# Foundations of Implicit Function Types

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

Implicit parameters are used pervasively in Scala and are also present in a number of other programming and theorem proving languages. This paper describes a generalization of implicit parameters as they are currently found in Scala to *implicit function types*. We motivate the construct by a series of examples and provide formal foundations that closely follow the semantics implemented by the Scala compiler.

## 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* [16]) or external to it (e.g., Guice [23], Spring [8, 9], MacWire [26]).

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 and straightforward, but sometimes it's too much of a good thing. Programs can quickly become riddled with long parameter lists. A number of programming techniques have been invented to combat that problem. Currying and point-free style, or the reader monad [10], are some examples. Nevertheless, it's fair to say that the flexible composition of components with many functional dependencies remains challenging.

A straightforward way of dealing with the problem of having too many parameters is to make some of them *implicit*. Arguments to implicit parameters are synthesized according to the type of the parameter. Implicit parameters were first invented in Haskell [12]. They are widely used in Scala, where they model type classes [25], configurations, capabilities, type constraints [7] and many other forms of contextual abstractions. Implicit parameters have also been introduced to Agda [6] and to OCaml [27].

Implicit parameters in Scala avoid the tedium of having to pass many parameters to many functions. But they don't eliminate all repetition as they still have to be declared in every function that uses them. For instance, the *Dotty* compiler [14] 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. Implicit function types also appear under the name *instance arguments* in Agda [6, 13].

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 [18, 19] is motivated by examples from

2017. XXXX-XXXX/2017/1-ART1 $0
DOI: 10.1145/nnnnnnn.nnnnnnn

both Scala's implicits and Haskell's type classes. It comes close to expressing Scala's implemented semantics, but there are also important differences. Some differences are syntactic - their calculus demands queries of implicit values to be written explicitly, whereas in Scala implicit values are passed automatically as arguments to implicit functions. Others are semantic. For instance, the implicit calculus allows for the automatic inference of chains of implicits, which makes implicit resolution much more powerful, but also less predictable and more expensive than what Scala implements.

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 quite a bit 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 is done, the rest is regular type checking and inference, no separate calculus or algorithm for implicit resolution is needed.

*Organization.* In this paper we introduce a motivating example for implicit function types (Section 2). Next, we introduce a formalization of implicit function types based on System F (Section 3). Then, we navigate, through various different applications, to a range of new coding patterns available to the programmers (Section 4). In the end, we discuss the related work (Section 5).

## 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 compiler can automatically discover a required value and provide it, if available. We can call methods with implicit parameters as usual. What is required is the method to have an *implicit parameter list* which is identified by the keyword `implicit`.

When we call a method with implicits, the compiler looks, at the point of the call, for any implicit definitions or other implicit parameters that can 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 that the compiler performs is called *implicit resolution*. There are two basic groups of rules that the resolution algorithm applies: 1) it looks for implicits in the current scope and 2) in all associated types (e.g., companion objects and outer objects in the presence of nested types). In the previous example, the implicit value is defined in the current scope and since it is an `Int`, the compiler resolves the method call by passing `number` automatically.

Implicits are expressive enough to capture more advanced design patterns. For example, with implicits we can apply the type class-design pattern [25] in Scala. The following 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 object, providing *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)
```

The code above is quite compact as far as comparisons are concerned. In particular, it's nice that `intOrd` can be passed implicitly, not only to a simple call chain described by `comp`, but also in case we had nested calls. On the definition-side, however, every function that needs parameters to be passed implicitly, must be equipped with an additional implicit parameter, as we have seen.

Having to refactor existing code might not look so cumbersome, but it certainly leads to boilerplate code. In real-sized projects, this can get much worse. For instance, the Dotty compiler [14] uses implicit abstraction over contexts for most of its parts. Consequently it ends up with no fewer than 2641 occurrences of this particular implicit definition: `(implicit ctx: Context)`.

## 2.1 Introducing Implicit Function Types

What we can do currently, as a small refactoring step, is to move the implicit value to the right of the equals sign:

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

However, the return type of `comp` is now `Ord[T] => Boolean`, so the implicit nature of the input is not reflected anymore.

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

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

We are now able to give a proper type to the `comp` function:

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

Additionally, we can now alias the type giving a descriptive name to the concept that it represents:

```
type Ordered[T] = implicit Ord[T] => Boolean

def comp[T](x: T, y: T): Ordered[T] = {
  implicitly[Ord[T]].compare(x, y)
}
```

Observing the examples closely, we have also eliminated the need to use names for implicit parameters. Parameter names, are now auto-generated by the compiler and we can use `implicitly` to refer to them. `implicitly` is already defined in the standard library as the implicit identity function:

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

It simply looks in the implicit scope for a value of type `T` and returns it.

## 3 FORMALIZATION

We formalize implicit function types in the system SI, which is an extension of System F with implicit function types. The formulation is unique compared to other implicit calculi [18, 19]:

(1) It supports automatic application of implicit parameters without explicit query.
(2) It supports automatic expansion of expressions into implicit lambdas.
(3) It unifies type checking with implicit resolution.

### 3.1 Syntax

| | | | | | |
|---|---|---|---|---|---|
| t = | | Term | R = | | Restricted type |
| | x | variable | | X | type variable |
| | y | implicit variable | | T → T | function type |
| | λx.t | function | | | |
| | t t | application | T = | | Full type |
| | ? | implicit query | | R | restricted type |
| | let x : T = t in t | explicit let | | T ?→ 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). The separation saves us the effort to maintain two different environments in the typing rules. 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.

Lambdas don't have type annotations on parameters, which will be inferred from the type checking context. There are no lambdas for implicit functions, as they can be synthesized from types in semantic translation. The same holds true for type lambdas.

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

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

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

$$\frac{\Gamma,\; X \vdash t \triangleleft\!: \; T}{\Gamma \vdash t \triangleleft\!: \; \forall X.T} \;(\forall\, \mathrm{I}) \qquad \frac{\Gamma \vdash t \;:\triangleright \forall X.T}{\Gamma \vdash t \;:\triangleright [X := T']T} \;(\forall\, \mathrm{E})$$

$$\frac{\Gamma \vdash t \triangleleft\!: \; T \qquad \Gamma, x : T \vdash t' \;:\triangleright T'}{\Gamma \vdash \mathrm{let}\; x : T = t \;\mathrm{in}\; t' \;:\triangleright T'} \;(\textsc{Let-Ex}) \qquad \frac{\Gamma \vdash t \triangleleft\!: \; T \qquad \Gamma, y : T \vdash t' \;:\triangleright T'}{\Gamma \vdash \mathrm{let}\; ? : T = t \;\mathrm{in}\; t' \;:\triangleright T'} \;(\textsc{Let-Im})$$

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

Fig. 2. SI Typing Rules

In the calculus, implicit parameters get resolved and applied automatically. The query operator (denoted as ?) is for accessing implicit parameters of implicit functions (which will be synthesized in semantic translation). Programmers can also use the query operator to trigger implicit resolution explicitly – this corresponds then to the use of `implicitly` in Scala.

The implicit let-binding allows us to introduce implicit variables. The explicit let-binding is designed to be the dual of the implicit let-binding. The let-bindings introduce expected types, so that we can omit type annotations on lambdas.

For types, in addition to the normal function type, we have implicit function types ($T \,?\!\!\to T$). The types are divided into restricted types and non-restricted types. Restricted types consist of normal function types and type variables. Restricted types is useful in the typing rules as we will discuss below.

### 3.2 Type System

The type system of SI is presented in Figure 2. The type system is based on *bidirectional type checking*. The judgement form $\Gamma \vdash t \;:\triangleright T$ means that the term t gets a *synthesized* type T under the context $\Gamma$, while $\Gamma \vdash t \triangleleft\!: \; T$ means that the term t is *checked* to be compatible with the expected type T under the context $\Gamma$.

Rule (Var) is standard. Its implicit analogue (Query) can be thought as implicit resolution for the query ?. This is a synthesis rule, it doesn't specify which implicit variable to choose from the environment. We will discuss this ambiguity in Section 3.5.

The typing rule ($\to$ I) is standard. The only difference is that in our system, the type annotation for the lambda parameter can be inferred from the context, thus can be omitted.

The typing rule ($\rightarrow$ E) is also standard. Note that in the typing of the argument $t_2$, we use the checking judgement, as the type S is already known from the type $S \rightarrow T$ of the function $t_1$.

The typing rule ( $?\!\!\rightarrow$ I) type checks an expression by assuming its expansion to an implicit function, but only if the expected type of the expression is of an implicit function type.

That's why we don't have a syntax for implicit functions: if we did have a syntax for implicit functions, this rule would be applied endlessly! While it's a trivial check in the compiler to stop applying this rule if the term is already an implicit function, it's tricky to specify precedence of rules in the calculus.

The typing rule ( $?\!\!\rightarrow$ E) is where automatic resolution of implicits happens. The rule says that if a term t is of the implicit function type $S \;?\!\!\rightarrow T$ and implicit resolution can synthesize a term of the type S, then we can type the term with the type T. The resolution of the implicit parameter could 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 rule ($\forall$ I) and ($\forall$ E) deal with type lambdas. There is no explicit syntax for type lambdas in order to avoid verbosity in type abstraction and application. The rule ($\forall$ E) doesn't specify how to pick the type $T'$. In compiler implementations, this is handled by type inference.

The typing rule (Let-Ex) type checks explicit let-bindings, while the rule (Let-Im) type checks implicit let-bindings. Note that the let-bindings here have nothing to do with let-polymorphism. Let-bindings are the only place programmers give type annotations, so that they can omit type annotations in lambdas. We could follow System F and require type annotations on lambdas to get rid of the explicit let-bindings. However, given than implicit let-bindings are needed anyway to avoid introducing syntax for implicit functions – which would be at odds with the rule ( $?\!\!\rightarrow$ I) – we prefer to have explicit let-bindings as the dual of implicit let-bindings for aesthetic reasons.

The rule (Stitch) says that in order to check that a term t can take the type R, it suffices to show that the synthesized type for t is R. This is the switch where we go 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).

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 $T \;?\!\!\rightarrow T'$, create an implicit closure by entering in the environment an anonymous implicit value and proceed type checking t with $T'$ as expected type.
(2) If the expected type is a restricted type R, type check t. If that succeeds with type T, post-process T in step (3).
(3) If the type of t is an implicit function type $T \;?\!\!\rightarrow T'$, perform an implicit search for T. If a unique term $t'$ is found that matches T, continue by type checking $t\ t'$.

Compared to the implicit calculi proposed in the literature [18, 19], the system SI is the closest modeling of Scala implicits. It presents several advantages over existing implicit calculi.

First, the calculus supports automatic application of implicit parameters. The implicit calculi in the literature usually require an explicit query operator like ?T to trigger implicit resolution for the type T. In contrast, in our calculus a term t of the type $S \;?\!\!\rightarrow T$ will trigger implicit resolution automatically. This is the beauty of implicits and the reason why they are so popular in Scala and other languages. Previous calculi all fall short in this regard.

Second, the calculus unifies type checking and implicit resolution. The implicit calculi in the literature usually require separate rules for implicit resolution. The unification of type checking and implicit resolution results in a simpler system, which is also closer to the compiler implementation.

Third, the calculus automatically expands expressions of implicit function types to implicit functions during semantic translation. No syntax for implicit functions is needed. It reduces boilerplate in using implicit function types, which is a big advantage over existing calculi in the literature.

### 3.3 Type Checking Examples

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

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

f : Boolean
```

In the code above, read the type `implicit S => T` as $S \; ?\!\!\to T$. The implicit variable definition `implicit val f: T = ...` is equivalent to `let ?:T = ...`, and `f : Boolean` is equivalent to `let x:Boolean = f in x` in the calculus.

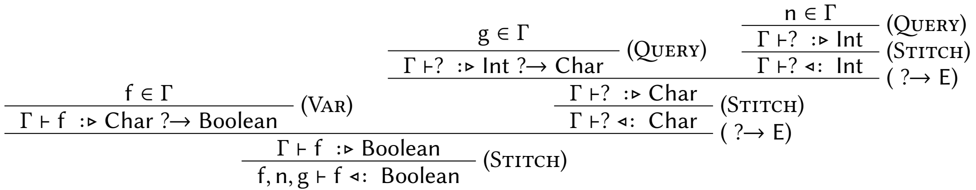

Fig. 3. Type Derivation for `f: Boolean`

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

Here's is another example which covers the rule ( ?→ I):

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

h : Boolean
```

The type derivation for type checking h with the expected type Boolean is given in Figure 4. An essential difference compared to the previous example is that in the resolution of the implicit parameter for h (with the type Int ?→ Char), we cannot apply the rule (Stitch) as it only works for restricted types, not implicit function types. The only possible choice here is to use the rule

( ?→ I), which does type checking by assuming expansion of expressions into implicit functions. The oracle rule (Query) is also used twice to pick the right implicits g and y from the environment. Note that the oracle prefers inner-most implicit in the environment, thus y is chosen instead of n.

$$
\dfrac{
  \dfrac{
    \dfrac{h \in \Gamma}{\Gamma \vdash h \ :\triangleright (\mathsf{Int} \ ?\!\!\to \mathsf{Char}) \ ?\!\!\to \mathsf{Boolean}}\ (\textsc{Var})
    \qquad
    \dfrac{
      \dfrac{
        \dfrac{g \in \Gamma'}{\Gamma' \vdash ? \ :\triangleright \mathsf{Int} \ ?\!\!\to \mathsf{Char}}\ (\textsc{Query})
        \qquad
        \dfrac{\dfrac{y \in \Gamma'}{\Gamma' \vdash ? \ :\triangleright \mathsf{Int}}\ (\textsc{Query})}{\Gamma' \vdash ? \triangleleft: \ \mathsf{Int}}\ (\textsc{Stitch})
      }{\Gamma, y{:}\mathsf{Int} \vdash ? \triangleleft: \ \mathsf{Char}}\ (\ ?\!\!\to \mathrm{E})
    }{\Gamma \vdash ? \triangleleft: \ \mathsf{Int} \ ?\!\!\to \mathsf{Char}}\ (\ ?\!\!\to \mathrm{I})
  }{\Gamma \vdash h \ :\triangleright \mathsf{Boolean}}\ (\ ?\!\!\to \mathrm{E})
}{h, n, g \vdash h \triangleleft: \ \mathsf{Boolean}}\ (\textsc{Stitch})
$$

Fig. 4. Type Derivation for `h: Boolean`

### 3.4 Translation to System F

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

The translation of types is given below:

$$
\begin{array}{lcl}
(S \ ?\!\!\to T)^* & = & S^* \to T^* \\
(S \to T)^* & = & S^* \to T^* \\
(\forall X.T)^* & = & \forall X.T^* \\
T^* & = & T, \text{otherwise}
\end{array}
$$

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:

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

The translation is presented in Figure 5. Note that we don't translate implicit variables and assume that the target language treats implicit variables and explicit variables the same way. The translation rules are straight-forward, thus we omit the explanation.

Theorem 3.1 (Type-preserving Translation). *Let* t *be a SI term of type* T*, and* t′ *be an System F term. If* $\varnothing \vdash t : T \rightsquigarrow t'$*, then* $\varnothing \vdash t' : T^*$.

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

Now the type safety of SI can be expressed in terms of type safety of System F. We define the dynamic semantics of SI as the following:

$$\mathrm{eval}(t) = v \text{ where } \varnothing \vdash t : T \rightsquigarrow t' \text{ and } t' \to^* v$$

In the above, $\to^*$ is the reflexive, transitive closure of System F's standard small-step call-by-value reduction relation.

The type safety of SI is stated by the following theorem:

Theorem 3.2 (Type Safety). *If* $\varnothing \vdash t : T$*, then* eval(t) = v *for some System F value* v.

Proof. Immediate from normalization of System F and type-preservation translation. □

$$\frac{x : T \in \Gamma}{\Gamma \vdash x \;:\triangleright T \rightsquigarrow x} \quad \text{(Var-Ex)}$$

$$\frac{y : T \in \Gamma}{\Gamma \vdash ? \;:\triangleright T \rightsquigarrow y} \quad \text{(Query)}$$

$$\frac{\Gamma,\; x : S \vdash t \;\triangleleft:\; T \rightsquigarrow u}{\Gamma \vdash \lambda x.t \;\triangleleft:\; S \to T \rightsquigarrow \lambda x{:}S^*.u} \quad (\to \mathrm{I})$$

$$\frac{\Gamma \vdash t_1 \;:\triangleright S \to T \rightsquigarrow u \qquad \Gamma \vdash t_2 \;\triangleleft:\; S \rightsquigarrow u'}{\Gamma \vdash t_1\; t_2 \;:\triangleright T \rightsquigarrow u\; u'} \quad (\to \mathrm{E})$$

$$\frac{y \text{ fresh} \qquad \Gamma,\; y : S \vdash t \;\triangleleft:\; T \rightsquigarrow u}{\Gamma \vdash t \;\triangleleft:\; S \;?\!\!\to T \rightsquigarrow \lambda y{:}S^*.u} \quad (\;?\!\!\to \mathrm{I})$$

$$\frac{\Gamma \vdash t \;:\triangleright S \;?\!\!\to T \rightsquigarrow u \qquad \Gamma \vdash ? \;\triangleleft:\; S \rightsquigarrow u'}{\Gamma \vdash t \;:\triangleright T \rightsquigarrow u\; u'} \quad (\;?\!\!\to \mathrm{E})$$

$$\frac{\Gamma,\; X \vdash t \;\triangleleft:\; T \rightsquigarrow u}{\Gamma \vdash t \;\triangleleft:\; \forall X.T \rightsquigarrow \Lambda X.u} \quad (\forall\, \mathrm{I})$$

$$\frac{\Gamma \vdash t \;:\triangleright \forall X.T \rightsquigarrow u}{\Gamma \vdash t \;:\triangleright [X := T']T \rightsquigarrow u\; [T'^*]} \quad (\forall\, \mathrm{E})$$

$$\frac{\Gamma \vdash t \;\triangleleft:\; T \rightsquigarrow u \qquad \Gamma, x : T \vdash t' \;:\triangleright T' \rightsquigarrow u'}{\Gamma \vdash \text{let } x : T = t \text{ in } t' \;:\triangleright T' \rightsquigarrow (\lambda x{:}T^*.u')\; u} \quad \text{(Let-Ex)}$$

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

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

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

### 3.5 Ambiguity

The calculus presented in Figure 2 is ambiguous. The problem is that the rule (Query) doesn't specify how to pick an implicit variable from the environment. By the rule itself, any implicit variable in the environment qualifies, thus it's ambiguous.

The Scala compiler contains rules for disambiguating based on specificity and nesting. Generally, an implicit candidate A is as good as B if it wins or draws in a tournament that awards one point for each of the following comparisons:

(1) A is nested more deeply than B.
(2) The nesting levels of A and B are the same, and A's owner derives from B's owner.
(3) A's type is more specific than B's type.

The nesting rule can be expressed in the language of the calculus with the definition of *well-scopedness.*

*Definition 3.3.* A typing derivation D is well-scoped, if for every subgoal $\Gamma \vdash ? : R \rightsquigarrow y\; \overline{t}$ in D, if $\Gamma \vdash ? : R \rightsquigarrow y'\; \overline{t}'$ is derivable, then $y = y'$ or $y'$ is bound in $\Gamma$ to the left of y.

In a sense, well-scoped typing derivations always pick the rightmost (innermost) eligible implicit. We can show that the following holds for the monomorphic system without polymorphic function types:

Proposition 3.4. *Given* $\Gamma$ *and* t*, there is at most one restricted type* R *and one term* u *such that* $\Gamma \vdash$ t $:\triangleright$ R $\rightsquigarrow$ u *by a well-scoped typing derivation.*

Proposition 3.5. *Given* $\Gamma$, t *and* T*, there is at most one term* u *such that* $\Gamma \vdash$ t $\triangleleft:$ T $\rightsquigarrow$ u *by a well-scoped typing derivation.*

The propositions don't 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.

There are some deep questions to cover here, in particular, the decision which type variables should be instantiated before doing an implicit search and which type variables should be left uninstantiated, so that they can be further constrained by the implicit resolution. Instantiating type variables too early risks cutting off parts of the solution space which might result in failures to find a matching implicit instance. On the other hand , instantiating type variables too late might lead to ambiguity errors. All of this is for future work, however.

### 3.6 Examples

Next, we present two examples that apart from using syntax equivalent to the notation of the calculus they are also verified by the Dotty compiler. The implicit function type S ?→ T is written `implicit S => T` in Scala. Instead of the query operator ? we write `implicitly` and instead of the `let` constructs of the calculus we use `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.

*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]`. Since higher-order implicits are modeled by our calculus, they can be implemented easily in SI and Scala.

```
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
```

## 4 APPLICATIONS USING IMPLICIT FUNCTION TYPES

This section introduces three applications that demonstrate the expressive power of programming with implicit function types. Additionally we provide a slightly larger case study, introducing a new encoding of Free Monads.

### 4.1 Reader Monad: Use Contextual Abstraction

The reader monad represents computations with the ability to read from an environment and pass values retrieved from it to functions. It is defined in term of two operations, `ask`, to retrieve the environment, and `local` to modify the environment for sub-computations. A simple example is `Reader`, a trait defined as follows (we omit the definition of the monad instance):

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

Reader expressions are formed using the for-comprehension syntax of Scala (equivalent to the do-notation syntax of Haskell)):

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

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

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

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

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

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

This pattern is very common is large-scale applications. For instance, in web programming, the majority of functions takes 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.

### 4.2 Builder Pattern: Expressive DSL

Implicit function types have considerable expressive power. In this application, we demonstrate how we can support the builder pattern using the new type. The goal is to provide a type-safe, declarative API to program builders [1] without any boilerplate on the client-side. The following code presents a small DSL that offers constructs to create tables:

```
table {
  row {
    cell("top left")
    cell("top right")
  }
  row {
    cell("botttom left")
    cell("bottom right")
```

```
  }
}
```

Each keyword–table, row and cell–is a regular factory method. Every call of these functions, constructs a node (resembling to an AST) that comprises our table. For instance, the Table class contains the necessary machinery to store rows and a way to print itself.

What we need, is to update a Table instance through the table method and to propagate it in the subsequent calls. The following code shows how this dependency is represented using contextual abstraction where the context is the Table. Consequently, the argument of the table method is parameterized by an implicit function which accepts a Table, implicitly. We omit the definitions for row and cell as they follow the same philosophy.

```
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) = {
  implicit val t = new Table
  init
  t
}
```

The table construction above is translated into the following:

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

As we notice the compiler translated the builder example in method invocations passing lambdas that take the expected implicit arguments as parameters.

### 4.3 Tagless Interpreters: Abstracting over the number of constraints

Implicit function types are introducing a very useful indirection that enables us to abstract the number of implicit parameters being introduced in some scope. We demonstrate this pattern through a tagless interpreter [5] (or object algebra [17] as popularized in the OOP domain) for expressions. Tagless interpreters use a final encoding for DSL construction. They solve the extensibility problem so we can add both new syntactic elements and interpretations without breaking the existing hierarchy effectively solving the Expression Problem [24].

We present the tagless definition of a small expression language with only two constructs: lit and add. The trait Exp defines functions (and not constructors) for both constructs and is parameterized by a generic type of interpreters. We can define the semantic domain of these two functions by providing instances e.g., for integer expressions.

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

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

As a result, the example defined with function `tf1` can be used to evaluate the expression $8 + (1 + 2)$. `tf1` requires an interpreter of type `Exp` which is expressed as a *context bound* on the generic type– which is a concise form in place of an implicit parameter `implicit ev: Exp[T]`. We then proceed with the definition of multiplication for this language which is defined similarly. Evaluating an expression with multiplication with `tfm1` now requires both `Exp` and `Mul` interpreters. Both examples need the evidence for expressions of integers:

```
implicit val evalExp: Exp[Int] = new Exp[Int] {
  def lit(i: Int): Int = i
  def add(l: Int, r: Int): Int = l + r
}
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 of context bounds and implicit parameters cannot be used to abstract over the arity of such constraints. On the contrary, implicit function types can be aliased. For instance, we can group both constraints using a `Ring` type which liberates us from maintaining an explicit list of "requirements" in all use-sides. The latter methods, `tfm1` and `tfm2` demonstrate the aforementioned desirable property.

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

def tfm1[T]: Ring[T] = add(lit(7), mul(lit(1), lit(2)))
def tfm2[T]: Ring[T] = mul(lit(7), tf1)
```

### 4.4 Case Study: a New Encoding of Free Monads

This section presents a new implementation of free monads using implicit function types. The ability to factor out type class constraints in a type alias lead to an encoding that is both simpler and faster without introducing addition verbosity. Firstly, we present traditional encoding for free monads followed then, by a new implementation. We summarize this case study by comparing the two approaches.

*4.4.1 Approach with implicits: À la carte encoding.* Data types à la carte [21] popularized the free monad pattern, an abstraction to decouple monadic expressions and semantics. Typical Scala implementations [4] of this idea use a (generalized) algebraic data type to encapsulate the monadic structure:

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

The primitive operations to be used in a free monad are also defined using a generalized algebra data type which is often called an algebra:

```
sealed trait KVStoreA[T] // Key Value store Algebra
case class Put(key: String, value: Int) extends KVStoreA[Unit]
case class Get(key: String) extends KVStoreA[Option[Int]]
```

Expressions in this newly defined languages are instances of the `KVStore` type, defined as follows:

```
type KVStore[T] = Free[KVStoreA, T]
```

In order to beautify the definition of expressions, users of free monads define additional boilerplate to lift the algebra into the Free structure:

```
def put(key: String, value: Int): KVStore[Unit] =
  Suspend[KVStoreA, Unit](Put(key, value))

def get(key: String): KVStore[Option[Int]] =
  Suspend[KVStoreA, Option[Int]](Get(key))
```

Expressions are defined in `for`-comprehensions, using `KVStore` as a monad (a monad instance for `Free` is provided as part of the free monad library):

```
val alacarteExpr: KVStore[Option[Int]] =
  for {
    _ <- put("foo", 2)
    _ <- put("bar", 5)
    n <- get("foo")
  } yield n
```

Finally, interpretation is done through the definition of a natural transformation and a `foldMap` function. The later is being provided as part of the infrastructure of the free monad library:

```
def alacarteInterpreter = new Natural[KVStoreA, Future] {
  def apply[T](ft: KVStoreA[T]): Future[T] = ???
}

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

val alacarteOutput: Future[Option[Int]] = foldMap(alacarteExpr)(alacarteInterpreter)
```

*4.4.2 Approach with 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`: a function accepting a natural transformation from the algebra to a monad, returning an interpretation of the expression in the given monad:

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

The algebra can be defined as an enumeration [15]:

```
enum KVStoreB[T] {
  case Put(key: String, value: Int) extends KVStoreB[Unit]
  case Get(key: String) extends KVStoreB[Option[Int]]
}
import KVStoreB._

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

In this new encoding of free monads, expressions are function type parametric in the monad used for interpretation:

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

In this setup, `lift` is an extension method that applies a natural transformation:

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

Interpreters are, as before, natural transformations. However, the interpretation doesn't require any library infrastructure, it a simple function application of the `expr` function with an interpreter (the monad instance for `Future` is passed as an implicit parameter):

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

*4.4.3 Comparing Encodings.* The basic benefit, in this new design of free monad, is that it doesn't require any library support, (besides the orthogonal definitions of monad and natural transformations). From a user perspective, there is also absolutely no boilerplate involved.

The two encodings can be shown to be equivalent by defining a bijection between representations. Consequently, the conversion to the *À la carte encoding* is done via a interpretation of `KVStoreA` as a `Free[KVStoreA, ?]` using an interpreter (with `A = KVStoreA`): [1]

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

Conversion from the *À la carte encoding* also goes through an interpretation, from `KVStoreA` to new `Expr` trait, capturing the polymorphism in expressions by the implicit function type encoding[2] (with `A = KVStoreA`):

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

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

*4.4.4 Performance Evaluation.* 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!

*Benchmark Description.* We benchmark our free monad encoding against three other implementations taken from established Scala libraries, Scalaz 7.2 [3], Cats 0.9 [2] and Eff 4.2 [22]. Scalaz and Cats are general-purpose, functional programming libraries that provide definition of standard type classes and implementations of commonly used functional data structures, including free monads. Eff is a Scala implementation of the Freer Monads described in [11], used with a single effect/interpreter in this experiment.

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 in the simplest possible way by storing the state in mutable variables.

*Results.* Figure 6[3] 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[21]. The Cats implementation is a slight simplification over Scalaz', in that it does not require a functor instance. Eff is a Scala implementation of the Freer Monads described in [11], used with a single effect/interpreter in this experiment. The implicit function type encoding described in this section has the best performance among all implementations.

[1]The `[T] => Expr[A, T]` uses the new syntax for type lambdas implemented in the Dotty compiler.

[2]Language support for polymorphic functions would remove the need for a `Expr` trait.

[3]These measurements 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 2.11.11. We use the Java Microbenchmark Harness (JMH) [20] tool with default settings: each run averages throughput of execution over 1 second period for 20 warp-up iteration and 20 measurements iteration. The results are statistically significant.

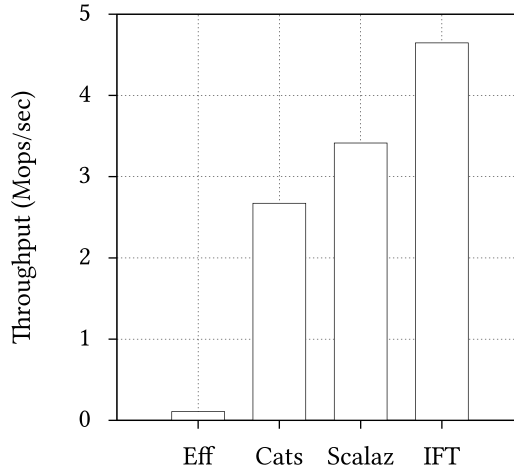


Fig. 6. Benchmark of free monad implementations

Overall, Scalaz's 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 problems. For example, the technique can be used to build Free counterparts of other type classes such as applicative functors and co-monads.

## 5 RELATED WORK

The Haskell programming language supports implicit parameters [12] 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. Several language extensions are available to generalize the use of constraints in the language. For example, the Generalized Algebraic Data Types extension allows programmers to write constraints on data type definitions.

The Glasgow Haskell Compiler also supports the `ConstraintKinds` extension, that relaxes the kind of constraints supported by the compiler. In particular, it can be used to group type classes in tuples and use this tuple as constraint. Implicit function types enable implicits (and thus type classes) to be factored out Scala, similarly to `ConstraintKinds` language extension.

Agda instance arguments [6] are closely related to Haskell's type class constraints and were inspired by both Scala's implicits and Agda's existing implicit arguments. With implicit function types, we lift the limitations that are described in Devriese et. al [6, 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.

### 5.1 Implicit Calculi

The Implicit Calculus [18] 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[19] is a recent calculus that tries to combine the strength of Haskell typeclasses and Scala implicits, i.e. the combination of ease of reasoning and flexibility. Cochis supports local implicits and meanwhile guarantees coherency. Coherency in Cochis means that substitution of equals doesn't change the semantics of programs, which is a reasonable property of pure functional programs.

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

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

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

Finally, 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.

## 6 CONCLUSION

We proposed *implicit function types* as a simple and powerful tool for dealing with contexts in programming. We formalized the concept in the system SI, which is so far the calculus that models Scala implicits most closely. Implicit function types have been implemented in Dotty, the prospective Scala compiler. We presented several applications, from expressive DSL to elegant contextual abstraction, which evidence the utility and beauty of implicit function types.

*Acknowledgements.* We thank Sandro Stucki and Arjen Rouvoet for discussions we had on the in the SI system. The conference management example in the paper was inspired by a talk of Nadia Polikarpova.

## REFERENCES

[1] 2014. Type-Safe Groovy Style Builders in Kotlin. https://web.archive.org/web/20170607150651/https://kotlinlang.org/docs/reference/type-safe-builders.html. (2014).

[2] 2017. Cats: Lightweight, modular, and extensible library for functional programming. https://web.archive.org/web/20160219022626/https://github.com/typelevel/cats. (2017).

[3] 2017. Scalaz: An extension to the core Scala library for functional programming. https://web.archive.org/web/20170305041357/https://github.com/scalaz/scalaz/. (2017).

[4] Runar Bjarnason. 2012. Stackless Scala With Free Monads. https://web.archive.org/web/20170107134129/http://blog.higher-order.com/assets/trampolines.pdf. (April 2012).

[5] 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

[6] 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.

[7] 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.

[8] Rod Johnson. 2002. *Expert One-on-One J2EE Design and Development.* Wrox Press Ltd., Birmingham, UK, UK.

[9] Rod Johnson. 2004. Spring Framework Reference Documentation. https://web.archive.org/web/20170606160435/https://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/. (2004).

[10] 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.

[11] Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. *SIGPLAN Not.* 50, 12 (Aug. 2015), 94–105. DOI: http://dx.doi.org/10.1145/2887747.2804319

[12] 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.

[13] 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).

[14] Martin Odersky. 2017. Dotty Compiler: A Next Generation Compiler for Scala. https://web.archive.org/web/20170325001401/http://dotty.epfl.ch/. (2017).

[15] Martin Odersky. 2017. Dotty Compiler: Enums. https://web.archive.org/web/20170612075027/http://dotty.epfl.ch/docs/reference/enums/enums.html. (2017).

[16] 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.

[17] 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.

[18] 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.

[19] Tom Schrijvers, Bruno C d S Oliveira, and Philip Wadler. 2017. *Cochis: Deterministic and Coherent Implicits.* Technical Report CW705. KU Leuven.

[20] Aleksey Shipilev and others. 2014. OpenJDK Code Tools: jmh. https://web.archive.org/web/20161211145153/http://openjdk.java.net/projects/code-tools/jmh/. (2014).

[21] Wouter Swierstra. 2008. Data Types à La Carte. *J. Funct. Program.* 18, 4 (July 2008), 423–436.

[22] Eric Torreborre. 2017. Eff monad for cats. https://web.archive.org/web/20170618102432/https://github.com/atnos-org/eff. (2017).

[23] Robbie Vanbrabant. 2008. *Google Guice: Agile Lightweight Dependency Injection Framework (Firstpress).* APress.

[24] 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).

[25] 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.

[26] Adam Warski. 2013. MacWire: Lightweight and Nonintrusive Scala Dependency Injection Library. https://web.archive.org/web/20170222065306/https://github.com/adamw/macwire. (2013).

[27] L. White, F. Bour, and J. Yallop. 2015. Modular implicits. *ArXiv e-prints* (Dec. 2015). arXiv:cs.PL/1512.01895