Simplicitly

Foundations and Applications of Implicit Function Types

MARTIN ODERSKY, EPFL, Switzerland
OLIVIER BLANVILLAIN, EPFL, Switzerland
FENGYUN LIU, EPFL, Switzerland
AGGELOS BIBOUDIS, EPFL, Switzerland
HEATHER MILLER, EPFL, Switzerland and Northeastern University, USA
SANDRO STUCKI, EPFL, Switzerland

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.

CCS Concepts: • **Software and its engineering → Multiparadigm languages**; **Formal language definitions**; **Compilers**;

Additional Key Words and Phrases: implicit parameters, Scala, Dotty

ACM Reference Format:
Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2018. Simplicitly: Foundations and Applications of Implicit Function Types. *Proc. ACM Program. Lang.* 2, POPL, Article 42 (January 2018), 30 pages. https://doi.org/10.1145/3158130

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 is too much of a good thing. Programs can quickly become riddled with long parameter lists. A number of programming techniques have been

2475-1421/2018/1-ART42
https://doi.org/10.1145/3158130

invented to combat that problem. Currying and point-free style, or the reader monad [Jones 1995], are some examples. Nevertheless, it is 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 word 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) and the implementation of implicit function types in the compiler Dotty (Section 4). 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. The ability to omit arguments to functions gives rise to many interesting coding styles and patterns that revolve around the concept of *contextual abstraction*. The contextual property in the following code is the *evidence* that a definition that a method needs appears implicitly. On every call to functions 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*. The resolution algorithm looks for implicits 1) in the current scope and 2) in the companion objects if all classes associated with the query type. 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 type classes [Wadler and Blott 1989] as a design pattern in Scala. We give an example of an 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)
```

We have briefly introduced implicit parameters and the kinds of contextual abstraction they can offer. They are very helpful to avoid clutter in function applications, but implicit formal parameter lists still have to be explicitly written in every method that has them, which can lead to unwanted verbosity. The next subsection introduces a remedy to this problem by providing the means to type-abstract implicit functions.

2.1 Introducing Implicit Function Types

Implicit Function Types are types for implicit function values. They extend the support for first-class functions to implicit arguments. Whereas `A => B` is the type of functions from `A` to `B`, `implicit A => B` is the type of functions that take an implicit argument of type `A` to a result of type `B`. Using an implicit function type, the `comp` function can 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` the implicit argument of type `Ord[T]` is anonymous. The compiler will desugar the body of this definition into `{ implicit ev$n: Ord[T] => ... }` where `ev$n` is a freshly generated name. The implicit argument can still be referred using the idiom `implicitly[Ord[T]]`. `implicitly` is defined in the standard library as the implicit identity function:

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

The new formulation of `comp` might seem like a minor syntactic twist, but it has far-reaching consequences. Because `implicit Ord[T] => Boolean` is now a *type*, it can be abstracted over by giving it a name and using only the name afterwards. This is exploited the following example, which shows how to approach the *configuration problem* [Kiselyov and Shan 2004] in a type-safe manner. The goal is to propagate a run-time value, the modulus, while doing arithmetic with the usual functions, `add` and `mul`, uncluttered.

```
case class Modulo[T](m: T)

type WithModulo[T] = implicit Modulo[Int] => T // Implicit Function Type

def add(a: Int, b: Int): WithModulo[Int] = (a + b) % implicitly[Modulo[Int]].m
def mul(a: Int, b: Int): WithModulo[Int] = (a * b) % implicitly[Modulo[Int]].m

def test1(a: Int, b: Int): WithModulo[Int] =
  add(mul(a, a), mul(b, b))

implicit val mod4 = Modulo(4)
val output = test1(2, 3)
```

`test1` implements the operation: $(a * a) + (b * b)$ in a very clean and concise manner. The implicit value `mod4`, that is brought into scope, propagates the run-time value 4 in our test case and what is performed instead is the operation $((a * a) \bmod 4 + (b * b) \bmod 4) \bmod 4$. We achieved two things: a) to transform a configuration requirement into a type that communicates this context-passing behavior and b) to have compile-time guarantees that all methods are going to be passed the same value.

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		$T \longrightarrow T$	function type
	λx.t	function			
	t t	application	S, T =		Full type
	?	implicit query		R	restricted type
	let x : T = t **in** t	explicit let		$T\ ?\!\!\longrightarrow T$	implicit function type
	let ? : T = t **in** t	implicit let		∀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 of maintaining 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, i.e. without domain annotations on the bound variables. Instead, type annotations are introduced separately through let-expressions. There are two variants of let-bindings: explicit and implicit. The latter introduce implicit values.

Another peculiarity is that there is no syntax for implicit function abstraction, nor for type instantiation or implicit function application. All these operations are to be inferred. The syntax is hence quite familiar to a Haskell programmer, in that types are given separately for let-bound variables, closures do not carry type annotations themselves and all type arguments are inferred.

In the calculus, implicit parameters get resolved and applied automatically. The query operator (denoted by ?) 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: readers familiar with Scala might object 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 the ? operator.)

For types, in addition to normal function types, we have implicit function types ($T\ ?\!\!\longrightarrow T$). 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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x \;:\triangleright T}\ (\textsc{Var}) \qquad \frac{y : T \in \Gamma}{\Gamma \vdash ? \;:\triangleright T}\ (\textsc{Query})$$

$$\frac{\Gamma,\ x : S \vdash t \triangleleft\!: \ T}{\Gamma \vdash \lambda x.t \triangleleft\!: \ S \longrightarrow T}\ (\longrightarrow \mathrm{I}) \qquad \frac{\Gamma \vdash t_1 \;:\triangleright S \longrightarrow T \qquad \Gamma \vdash t_2 \triangleleft\!: \ S}{\Gamma \vdash t_1\ t_2 \;:\triangleright T}\ (\longrightarrow \mathrm{E})$$

$$\frac{\Gamma,\ y : S \vdash t \triangleleft\!: \ T}{\Gamma \vdash t \triangleleft\!: \ S\ ?\!\!\longrightarrow T}\ (?\!\!\longrightarrow \mathrm{I}) \qquad \frac{\Gamma \vdash t \;:\triangleright S\ ?\!\!\longrightarrow T \qquad \Gamma \vdash ? \triangleleft\!: \ S}{\Gamma \vdash t \;:\triangleright T}\ (?\!\!\longrightarrow \mathrm{E})$$

$$\frac{\Gamma,\ X \vdash t \triangleleft\!: \ T}{\Gamma \vdash t \triangleleft\!: \ \forall X.T}\ (\forall\ \mathrm{I}) \qquad \frac{\Gamma \vdash t \;:\triangleright \forall X.T}{\Gamma \vdash t \;:\triangleright [X := S]T}\ (\forall\ \mathrm{E})$$

$$\frac{\Gamma \vdash t \triangleleft\!: \ T \qquad \Gamma, x : T \vdash s \;:\triangleright R}{\Gamma \vdash \mathbf{let}\ x : T = t\ \mathbf{in}\ s \;:\triangleright R}\ (\textsc{Let-Ex}) \qquad \frac{\Gamma \vdash t \triangleleft\!: \ T \qquad \Gamma, y : T \vdash s \;:\triangleright R}{\Gamma \vdash \mathbf{let}\ ? : T = t\ \mathbf{in}\ s \;:\triangleright R}\ (\textsc{Let-Im})$$

$$\frac{\Gamma \vdash t \;:\triangleright R}{\Gamma \vdash t \triangleleft\!: \ R}\ (\textsc{Stitch})$$

Fig. 2. SI Typing Rules

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 in one direction and a term t can be checked to have a type T in 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.

The rule (Var) is standard. Its implicit analogue (Query) can be thought of 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 3.5 how to address this source of non-determinism.

The typing rule ($\longrightarrow$ 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 ($\longrightarrow$ 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 $S \longrightarrow T$ of the function t_1. The typing rule ($?\!\!\longrightarrow$ 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 is 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 ($?\!\!\longrightarrow$ E) is where automatic resolution of implicits happens. This rule says that if a term t is of the implicit function type $S\ ?\!\!\longrightarrow T$ 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 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 ($\forall$ I) and ($\forall$ 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, ($\forall$I) instances are inserted deterministically depending on the expected type. For ($\forall$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 (?$\rightarrow$ I), and universal types will go through the rule ($\forall$ E) before the rule is applied. Conversely, the introduction rules (?$\rightarrow$ I) and ($\forall$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 (?$\rightarrow$ 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 (?$\rightarrow$ 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 $T \mathbin{?\!\!\rightarrow} T'$, create an implicit closure by entering in the environment an anonymous implicit value and proceed by 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 $T \mathbin{?\!\!\rightarrow} T'$, perform an implicit search for T. If a unique term t' is found that matches T, continue by type checking $t\ t'$.

3.3 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:

$$\begin{array}{lcll}
(S \mathbin{?\!\!\rightarrow} T)^* & = & S^* \rightarrow T^* & \\
(S \rightarrow T)^* & = & S^* \rightarrow T^* & \\
(\forall X.T)^* & = & \forall X.T^* & \\
T^* & = & T & \text{otherwise}
\end{array}$$

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:

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

The translation rules are presented in Figure 3. 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* $\varnothing \vdash t : T \leadsto t'$, *then* $\varnothing \vdash t' : T^*$.

Proof. Straight-forward induction on typing derivations. □

$$\frac{x : T \in \Gamma}{\Gamma \vdash x \mathrel{:\triangleright} T \rightsquigarrow x}\;(\textsc{Var-Ex}) \qquad \frac{y : T \in \Gamma}{\Gamma \vdash ? \mathrel{:\triangleright} T \rightsquigarrow y}\;(\textsc{Query})$$

$$\frac{\Gamma,\ x : S \vdash t \mathrel{\triangleleft:}\ T \rightsquigarrow u}{\Gamma \vdash \lambda x.t \mathrel{\triangleleft:}\ S \rightarrow T \rightsquigarrow \lambda x{:}S^*.u}\;(\rightarrow \text{I}) \qquad \frac{\Gamma \vdash t_1 \mathrel{:\triangleright} S \rightarrow T \rightsquigarrow u \qquad \Gamma \vdash t_2 \mathrel{\triangleleft:}\ S \rightsquigarrow u'}{\Gamma \vdash t_1\ t_2 \mathrel{:\triangleright} T \rightsquigarrow u\ u'}\;(\rightarrow \text{E})$$

$$\frac{y \text{ fresh} \qquad \Gamma,\ y : S \vdash t \mathrel{\triangleleft:}\ T \rightsquigarrow u}{\Gamma \vdash t \mathrel{\triangleleft:}\ S\, ?\!\!\rightarrow T \rightsquigarrow \lambda y{:}S^*.u}\;(?\!\!\rightarrow \text{I}) \qquad \frac{\Gamma \vdash t \mathrel{:\triangleright} S\, ?\!\!\rightarrow T \rightsquigarrow u \qquad \Gamma \vdash ? \mathrel{\triangleleft:}\ S \rightsquigarrow u'}{\Gamma \vdash t \mathrel{:\triangleright} T \rightsquigarrow u\ u'}\;(?\!\!\rightarrow \text{E})$$

$$\frac{\Gamma,\ X \vdash t \mathrel{\triangleleft:}\ T \rightsquigarrow u}{\Gamma \vdash t \mathrel{\triangleleft:}\ \forall X.T \rightsquigarrow \Lambda X.u}\;(\forall\ \text{I}) \qquad \frac{\Gamma \vdash t \mathrel{:\triangleright} \forall X.T \rightsquigarrow u}{\Gamma \vdash t \mathrel{:\triangleright} [X := S]T \rightsquigarrow u\ [S^*]}\;(\forall\ \text{E})$$

$$\frac{\Gamma \vdash t \mathrel{\triangleleft:}\ T \rightsquigarrow u \qquad \Gamma, x : T \vdash s \mathrel{:\triangleright} R \rightsquigarrow u'}{\Gamma \vdash \textbf{let}\ x : T = t\ \textbf{in}\ s \mathrel{:\triangleright} R \rightsquigarrow (\lambda x{:}T^*.u')\ u}\;(\textsc{Let-Ex}) \qquad \frac{\Gamma \vdash t \mathrel{\triangleleft:}\ T \rightsquigarrow u \qquad y \text{ fresh} \qquad \Gamma, y : T \vdash s \mathrel{:\triangleright} R \rightsquigarrow u'}{\Gamma \vdash \textbf{let}\ ? : T = t\ \textbf{in}\ s \mathrel{:\triangleright} R \rightsquigarrow (\lambda y{:}T^*.u')\ u}\;(\textsc{Let-Im})$$

$$\frac{\Gamma \vdash t \mathrel{:\triangleright} R \rightsquigarrow u}{\Gamma \vdash t \mathrel{\triangleleft:}\ R \rightsquigarrow u}\;(\textsc{Stitch})$$

Fig. 3. Type-directed translation from SI to System F

3.4 Examples

To demonstrate the expressiveness of the calculus, we present two Scala examples, both of which can be expressed in our calculus with only minor changes in notation. Instead of the query operator ? we write `implicitly` and instead of the `let` constructs of the calculus we use `def`s. Since Scala does not allow `def`s 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. Both examples are accepted by the Dotty compiler.

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
  }

  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
```

3.5 Disambiguation

The calculus presented in Figure 2 is ambiguous in two aspects. First, rule (∀ 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 may be 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 ∀ 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 (∀ E) and (∀ 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].

We write $size(\mathsf{T})$ for the standard size measure on type expressions T, i.e. $size(\mathsf{T}) = 1 + \sum size(\mathsf{T}_i)$, with $\{\mathsf{T}_i\}$ the (possibly empty) set of subexpressions of T.

Definition 3.2. Extend $size(\mathsf{T})$ to judgements as follows:

$$size(\Gamma \vdash \mathsf{t} \lhd\!: \mathsf{T}) \quad = \quad size(\mathsf{T}) \tag{1}$$

$$size(\Gamma \vdash \mathsf{t} :\!\rhd \mathsf{T}) \quad = \quad -size(\mathsf{T}) \tag{2}$$

Lemma 3.3. *Let* J *be a judgement over the term* t*. Then for any other judgment* J' *over the same term* t*, and any non-syntax-directed inference rule that derives* J' *and that has* J *as its leftmost premise:* $\mathrm{size}(\mathsf{J}) < \mathrm{size}(\mathsf{J}')$.

Proof. By inspection of the typing rules □

Definition 3.4. The *resolved implicit variable* $res(\mathcal{D})$ in a derivation tree $\mathcal{D}$ ending in a judgement $\Gamma \vdash ? \lhd\!: \mathsf{T}$ or $\Gamma \vdash ? :\!\rhd \mathsf{T}$ is defined as follows (by recursion on typing derivations).

(1) If $\mathcal{D}$ consists of an application of a (Query) instance

$$\frac{\mathsf{y} : \mathsf{T} \in \Gamma}{\Gamma \vdash ?\ :\!\rhd \mathsf{T}},$$

the variable y.

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

$$\frac{\Gamma \vdash ?\ :\triangleright S \mathbin{?\!\!\longrightarrow} T \qquad \Gamma \vdash\ ? \triangleleft:\ S}{\Gamma \vdash ?\ :\triangleright T},$$

the resolved implicit variable of the derivation tree of $\Gamma \vdash ?\ :\triangleright S \mathbin{?\!\!\longrightarrow} T$.

(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 (?⟶ I) instance

$$\frac{\Gamma,\ y : S \vdash ? \triangleleft:\ T}{\Gamma \vdash ? \triangleleft:\ S \mathbin{?\!\!\longrightarrow} T},$$

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 subderivation $\mathcal{D}'$ of a resolution judgment $J = \Gamma \vdash ? \triangleleft: T$ or $J = \Gamma \vdash ?\ :\triangleright T$ in $\mathcal{D}$, and every other derivation $\mathcal{D}''$ of J, $res(\mathcal{D}') = res(\mathcal{D}'')$ or $res(\mathcal{D}'')$ is defined in Γ to the left of $res(\mathcal{D}')$.

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 is 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 (?⟶ 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).

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

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 ($?\!\!\longrightarrow$ 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 ($\longrightarrow$ E).

$$\frac{\Gamma \vdash t_1 \;:\triangleright\; S \longrightarrow T' \qquad \Gamma \vdash t_2 \;\triangleleft\!:\; S}{\Gamma \vdash t_1\ t_2 \;:\triangleright\; T'}$$

By the inductive hypothesis the typing derivations of $\Gamma \vdash t_1 \;:\triangleright\; S \longrightarrow T'$ and $\Gamma \vdash t_2 \;\triangleleft\!:\; S$ are both unique. Hence, any other derivation $\mathcal{D}_0'$ of $\Gamma \vdash t_1\ t_2 \;:\triangleright\; T'$ 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 $S_1 ?\!\!\longrightarrow \dots ?\!\!\longrightarrow S_n ?\!\!\longrightarrow R$, 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 ($?\!\!\longrightarrow$ E).

$$\frac{\Gamma \vdash t \;:\triangleright\; S_1 ?\!\!\longrightarrow (S_2 ?\!\!\longrightarrow \dots ?\!\!\longrightarrow S_n ?\!\!\longrightarrow R) \qquad \Gamma \vdash ? \;\triangleleft\!:\; S_1}{\Gamma \vdash t \;:\triangleright\; S_2 ?\!\!\longrightarrow \dots ?\!\!\longrightarrow S_n ?\!\!\longrightarrow R}$$

By the induction hypothesis, the derivation of $\Gamma \vdash t \;:\triangleright\; S_1 ?\!\!\longrightarrow (S_2 ?\!\!\longrightarrow \dots ?\!\!\longrightarrow S_n ?\!\!\longrightarrow R)$ 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 $S ?\!\!\longrightarrow T'$ the only applicable rule that ends in this judgement is ($?\!\!\longrightarrow$ 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 ($\longrightarrow$ 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 ($\longrightarrow$ 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 ($\longrightarrow$ 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. □

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.

$$
\begin{array}{lcll}
\mathit{synth}\llbracket \Gamma \vdash \mathsf{x} \rrbracket & = & \mathit{elim}\llbracket \Gamma \vdash \mathsf{x} \;:\!\triangleright\; \Gamma(\mathsf{x}) \rrbracket & \\
\mathit{synth}\llbracket \Gamma \vdash \mathsf{y} \rrbracket & = & \mathit{elim}\llbracket \Gamma \vdash \mathsf{y} \;:\!\triangleright\; \Gamma(\mathsf{y}) \rrbracket & \\
\mathit{synth}\llbracket \Gamma \vdash \mathsf{t}_1\ \mathsf{t}_2 \rrbracket & = & (\mathsf{T},\ \mathsf{t}_1'\ \mathsf{t}_2') & \textbf{if}\quad (\mathsf{S} \to \mathsf{T},\ \mathsf{t}_1') = \mathit{synth}\llbracket \Gamma \vdash \mathsf{t}_1 \rrbracket \\
 & & & \qquad \mathsf{t}_2' = \mathit{check}\llbracket \Gamma \vdash \mathsf{t}_2 \triangleleft\!:\ \mathsf{S} \rrbracket \\
 & & \mathit{error} & \textbf{otherwise} \\
\mathit{synth}\llbracket \Gamma \vdash \textbf{let}\ \mathsf{x} : \mathsf{S} = \mathsf{s}\ \textbf{in}\ \mathsf{t} \rrbracket & = & (\mathsf{T},\ (\lambda \mathsf{x} : \mathsf{S}^*.\ \mathsf{t}')\ \mathsf{s}') & \textbf{if}\quad \mathsf{s}' = \mathit{check}\llbracket \Gamma \vdash \mathsf{s} \triangleleft\!:\ \mathsf{S} \rrbracket \\
 & & & \qquad (\mathsf{T},\ \mathsf{t}') = \mathit{synth}\llbracket \Gamma, \mathsf{x} : \mathsf{S} \vdash \mathsf{t} \rrbracket \\
 & & \mathit{error} & \textbf{otherwise} \\
\mathit{synth}\llbracket \Gamma \vdash \textbf{let}\ ? : \mathsf{S} = \mathsf{s}\ \textbf{in}\ \mathsf{t} \rrbracket & = & (\mathsf{T},\ (\lambda \mathsf{y} : \mathsf{S}^*.\ \mathsf{t}')\ \mathsf{s}') & \textbf{if}\quad \mathsf{s}' = \mathit{check}\llbracket \Gamma \vdash \mathsf{s} \triangleleft\!:\ \mathsf{S} \rrbracket \\
 & & & \qquad (\mathsf{T},\ \mathsf{t}') = \mathit{synth}\llbracket \Gamma, \mathsf{y} : \mathsf{S} \vdash \mathsf{t} \rrbracket \\
 & & & \qquad \mathsf{y}\ \text{fresh} \\
 & & \mathit{error} & \textbf{otherwise} \\
 & & & \\
\mathit{elim}\llbracket \Gamma \vdash \mathsf{t} \;:\!\triangleright\; \mathsf{R} \rrbracket & = & (\mathsf{R},\ \mathsf{t}) & \\
\mathit{elim}\llbracket \Gamma \vdash \mathsf{t} \;:\!\triangleright\; \mathsf{S} \mathrel{?\!\!\to} \mathsf{T} \rrbracket & = & (\mathsf{R},\ \mathsf{t}') & \textbf{if}\quad \mathsf{s}' = \mathit{query}\llbracket \Gamma \vdash \mathsf{S} \rrbracket\ \Gamma \\
 & & & \qquad (\mathsf{R},\ \mathsf{t}') = \mathit{elim}\llbracket \Gamma \vdash \mathsf{t}\ \mathsf{s}' \;:\!\triangleright\; \mathsf{T} \rrbracket \\
 & & \mathit{error} & \textbf{otherwise} \\
 & & & \\
\mathit{query}\llbracket \Gamma \vdash \mathsf{T} \rrbracket\ () & = & \mathit{error} & \\
\mathit{query}\llbracket \Gamma \vdash \mathsf{T} \rrbracket\ (\Gamma',\ \mathsf{x} : \mathsf{S}) & = & \mathit{query}\llbracket \Gamma \vdash \mathsf{T} \rrbracket\ \Gamma' & \\
\mathit{query}\llbracket \Gamma \vdash \mathsf{T} \rrbracket\ (\Gamma',\ \mathsf{y} : \mathsf{S}) & = & \mathsf{t} & \textbf{if}\quad (\mathsf{T},\ \mathsf{t}) = \mathit{synth}\llbracket \Gamma \vdash \mathsf{y} \rrbracket \\
 & & \mathit{query}\llbracket \Gamma \vdash \mathsf{T} \rrbracket\ \Gamma' & \textbf{otherwise} \\
 & & & \\
\mathit{check}\llbracket \Gamma \vdash \lambda \mathsf{x}.\ \mathsf{t} \triangleleft\!:\ \mathsf{S} \to \mathsf{T} \rrbracket & = & \lambda \mathsf{x} : \mathsf{S}^*.\ \mathsf{t}' & \textbf{if}\quad \mathsf{t}' = \mathit{check}\llbracket \Gamma, \mathsf{x} : \mathsf{S} \vdash \mathsf{t} \triangleleft\!:\ \mathsf{T} \rrbracket \\
 & & \mathit{error} & \textbf{otherwise} \\
\mathit{check}\llbracket \Gamma \vdash \mathsf{t} \triangleleft\!:\ \mathsf{S} \mathrel{?\!\!\to} \mathsf{T} \rrbracket & = & \lambda \mathsf{y} : \mathsf{S}^*.\ \mathsf{t}' & \textbf{if}\quad \mathsf{t}' = \mathit{check}\llbracket \Gamma, \mathsf{y} : \mathsf{S} \vdash \mathsf{t} \triangleleft\!:\ \mathsf{T} \rrbracket \\
 & & & \qquad \mathsf{y}\ \text{fresh} \\
 & & \mathit{error} & \textbf{otherwise} \\
\mathit{check}\llbracket \Gamma \vdash \mathsf{t} \triangleleft\!:\ \mathsf{R} \rrbracket & = & \mathsf{t}' & \textbf{if}\quad (\mathsf{R},\ \mathsf{t}') = \mathit{synth}\llbracket \Gamma \vdash \mathsf{t} \rrbracket \\
 & & \mathit{query}\llbracket \Gamma \vdash \mathsf{R} \rrbracket\ \Gamma & \textbf{if}\quad \mathsf{t} = ? \\
 & & \mathit{error} & \textbf{otherwise}
\end{array}
$$

Fig. 4. Type checking algorithm for SI

3.6 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:

$$
\begin{array}{llcll}
\mathit{synth} & \llbracket \Gamma \vdash \mathsf{t} \rrbracket & = & (\mathsf{T}, \mathsf{t}) & \mid \mathit{error} \\
\mathit{check} & \llbracket \Gamma \vdash \mathsf{t} \triangleleft\!:\ \mathsf{T} \rrbracket & = & \mathsf{t}' & \mid \mathit{error} \\
\mathit{elim} & \llbracket \Gamma \vdash \mathsf{t} \;:\!\triangleright\; \mathsf{T} \rrbracket & = & (\mathsf{R}, \mathsf{t}) & \mid \mathit{error} \\
\mathit{query} & \llbracket \Gamma \vdash \mathsf{t} \rrbracket\ \Gamma' & = & \mathsf{t}' & \mid \mathit{error}
\end{array}
$$

The definitions of these functions are given in Figure 4. *synth* elaborates synthesis rules in Figure 3 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* *synth*$[\![\Gamma \vdash \mathsf{t}]\!] = (\mathsf{T}, \mathsf{t}')$ *then* $\Gamma \vdash \mathsf{t} \mathrel{:\!\triangleright} \mathsf{T} \rightsquigarrow \mathsf{t}'$.
(2) *If* *check*$[\![\Gamma \vdash \mathsf{t} \mathrel{\triangleleft\!:} \mathsf{T}]\!] = \mathsf{t}'$ *then* $\Gamma \vdash \mathsf{t} \mathrel{\triangleleft\!:} \mathsf{T} \rightsquigarrow \mathsf{t}'$

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 3.

The definitions given in Figure 4 do not handle query terms in the source that are in synthesis position. In other words, every query term ? must have an expected type. For source terms satisfying that condition, the definitions in Figure 4 consititute 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 for terms t without embedded queries ? in synthesis position:

Conjecture 3.9. *(Semi-Completeness)*

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

3.7 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 3.4.

$$\dfrac{\dfrac{\dfrac{\dfrac{\_\_2 : (\forall \mathtt{T}.\mathtt{Ord[T]} \mathbin{?\!\!\rightarrow} \mathtt{Ord[List[T]]}) \in \Gamma}{\Gamma \vdash \, ? \mathrel{:\!\triangleright} \forall \mathtt{T}.\mathtt{Ord[T]} \mathbin{?\!\!\rightarrow} \mathtt{Ord[List[T]]} \rightsquigarrow \_\_2}\,(\textsc{Query})}{\Gamma \vdash \, ? \mathrel{:\!\triangleright} \mathtt{Ord[Int]} \mathbin{?\!\!\rightarrow} \mathtt{Ord[List[Int]]} \rightsquigarrow \_\_2\mathtt{[Int]}}\,\forall\mathrm{E} \qquad \dfrac{\dfrac{\_\_1 : \mathtt{Ord[Int]} \in \Gamma \rightsquigarrow \_\_1}{\Gamma \vdash \, ? \mathrel{:\!\triangleright} \mathtt{Ord[Int]} \rightsquigarrow \_\_1}\,(\textsc{Query})}{\Gamma \vdash \, ? \mathrel{\triangleleft\!:} \mathtt{Ord[Int]} \rightsquigarrow \_\_1}\,(\textsc{Stitch})}{\Gamma \vdash \, ? \mathrel{:\!\triangleright} \mathtt{Ord[List[Int]]} \rightsquigarrow \_\_2\mathtt{[Int]}(\_\_1)}\,(?\!\!\rightarrow \mathrm{E})}{\Gamma \vdash \, ? \mathrel{\triangleleft\!:} \mathtt{Ord[List[Int]]} \rightsquigarrow \_\_2\mathtt{[Int]}(\_\_1)}\,(\textsc{Stitch})$$

In this derivation, each application of (Query) selects a simple variable (__1 and __2), but the end result is the more complex term __2[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 (∀E) 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 is 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 3.5 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 *Dotty*, 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 3

- If the expected type of a term is an implicit function type, an implicit closure is unconditionally generated, as prescribed by rule [?→ I] in Figure 3,
- If the synthesized type of a term is an implicit function type, the term is immediately applied to implicit arguments, as prescribed by rule [?→ 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[1] 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.

[1]A companion object is a an object associated with a class or a trait of the same name.

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, redendering the calculus ambiguous. Adding the well-scopedness condition means that the rightmost (i.e. innermost) matching definition is chosen as is described in Figure 4. The full-language also uses nesting to disambiguate but combines this with specifity. 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 is 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.

In the example below, we demonstrate infinite expansion during implicit search. The Dotty compiler has some heuristics to break infinite expansion after five recursion steps. The following code is valid according to the declarative type system we present in the paper. Specifically, in the compiler type checks because we provide the implicit value x (that corresponds to six expansions). On the contrary, type checking in the algorithm does not terminate because it is not equipped with similar heuristics.

```
object Outer {
    class C[T]
    implicit val x : C[C[C[C[C[C[String]]]]]] = ???
    object Inner {
        implicit def fGen[T](implicit ev: T): C[T] = ???
        implicit def fString(implicit ev: C[Int]): C[String] = ???
        implicit def fInt: C[Int] = ???

        implicitly[C[String]]
        implicitly[C[C[String]]]
        ...
        // error if the compiler didn't have any termination checks (like in the algorithm)
        implicitly[C[C[C[C[C[C[C[String]]]]]]]]
    }
}
```

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.explicitely` 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`.

```
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
```

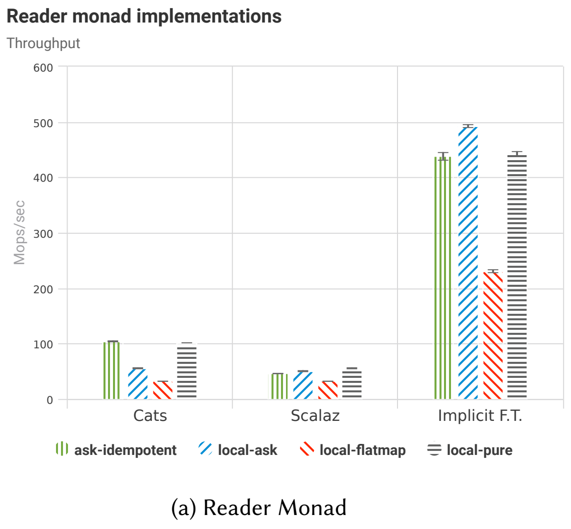
`Env` values can be obtained using `implicitly[Env]` in the body of `ReaderIFT` expression, which corresponds to the `ask` operation on `Reader`. Analogously, a new `Env` can be defined for all sub-expressions via by defining a local implicit of that type.

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.

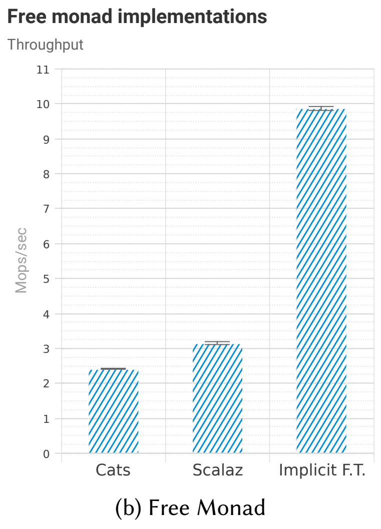
5.3.1 Performance Evaluation. We compare the implicit function type encoding of the reader monad with implementations from two widely-used Scala libraries: Scalaz 7.2 [Scalaz 2017] and Cats 0.9 [Cats 2017]. Scalaz and Cats are general-purpose, functional programming libraries that provide definitions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

We assess the performance of the aforementioned implementations using the monad reader laws:[2]

[2]The 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. Although all the experiments were run with scalac the IFT examples



(a) Reader Monad



(b) Free Monad

Fig. 5. JVM micro-benchmarks measuring throughput of various implementations in Mops/sec. Average of 10 runs, error bars for 99.9% confidence intervals.

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

Four micro-benchmarks comprise our evaluation set; each one represents a reader monad law[3] instantiated with a `String` environment and identity functions.

Results. Figure 5a 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 Structures for Free

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]
```

were written for Dotty and originally benchmarked, compiled with dotc. However, at the time of writing, the cats and scalaz libraries were not Dotty-ready. Consequently, we decided to use only scalac and the ITF examples were manually desugarded into Scala. We present numbers obtained using a single compiler for benchmarking fairness.

[3]https://github.com/typelevel/cats/blob/v0.9.0/laws/src/main/scala/cats/laws/MonadReaderLaws.scala

```
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]
}
```

Interpretation of `Free` expressions is done thought 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 `[X] => Free[A, X]`: [4]

```
def initialize[A[_]] = new Natural[A, [X] => Free[A, X]] {
  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 encoding[5]:

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

def finalize[A[_]] = new Natural[A, [X] => Expr[A, X]] {
  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 5b shows the throughput for creating and interpreting of a simple expression using different free monad implementations. The Scalaz implementation closely follows the pattern 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.

[4]The `[X] => Free[A, X]` syntax was introduced in the Dotty compiler to express type lambda.
[5]Language support for polymorphic functions would remove the need for a `Expr` trait.

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.[6]

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.

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 limitation that implicit arguments are not first-class citizens as described in 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] are a proposed 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 provides implicit arguments where function parameters can be either implicit or they can be *maximally inserted*, or even both [Coq Team 2017, Section 2.7]. The distinction comes into play when a partially applied function expects another argument which is defined as implicit. If the corresponding flag is enabled we say that the implicit value is maximally inserted. Scala is always maximally inserting implicits. However, if the expected type is an implicit function type, the insertion may be cancelled by auto-expansion (typing rule $?\!\longrightarrow \mathrm{I}$).

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

[6]O. 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

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.

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 this explicit type query loses the essential appeal of implicit programming. In this regard, our calculus is closer to Scala implicits.

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, Rouvoet [2016] presents a formal development of implicits, based on 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 word 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.

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 count[7]. 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 val`s, `implicit def`s, `implicit object`s, and implicit parameters.

Results. Some of the more interesting insights are shown in Figure 6. 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 val`s. 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.

[7]Stars 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.

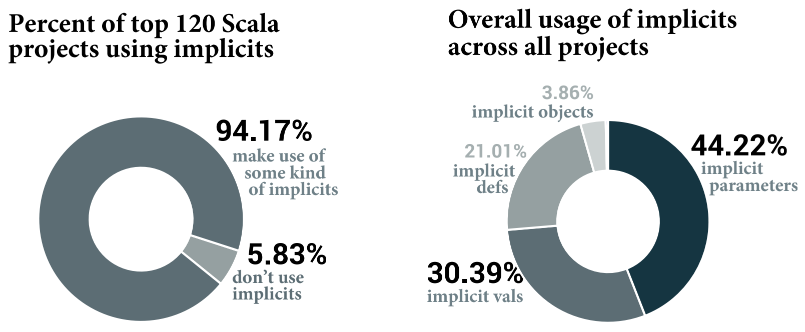


Fig. 6. Usage of implicits in the Scala ecosystem.

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

Project	Stars				
apache/spark	14905	laurilehmijoki/s3_website	1916	outworkers/phantom	939
apache/incubator-predictionio	10623	MojoJolo/textteaser	1897	sangria-graphql/sangria	937
shadowsocks/shadowsocks-android	10066	twitter/summingbird	1895	japgolly/scalajs-react	930
playframework/playframework	9846	spark-jobserver/spark-jobserver	1798	filodb/FiloDB	927
scala/scala	9117	twitter/finatra	1665	sbt/sbt-native-packager	925
akka/akka	7786	lagom/lagom	1614	functional-streams-for-scala/fs2	915
gitbucket/gitbucket	6634	twitter/algebird	1576	monix/monix	910
twitter/finagle	6047	Netflix/atlas	1553	vkostyukov/scalacaster	909
lhartikk/ArnoldC	5104	mesos/spark	1461	twitter/cassovary	906
yahoo/kafka-manager	4312	GravityLabs/goose	1455	sryza/aas	904
airbnb/aerosolve	4169	lihaoyi/Ammonite	1452	kamon-io/Kamon	895
mesos/chronos	3844	lw-lin/CoolplaySpark	1336	http4s/http4s	881
twitter/snowflake	3753	datastax/spark-cassandra-connector	1299	akka/reactive-kafka	871
snowplow/snowplow	3658	PkmX/lcamera	1297	lihaoyi/Metascala	862
rtyley/bfg-repo-cleaner	3536	rickynils/scalacheck	1285	julien-truffaut/Monocle	846
ornicar/lila	3493	twitter/iago	1252	jrudolph/sbt-dependency-graph	844
mesosphere/marathon	3483	ensime/ensime-server	1243	sryza/spark-timeseries	843
fpinscala/fpinscala	3423	foundweekends/giter8	1241	scala/pickling	823
sbt/sbt	3312	non/spire	1239	eligosource/eventsourced	821
scalaz/scalaz	3305	guardian/grid	1226	scala/async	818
gatling/gatling	3263	jaliss/securesocial	1216	scalikejdbc/scalikejdbc	817
scala-js/scala-js	3150	scala-exercises/scala-exercises	1196	lihaoyi/scala.rx	816
twitter-archive/flockdb	3110	quantifind/KafkaOffsetMonitor	1152	databricks/spark-csv	814
scala-native/scala-native	3049	finagle/finch	1111	twitter/ostrich	790
twitter/scalding	2903	mauricio/postgresql-async	1110	twitter/twitter-server	765
twitter/diffy	2895	lift/framework	1110	adamw/macwire	754
linkerd/linkerd	2867	ThoughtWorksInc/Binding.scala	1101	adamw/elasticmq	736
twitter-archive/kestrel	2790	mpeltonen/sbt-idea	1092	wartremover/wartremover	735
spray/spray	2532	circe/circe	1089	ReactiveMongo/ReactiveMongo	734
milessabin/shapeless	2262	coursier/coursier	1055	nscala-time/nscala-time	731
scalanlp/breeze	2262	getquill/quill	1042	playframework/play-slick	724
scalatra/scalatra	2227	killrweather/killrweather	1040	ReactiveX/RxScala	718
twitter-archive/gizzard	2143	sksamuel/elastic4s	1029	JetBrains/intellij-scala	710
typelevel/cats	2115	tumblr/colossus	1011	jdegoes/blueeyes	704
pocorall/scaloid	2099	json4s/json4s	971	etaty/rediscala	702
intel-analytics/BigDL	2089	paypal/squbs	968	scalaj/scalaj-http	699
apache/incubator-openwhisk	2073	amplab/shark	968	unfiltered/unfiltered	688
lampepfl/dotty	1969	tpolecat/doobie	961	typesafehub/sbteclipse	677
twitter/util	1944	yahoojapan/objc2swift	949	miguno/kafka-storm-starter	675
slick/slick	1926	pathikrit/better-files	943	spray/spray-json	674

ACKNOWLEDGMENTS

We thank the anonymous reviewers of both the program committee and the artifact evaluation committee for their constructive comments. We gratefully acknowledge funding by the Swiss

National Science Foundation under grants 200021_166154 (`Effects as Implicit Capabilities`) and 407540_167213 (`Programming Language Abstractions for Big Data`). We thank Oleg Kiselyov and Arjen Rouvoet for discussions we had. The conference management example in the paper was inspired by a talk of Nadia Polikarpova. Tiark Rompf first proposed and implemented the second search space truncation technique described in Section 4.4.

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).

Coq Team. 2017. The Coq Proof Assistant Reference Manual, Version 8.7.0. (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 fonctionelle 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.

Oleg Kiselyov and Chung-chieh Shan. 2004. Functional Pearl: Implicit Configurations–or, Type Classes Reflect the Values of Types. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell '04).* ACM, New York, NY, USA, 33–44. DOI:http://dx.doi.org/10.1145/1017472.1017481

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.

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.