 2013. MacWire: Lightweight and Nonintrusive Scala Dependency Injection Library. <https://web.archive.org/web/20170222065306/https://github.com/adamw/macwire>. (2013).
- L. White, F. Bour, and J. Yallop. 2015. Modular implicits. *ArXiv e-prints* (Dec. 2015). arXiv:cs.PL/1512.01895
- Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’99)*. ACM, New York, NY, USA, 214–227.

A APPENDIX: USE OF IMPLICITS IN THE SCALA ECOSYSTEM

We performed a small empirical analysis of the most popular Scala projects on GitHub. The projects in our corpus range from widely-used frameworks for big data processing like Apache Spark, to open source and mission-critical systems at companies like Twitter, to widely depended-on open source web frameworks, to community-built libraries for functional programming, and more.

We selected the top 120 Scala projects hosted on GitHub, ranked by their star count⁶. Table 1 lists all 120 projects we analyzed. On average, projects contained 31,135 lines of code, and had 1,977 stars. In total, we analyzed over 3.7 million lines of Scala code.

Our analysis was syntactic; we counted definitions of different kinds of implicits in each code base, and how they were spread out in each code base. In particular, we looked at how much these code bases made use of `implicit vals`, `implicit defs`, `implicit objects`, and implicit parameters.

Results. Some of the more interesting insights are shown in Figure 8. Out of the 120 Scala projects analyzed, 94.17% made use of some form of implicits. Interestingly, we found that implicit parameters were the most commonly-preferred form of implicit across all code bases. That is, 42% of all usages of implicits across all projects analyzed are implicit parameters, and 30.39% are `implicit vals`. Taken together with the fact that 84% of all projects analyzed make use of implicit parameter lists specifically, this leads us to believe that real-world applications seem to overwhelmingly prefer to pass context implicitly.

⁶Stars are a way of “liking” projects on GitHub. A project with many stars doesn’t necessarily mean that it is widely-used. In our case, however, all but one or two projects analyzed are indeed well-known and widely depended-on or used projects.

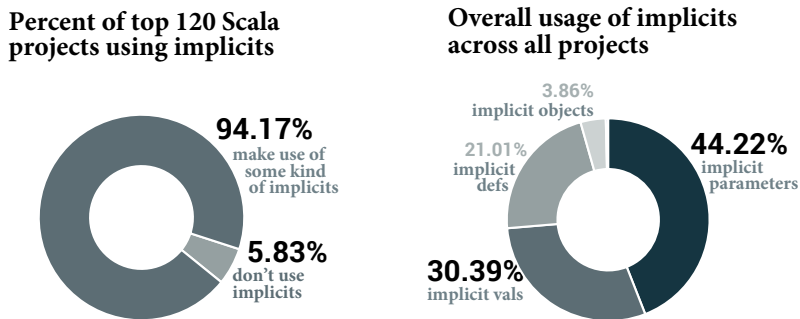


Fig. 8. Usage of implicits in the Scala ecosystem.

Project	Stars				
apache/spark	13365	laurilehmijoki/s3_website	1800	filodb/FiloDB	876
apache/incubator-predictionio	10267	lampepfl/dotty	1759	pathikrit/better-files	874
playframework/playframework	9495	spark-jobserver/spark-jobserver	1653	jaggolly/scalajs-react	874
scala/scala	8478	apache/incubator-openwhisk	1641	tpolecat/doobie	868
shadowsocks/shadowsocks-android	7969	twitter/finatra	1605	kamon-io/Kamon	859
akka/akka	7305	twitter/algebird	1527	vkostyukov/scalacaster	850
gitbucket/gitbucket	6424	mesos/spark	1462	sbt/sbt-native-packager	844
twitter/finagle	5799	GravityLabs/goose	1427	functional-streams-for-scala/fs2	839
lhartikk/ArnoldC	4992	lagom/lagom	1414	lihaoyi/Metascala	829
airbnb/aerosolve	3961	Netflix/atlas	1406	scala/pickling	816
yahoo/kafka-manager	3816	lihaoyi/Ammonite	1319	sryza/aas	814
mesos/chronos	3750	PkmX/lcamera	1284	eligosource/eventsourced	811
twitter/snowflake	3513	twitter/iago	1243	monix/monix	809
snowplow/snowplow	3432	rickynils/scalacheck	1231	akka/reactive-kafka	800
mesosphere/marathon	3333	datastax/spark-cassandra-connector	1222	sryza/spark-timeseries	799
ornicar/lila	3250	jaliss/secsuresocial	1211	scala/async	796
rtyley/bfg-repo-cleaner	3235	guardian/grid	1197	lihaoyi/scala.rx	791
fpinscala/fpinscala	3189	ensime/ensime-server	1193	julien-truffaut/Monocle	789
scalaz/scalaz	3139	non/spire	1192	http4s/http4s	787
sbt/sbt	3115	lw-lin/CoolplaySpark	1186	twitter/ostrich	782
twitter-archive/flockdb	3112	foundweekends/giter8	1158	sangria-graphql/sangria	778
gatling/gatling	3049	lift/framework	1090	jrudolph/sbt-dependency-graph	768
scala-js/scala-js	3012	mpeltonen/sbt-idea	1085	scalikejdbc/scalikejdbc	765
scala-native/scala-native	2885	finagle/finch	1065	databricks/spark-csv	764
twitter/diffy	2858	scala-exercises/scala-exercises	1051	twitter/twitter-server	734
twitter/scalding	2839	quantifind/KafkaOffsetMonitor	1048	ReactiveMongo/ReactiveMongo	718
twitter-archive/kestrel	2780	mauricio/postgresql-async	1041	adamw/macwire	711
spray/spray	2523	killrweather/killrweather	1018	playframework/play-slick	706
linkerd/linkerd	2315	ThoughtWorksInc/Binding.scala	994	jdegoes/blueeyes	702
scalatra/scalatra	2188	tumblr/colossus	989	nscala-time/nscala-time	696

Table 1. Top 120 open source Scala projects on GitHub, by star count.