# Modular implicits

Leo White Frédéric Bour Jeremy Yallop

We present *modular implicits*, an extension to the OCaml language for ad-hoc polymorphism inspired by Scala implicits and modular type classes. Modular implicits are based on type-directed implicit module parameters, and elaborate straightforwardly into OCaml's first-class functors. Basing the design on OCaml's modules leads to a system that naturally supports many features from other languages with systematic ad-hoc overloading, including inheritance, instance constraints, constructor classes and associated types.

## 1 Introduction

A common criticism of OCaml is its lack of support for *ad-hoc polymorphism*. The classic example of this is OCaml's separate addition operators for integers (`+`) and floating-point numbers (`+.`). Another example is the need for type-specific printing functions (`print_int`, `print_string`, etc.) rather than a single `print` function which works across multiple types.

In this paper, we propose a system for ad-hoc polymorphism in OCaml based on using modules as type-directed implicit parameters. We describe the design of this system, and compare it to systems for ad-hoc polymorphism in other languages.

A prototype implementation of our proposal based on OCaml 4.02.0 has been created and is available through the OPAM package manager[1].

### 1.1 Type classes and implicits

Ad-hoc polymorphism allows the dynamic semantics of a program to be affected by the types of values in that program. A program may have more than one valid typing derivation, and which one is derived when type-checking a program is an implementation detail of the type-checker. Jones et al. [9] describe the following important property:

> Every different valid typing derivation for a program leads to a resulting program that has the same dynamic semantics.

This property is called *coherence* and is a fundamental property that must hold in a system for ad-hoc polymorphism.

#### 1.1.1 Type classes

Type classes in Haskell [18] have proved an effective mechanism for supporting ad-hoc polymorphism. Type classes provide a form of constrained polymorphism, allowing constraints to be placed on type variables. For example, the `show` function has the following type:

```
show :: Show a => a -> String
```

[1] `opam switch 4.02.0+modular-implicits`

Submitted to:
ML 2014

This indicates that the type variable `a` can only be instantiated with types which obey the constraint `Show a`. These constraints are called *type classes*. The `Show` type class is defined as[2]:

```
class Show a where
  show :: a -> String
```

which specifies a list of methods which must be provided in order for a type to meet the `Show` constraint. The method implementations for a particular type are specified by defining an *instance* of the type class. For example, the instance of `Show` for `Int` is defined as:

```
instance Show Int where
    show = showSignedInt
```

Constraints on a function's type can be inferred based on the use of other constrained functions in the function's definition. For example, if a `show_twice` function uses the `show` function:

```
show_twice x = show x ++ show x
```

then Haskell will infer that `show_twice` has type `Show a => a -> String`.

Haskell's coherence in the presence of inferred type constraints relies on type class instances being *canonical* – the program contains at most one instance of a type class for each type. For example, a Haskell program can only contain at most one instance of `Show Int`, and attempting to define two such instances will result in a compiler error. Section 4.2 describes why this property cannot hold in OCaml.

Type classes are implemented using a type-directed implicit parameter-passing mechanism. Each constraint on a type is treated as a parameter containing a dictionary of the methods of the type class. The corresponding argument is implicitly inserted by the compiler using the appropriate type class instance.

### 1.1.2 Implicits

Implicits in Scala [14] provide similar capabilities to type classes via direct support for type-directed implicit parameter passing. Parameters can be marked `implicit` which then allows them to be omitted from function calls. For example, a `show` function could be specified as:

```
def show[T](x : T)(implicit s : Showable[T]): String
```

where `Showable[T]` is a normal Scala type defined as:

```
trait Showable[T] { def show(x: T): String }
```

The `show` function can be called just like any other:

```
object IntShowable extends Showable[Int] {
  def show(x: Int) = x.toString
}

show(7)(IntShowable)
```

However, the second argument can also be elided, in which case its value is selected from the set of definitions in scope which have been marked `implicit`. For example, if the definition of `IntShowable` were marked `implicit`:

[2]Some methods of `Show` have been omitted for simplicity.

```
implicit object IntShowable extends Showable[Int] {
  def show(x: Int) = x.toString
}
```

then `show` can be called on integers without specifying the second argument – which will automatically be inserted as `IntShowable` because it has the required type `Showable[Int]`:

```
show(7)
```

Unlike constraints in Haskell, Scala's implicit parameters must always be added to a function explicitly. The need for a function to have an implicit parameter cannot be inferred from the function's definition. Without such inference, Scala's coherence can rely on the weaker property of *non-ambiguity* instead of *canonicity*. This means that you can define multiple implicit objects of type `Showable[Int]` in your program without causing an error. Instead, Scala issues an error if the resolution of an implicit parameter is ambiguous. For example, if two implicit objects of type `Showable[Int]` are in scope when `show` is applied to an `Int` then the compiler will report an ambiguity error.

### 1.1.3 Modular type classes

Dreyer et al. [4] describe *modular type classes*, a type class system which uses ML module types as type classes and ML modules as type class instances.

As with traditional type classes, type class constraints on a function can be inferred from the function's definition. Unlike traditional type classes, modular type classes cannot ensure that type class instances are canonical (see Section 4.2). Maintaining coherence in the presence of constraint inference without canonicity requires a number of undesirable restrictions, which are discussed in Section 6.5.

## 1.2 Modular implicits

Taking inspiration from modular type classes and implicits, we propose a system for ad-hoc polymorphism in OCaml based on passing implicit *module* parameters to functions based on their *module type*. By basing our system on implicits, where a function's implicit parameters must be given explicitly, we are able to avoid the undesirable restrictions of modular type classes. Fig. 1 demonstrates the `show` example written using our proposal.

The `show` function (line 6) has two parameters: an implicit module parameter `S` of module type `Show`, and an ordinary parameter `x` of type `S.t`. When `show` is applied the module parameter `S` does not need to be given explicitly. As with Scala implicits, when this parameter is elided the system will try to create a module of the appropriate type from the modules which have been made available for selection as an implicit argument.

For example, on line 24, `show` is applied to `5`. This will cause the system to try to create a module of type `Show with type t = int`. Since `ShowInt` is marked `implicit` and has the desired type, it will be used as the implicit argument of `show`.

The `ShowList` module, defined on line 18, is an *implicit functor* – note the use of the `{S : Show}` syntax for its parameter rather than `(S : Code)`. This indicates that `ShowList` can be applied to create implicit arguments, rather than used directly as an implicit argument.

For example, on line 26, `show` is applied to a list of integers. This causes the system try to create an implicit module of type `Show with type t = int list`. Such a module can be created by applying the implicit functor `ShowList` to the implicit module `ShowInt`, so `ShowList(ShowInt)` will be used as the implicit argument.

```
1   module type Show = sig
2     type t
3     val show : t -> string
4   end
5
6   let show {S : Show} x = S.show x
7
8   implicit module ShowInt = struct
9     type t = int
10    let show x = string_of_int x
11  end
12
13  implicit module ShowFloat = struct
14    type t = float
15    let show x = string_of_float x
16  end
17
18  implicit module ShowList {S : Show} = struct
19    type t = S.t list
20    let show x = string_of_list S.show x
21  end
22
23  let () =
24    print_endline ("Show an int: " ^ show 5);
25    print_endline ("Show a float: " ^ show 1.5);
26    print_endline ("Show a list of ints: " ^ show [1; 2; 3]);
```

Figure 1: ‘Show‘ using modular implicits

Fig. 2 shows another example, illustrating how a simple library for monads might look in our proposal.

The definitions of `map`, `join` and `unless` demonstrate our proposal's support for higher-kinded polymorphism, analogous to constructor classes in Haskell [7]. This is a more succinct form of higher-kinded polymorphism than is currently available in OCaml's core language. Currently, higher-kinded polymorphism is only supported directly using OCaml's verbose module language or indirectly through an encoding based on defunctionalisation [20].

The calls to `>>=` and `return` in the definitions of these functions leave the module argument implicit. These cause the system to try to create a module of the appropriate type. In each case, the implicit module parameter `M` of the function is selected because it has the appropriate type and implicit module parameters are automatically made available for selection as implicit arguments.

Like Scala's implicits, and unlike Haskell's type classes, our proposal requires all of a function's implicit module parameters to be explicitly declared. The `map` function (line 11) needs to be declared with the module parameter `{M : Monad}` – it could not be defined as follows:

```
let map m f =
  m >>= fun x -> return (f x)
```

because that would cause the system to try to resolve the implicit module arguments to `>>=` and `return` to one of the implicit modules available at the *definition* of `map`. In this case, this would result in an ambiguity error since either `MonadOption` or `MonadList` could be used.

## 1.3 Contributions

The contributions of this paper are as follows.

- We introduce *modular implicits*, a design for overloading centred around type-directed instantiation of implicit module arguments, that integrates harmoniously into a language with ML-style modules (Section 2). We show how to elaborate the extended language into standard OCaml, first by explicitly instantiating every implicit argument (Section 2.2) and then by translating functions with implicit arguments into packages (Section 2.3).
- The design of modular implicits involves only a small number of additions to the host language. However, the close integration with the existing module language means that modular implicits naturally support a rich array of features, from constructs present in the original type classes proposal such as instance constraints (Section 3.2) and subclasses (Section 3.3) to extensions to the original type class proposal such as constructor classes (Section 3.4), multi-parameter type classes (Section 3.5), associated types (Section 3.6) and backtracking (Section 3.7). Further, modular implicits support a number of features not available with type classes. For example, giving up canonicity – without losing the motivating benefit of coherence (Section 4) – makes it possible to support local instances (Section 3.8), and basing resolution on module type inclusion results in a system in which a single instance can be used with a variety of different signatures (Section 3.9).
- Both resolution of implicit arguments and type inference involve a number of subtleties related to the interdependence of resolution and inference (Section 5.1) and compositionality (Section 5.2). We describe these at a high level here, leaving a more formal treatment to future work.
- Finally, we contextualise the modular implicits design within the wide body of related work, including Haskell type classes (Section 6.1), Scala implicits (Section 6.2) canonical structures in Coq (Section 6.3), concepts in C++ (Section 6.4) and modular type classes in ML (Section 6.5).

```
1   module type Monad = sig
2     type +'a t
3     val return : 'a -> 'a t
4     val bind : 'a t -> ('a -> 'b t) -> 'b t
5   end
6
7   let return {M : Monad} x = M.return x
8
9   let (>>=) {M : Monad} m k = M.bind m k
10
11  let map {M : Monad} (m : 'a M.t) f =
12    m >>= fun x -> return (f x)
13
14  let join {M : Monad} (m : 'a M.t M.t) =
15    m >>= fun x -> x
16
17  let unless {M : Monad} p (m : unit M.t) =
18    if p then return () else m
19
20  implicit module MonadOption = struct
21    type 'a t = 'a option
22    let return x = Some x
23    let bind m k =
24      match m with
25      | None -> None
26      | Some x -> k x
27  end
28
29  implicit module MonadList = struct
30    type 'a t = 'a list
31    let return x = [x]
32    let bind m k = List.concat (List.map k m)
33  end
```

Figure 2: 'Monad' using modular implicits

# 2 The design of modular implicits

We present modular implicits as an extension to the OCaml language. The OCaml module system includes a number of features, such as first-class modules and functors, which make it straightforward to elaborate modular implicits into standard OCaml. However, the design of modular implicits is not strongly tied to OCaml, and could be integrated into similar languages in the ML family.

## 2.1 New syntax

Like several other designs for overloading based on implicit arguments, modular implicits are based on three new features. The first feature is a way to *call* overloaded functions. For example, we might wish to call an overloaded function `show`, implicitly passing a suitable value as argument, to convert an integer or a list of floating-point values to a string. The second feature is a way to *abstract* overloaded functions. For example, we might define a function `print` which calls `show` to turn a value into a string in order to send it to standard output, but which defers the choice of the implicit argument to pass to `show` to the caller of `print`. The third feature is a way to *define* values that can be used as implicit arguments to overloaded functions. For example, we might define a family of modules for building string representations for values of many different types, suitable for passing as implicit arguments to `show`.

Figure 3 shows the new syntactic forms for modular implicits, which extend the syntax of OCaml 4.02 [11].

There is one new form for types,

```
{ M : T } -> t
```

which makes it possible to declare `show` as a function with an implicit parameter `S` of module type `Show`, a second parameter of type `S.t`, and the return type `string`:

```
val show : {S: Show} -> S.t -> string
```

or to define `+` as a function with an implicit parameter `N` of module type `Num`, two further parameters of type `N.t`, and the return type `N.t`:

```
val ( + ) : {N: Num} -> N.t -> N.t -> N.t
```

There is a new kind of parameter for constructing functions with implicit arguments:

```
{ M : T }
```

The following definition of `show` illustrates the use of implicit parameters:

```
let show {S : Show} (v : S.t) = S.show v
```

The braces around the `S : Show` indicate that `S` is an implicit module parameter of type `Show`. The type `Show` of `S` is a standard OCaml module type, which might be defined as in Figure 1.

There is also a new kind of argument for calling functions with implicit arguments:

```
{ M }
```

For example, the `show` function might be called as follows using this argument syntax:

```
show {Show_int} 3
```

This is an explicitly-instantiated implicit application. Calls to `show` can also omit the first argument, leaving it to be supplied by a resolution procedure (described in Section 2.2):

```
show 3
```

Implicit application requires that the function have non-module parameters after the module parameter – implicit application is indicated by providing arguments for these later parameters without providing a module argument for the module parameter. This approach simplifies type-inference and is in keeping with how OCaml handles optional arguments. It also ensures that all function applications, which may potentially perform side-effects, are syntactically function applications.

There are two new declaration forms. Here is the first, which introduces an implicit module:

```
implicit module M ({M_i : T_i})* (: T)? = N
```

Implicit modules serve as the implicit arguments to overloaded functions like `show` and `+`. For example, here is the definition of an implicit module `Show_int` with two members: a type alias `t` and a value member `show` which uses the standard OCaml function `string_of_int`

```
implicit module Show_int = struct
  type t = int
  let show = string_of_int
end
```

Implicit modules can themselves have implicit parameters. For example, here is the definition of an implicit module `Show_list` with an implicit parameter which also satisfies the `Show` signature:

```
implicit module Show_list { A: Show } = struct
  type t = A.t list
  let show l = "["^ String.concat ", " (List.map A.show l) ^"]"
end
```

Implicit modules with implicit parameters are called *implicit functors*. Section 2.2 outlines how implicit modules are selected for use as implicit arguments.

The second new declaration form brings implicit modules into scope, making them available for use in resolution:

```
open implicit module−path
```

For example, the declaration

```
open implicit M
```

makes every implicit module bound in the module `M` available to the resolution procedure in the current scope.

There are also local versions of both declaration forms, which bind a module or bring implicits into scope within a single expression:

```
let implicit module M ({M_i : T_i})* (: T)? in expr
let open implicit module−path in expr
```

Implicit module declarations, like other OCaml declarations, bind names within modules, and so the signature language must be extended to support implicit module descriptions. There are two new forms for describing implicit modules in a signature:

```
implicit module M ({M_i : T_i})* : T
implicit module M ({M_i : T_i})* = N
```

**Types**

```
typexpr ::= ... | { module-name : module-type } -> typexpr
```

**Expressions**

```
parameter ::= ... | { module-name : module-type }

argument ::= ... | { module-expr }

expr ::= ...
| let implicit module module-name ({module-name : module-type})* (: module-type)? in expr
| let open implicit module-path in expr
```

**Bindings and declarations**

```
definition ::= ...
| implicit module module-name ({module-name : module-type})* (: module-type)? = module-expr
| open implicit module-path
```

**Signature declarations**

```
specification ::= ...
| implicit module module-name ({module-name : module-type})* : module-type
| implicit module module-name ({module-name : module-type})* = module-path
```

Figure 3: Syntax for modular implicits

The first form describes a binding for an implicit module by means of its type. For example, here is a description for the module `Show_list`:

```
implicit module Show_list {A : Show} : Show with type t = A.t list
```

The second form describes a binding for an implicit module by means of an equation [5]. For example, here is a description for a module `S`, which is equal to `Show_int`

```
implicit module module-name S = Show_int
```

## 2.2 Resolving implicit arguments

As we saw in Section 2.1, a function which accepts an implicit argument may receive that argument either implicitly or explicitly. The *resolution* process removes implicit arguments by replacing them with explicit arguments constructed from the modules in the implicit search space.

Resolving an implicit argument `M` involves two steps. The first step involves gathering constraints – that is, equations on types[3] within `M` – based on the context in which the application appears. For example, the application

```
show 5
```

should generate a constraint

[3]Constraints on module types and module aliases are also possible, but we leave them out of our treatment

```
t = int
```

on the implicit module argument passed to `show`. The second step involves searching for a module which satisfies the constrained argument type. Resolving the implicit argument for the application `show 5` involves searching for a module `S` with the type

```
Show
```

that also satisfies the constraint

```
t = int
```

The following sections consider these steps in more detail.

### 2.2.1 Generating argument constraints

Generating implicit argument constraints for an application `f x` with an implicit argument `M` of type `S` involves building a substitution which equates each type `t` in `S` with a fresh type variable `'a`, then using unification to further constrain `'a`. For example, `show` has type:

```
{S : Show} -> S.t -> string
```

and the module type `Show` contains a single type `t`. The constraint generation procedure generates the constraint

```
t = 'a
```

for the implicit parameter, and refines the remainder of the type of `show` to

```
'a -> string
```

Type-checking the application `show 5` using this type reveals that `'a` should be unified with `int`, resulting in the following constraint for the implicit parameter:

```
t = int
```

Generating implicit argument constraints for higher-kinded types involves some additional subtleties compared to generating constraints for basic types. With higher-kinded types, type constructors cannot be directly replaced by a type variable, since OCaml does not support higher-kinded type variables. Instead, each application of a parameterised type constructor must be replaced by a separate type variable.

For example, the `map` function has the following type:

```
{M : Monad} -> 'a M.t -> ('a -> 'b) -> 'b M.t
```

After substituting out the module parameter, the type becomes:

```
'c -> ('a -> 'b) -> 'd
```

with the following constraints:

```
'a t = 'c
'b t = 'd
```

Type-checking a call to `map` determines the type variables `'c` and `'d`. For example, the following call to `map`:

```
let f x =
  map [x; x] (fun y -> (y, y))
```

refines the constraints to the following:

```
'a t = 'e list
('a * 'a) t = 'd
```

where `'a`, `'d` and `'e` are all type variables representing unknown types.

We might be tempted to attempt to refine the constraints further, inferring that `'a = 'e` and that `s t = s list` for any type `s`. However, this inference is not necessarily correct. If, instead of `MonadList`, the following module was in scope:

```
implicit module MonadOdd = struct
  type 'a t = int list
  let return x = [1; 2; 3]
  let bind m f = [4; 5; 6]
end
```

then those inferences would be incorrect. Since the definition of the type `t` in `MonadOdd` simply discards its parameter, there is no requirement for `'e` to be equal to `'a`. Further, for any type `s`, `s t` would be equal to `int list`, not to `s list`.

In fact, inferring additional information from these constraints before performing resolution would constitute higher-order unification, which is undecidable in general.

Once the constraints have been used to resolve the module argument `M` to `MonadList`, we can safely substitute `list` for `M.t` which gives us the expected type equalities.

### 2.2.2 Searching for a matching module

Once the module type of the implicit argument has been constrained, the next step is to find a suitable module. A module is considered suitable for use as the implicit argument if it satisfies three criteria:

1. It is constructed from the modules and functors in the implicit search space.
2. It matches the constrained module type for the implicit argument.
3. It is unique – that is, it is the only module satisfying the first two criteria.

**The implicit search space** The implicit search space consists of those modules which have been bound with `implicit module` or `let implicit module`, or which are in scope as implicit parameters. For example, in the following code all of `M`, `P` and `L` are included in the implicit search space at the point of the expression `show v`

```
implicit module M = M1
module N = M2
let f {P : Show} v ->
  let implicit module L = M3 in show v
```

Furthermore, in order to avoid unnecessary ambiguity, resolution is restricted to those modules which are accessible using unqualified names. An implicit sub-module `M` in a module is not in scope unless `N` has been opened. Implicit modules from other modules can be brought into scope using `open implicit` or `let open implicit`.

**Module type matching** Checking that an implicit module `M` matches an implicit argument type involves checking that the signature of `M` matches the signature of the argument and that the constraints generated by type checking hold for `M`. As with regular OCaml modules, signature matching allows `M` to have more members than the argument signature. For example, the following module matches the module type `Show with type t = int`, despite the fact that the module has an extra value member, `read`:

```
implicit module Show_read_int = struct
  type t = int
  let show = string_of_int
  let read = int_of_string
end
```

Constraint matching is defined in terms of substitution: can the type variables in the generated constraint set be instantiated such that the equations in the set hold for `M`? For example, `MonadList` meets the constraint

```
'a t = int list
```

by replacing `'a` with `int`, giving the valid equation

```
int MonadList.t = int list
```

In simple cases, resolution is simply a matter of trying each implicit module in turn to see whether it matches the signature and generated constraints.

However, when there are implicit functors in scope the resolution procedure becomes more involved. For example, the declaration for `ShowList` from Figure 1 allows modules such as `ShowList(ShowInt)` to be used as implicit module arguments:

```
implicit module ShowList {S : Show} = struct
  type t = S.t list
  let show l = string_of_list S.show l
end
```

Checking whether an implicit functor can be used to create a module which satisfies an implicit argument's constraints involves substituting an application of the functor for the implicit argument and checking that the equations hold. For example, applying `ShowList` could meet the constraint:

```
t = int list
```

as substituting an application of the functor gives:

```
ShowList{S}.t = int list
```

which expands out to

```
S.t list = int list
```

This generates a constraint on the argument to `ShowList`:

```
t = int
```

Since `ShowInt` satisfies this new constraint, `ShowList(ShowInt)` meets the original constraint.

**Uniqueness** In order to maintain coherence, modular implicits require the module returned by resolution to be unique. Without a uniqueness requirement the result of resolution (and hence the behaviour of the program) might depend on some incidental aspect of type-checking.

To check uniqueness all possible solutions to a resolution must be considered. This requires that the search for possible resolutions terminate: if the resolution procedure does not terminate then we do not know whether there may be multiple solutions.

**Termination** Implicit functors can be used multiple times whilst resolving a single implicit argument. For example

```
show [ [1; 2; 3]; [4; 5; 6] ]
```

will resolve the implicit argument of `show` to `ShowList(ShowList(ShowInt))`.

This means that care is needed to avoid non-termination in the resolution procedure. For example, the following functor, which tries to define how to show a type in terms of how to show that type, is obviously not well-founded:

```
implicit module ShowIt {S : Show} = struct
  type t = S.t
  let show = show
end
```

Type classes ensure the termination of resolution through a number of restrictions on instance declarations. However, termination of an implicit parameter resolution depends on the scope in which the resolution is performed. For this reason, the modular implicits system places restrictions on the behaviour of the resolution directly and reports an error only when a resolution which breaks these restrictions is actually attempted.

When considering a module expression containing multiple applications of an implicit functor, such as the following:

```
ShowList(ShowList( ... ))
```

the system checks that the constraints that each application of the functor must meet are strictly smaller than the previous application of the functor. "Strictly smaller" is defined point-wise: all constraints must be smaller, and at least one constraint must be strictly smaller.

For example, resolving an implicit module argument of type `Show` with the following constraint

```
t = int list list
```

would involve considering `ShowList(ShowList(S))` where `S` is not yet determined. The first application of `ShowList` would generate a constraint on its argument:

```
t = int list
```

Thus the second application of `ShowList` must meet constraints which are strictly smaller than the constraints which the first application of `ShowList` met, and resolution can safely continue.

Whereas, considering `ShowIt(ShowIt(S))` for the same constraint, the first application of `ShowIt` would generate a constraint on its argument:

```
t = int list list
```

Thus the second application of `ShowIt` must meet constraints which are the same as the constraints which the first application of `ShowIt` met, and resolution would fail with a termination error.

As termination is required to check uniqueness, failure to meet the termination restrictions must be treated as an error. The system cannot simply ignore the non-terminating possibilities and continue to look for an alternative resolution.

## 2.3 Elaboration

Once all implicit arguments in a program have been instantiated there is a phrase-by-phrase elaboration which turns each new construct into a straightforward use of existing OCaml constructs. The elaboration makes use of OCaml's first-class modules (packages), turning functions with implicit arguments into first-class functors,

Figure 4 gives the elaboration from a fully-instantiated program into implicit-free OCaml. The types of functions which accept implicit arguments

```
{M: S} -> t
```

become first-class functor types

```
(module functor (M:S) -> sig val value : t end)
```

with a functor parameter in place of the implicit parameter `M` and a signature with a single value member of type `t` in place of the return type `t`. (The syntax used here for the first-class functor type is not currently accepted by OCaml, which restricts the types of first-class modules to named module types, but the restriction is for historical reasons only, and so we ignore it in our treatment. The other parts of the elaboration target entirely standard OCaml.)

An expression which constructs a function that accepts an implicit argument

```
fun {M: S} -> e
```

becomes an expression which packs a functor

```
(module functor (M: S) -> struct
     let value = e
   end)
```

following the elaboration on types, turning the implicit argument into a functor argument and the body into a single value binding `value`.

The applications of a function `f` to an instantiated implicit arguments `M`

```
f {M}
```

becomes an expression which unpacks `f` as a functor `F`, applies `F` to the module argument `M`, and projects the value component from the result:

```
let module F = (val f) in
let module F' = F(M) in
  F'.value
```

Care must, of course, be taken to ensure that the name `F` does not collide with any of the free variables in the module expression `M`.

Each implicit module binding

```
implicit module M { M1 : T1 } { M2 : T2 } ... { Mn : Tn } = N
```

**Types** The type

```
{M: S} -> t
```

elaborates to the package type

```
(module functor (M:S) -> sig val value : t end)
```

**Abstractions** The abstraction expression

```
fun {M: S} -> e
```

of type

```
{M: S} -> t
```

elaborates to the package expression

```
(module functor (M: S) -> struct
     let value = e
   end)
```

of type

```
(module functor (M: S) -> sig val value : t end))
```

**Applications** The application expression

```
f {M}
```

elaborates to the expression

```
let module F = (val f) in
let module F' = F(M) in
  F'.value
```

**Bindings and declarations** The implicit module binding

```
implicit module M { M1 : T1 } { M2 : T2 } ... { Mn : Tn } = N
```

elaborates to the expression

```
module M (M1 : T1) (M2 : T2) ... (Mn : Tn) = N
```

(and similarly for local bindings and signatures).
The statement

```
open implicit M
```

is removed from the program (and similarly for local `open implicit` bindings).

Figure 4: Elaboration from a fully-instantiated program into OCaml

becomes under the elaboration a binding for a regular module, turning implicit parameters into functor parameters:

```
module M (M1 : T1) (M2 : T2) ... (Mn : Tn) = N
```

The implicit module binding for `M` introduces `M` both into the implicit search space and the standard namespace of the program. The implicit search space is not used in the program after elaboration, and so the elaborated binding introduces `M` only into the standard namespace. The elaboration for local bindings and signatures is the same, mutatis mutandis.

The statement

```
open implicit M
```

serves no purpose after elaboration, and so the elaboration simply removes it from the program. Similarly, the statement

```
let open implicit M in e
```

is elaborated simply to the body:

```
e
```

## 2.4 Why target first-class functors?

The elaboration from an instantiated program into first-class functors is quite simple, but the syntax of implicit arguments suggests an even simpler translation which turns each function with an implicit parameter into a function (rather than a functor) with a first-class module parameter. For example, here is the definition of `show` once again:

```
let show {S : Show} (v : S.t) = S.show v
```

Under the elaboration in Figure 4 the definition of `show` becomes the following first-class functor binding:

```
let show =
  (functor (S: Show) -> struct
        let value = fun (v : S.t) -> S.show v
      end)
```

but we could instead elaborate into a function with a first-class module argument

```
let show (module S: Show) (v : S.t) = S.show v
```

of type

```
(module Show with type t = 'a) -> 'a -> string
```

Similarly, under the elaboration in Figure 4 the application of `show` to an argument

```
show {Show_int}
```

is translated to an expression with two local module bindings, a functor application and a projection:

```
let module F = (val show) in
let module F' = F(Show_int) in
F'.value
```

but under the elaboration into functions with first-class module arguments the result is a simple application of `show` to a packed module:

```
show (module Show_int)
```

However, the extra complexity in targeting functors rather than functions pays off in support for higher-rank and higher-kinded polymorphism.

### 2.4.1 Higher-rank polymorphism

It is convenient to have overloaded functions be first-class citizens in the language. For example, here is a function which takes an overloaded function `sh` and applies it both to an integer and to a string:

```
let show_stuff (sh : {S : Show} -> S.t -> string) =
  (sh {ShowInt} 3, sh {ShowString} "hello")
```

This application of the parameter `sh` at two different types requires `sh` to be polymorphic in the type `S.t`. This form of polymorphism, where function arguments themselves can be polymorphic functions, is sometimes called *higher-rank* polymorphism.

The elaboration of overloaded functions into first-class functors naturally supports higher-rank polymorphism, since functors themselves can behave like polymorphic functions, with type members in their arguments. Here is the elaboration of `show_stuff`:

```
let show_stuff (sh : (module functor (S : Show) -> sig
                        val value : S.t -> string
                      end)) =
  let module F1 = (val sh) in
  let module Res1 = F1(ShowInt) in
  let module F2 = (val sh) in
  let module Res2 = F2(ShowString) in
    (Res1.value 5, Res2.value "hello")
```

The two functor applications `F1(ShowInt)` and `F2(ShowString)` correspond to two instantiations of a polymorphic function.

In contrast, if we were to elaborate overloaded functions into ordinary functions with first-class module parameters then the result of the elaboration would not be valid OCaml. Here is the result of such an elaboration:

```
let show_stuff (sh : (module S with type t = 'a)
                        -> 'a -> string) =
  sh (module ShowInt) 3 ^ " " ^ sh (module ShowString) "hello"
```

Since `sh` is a regular function parameter, OCaml's type rules assign it a monomorphic type. The function is then rejected, because `sh` is applied to modules of different types within the body.

### 2.4.2 Higher-kinded polymorphism

First-class functors also provide support for *higher-kinded* polymorphism – that is, polymorphism in type constructors which have parameters. For example, Figure 2 defines a number of functions that are polymorphic in the monad on which they operate, such as `map`, which has the following type:

```
val map : {M : Monad} -> 'a M.t -> ('a -> 'b) -> 'b M.t
```

This type is polymorphic in the parameterised type constructor `M.t`.

Once again, elaborating overloaded functions into first-class functors naturally supports higher-kinded polymorphism, since functor arguments can be used to abstract over parameterised type constructors. Here is the definition of `map` once again:

```
let map {M : Monad} (m : 'a M.t) f =
  m >>= fun x -> return (f x)
```

and here its its elaboration:

```
let map =
  (functor (M: Monad) -> struct
     let value =
       let module F_bind = (val (>>=)) in
       let module R_bind = F_bind(M) in
       let module F_ret = (val return) in
       let module R_ret = F_ret(M) in
         R_bind.value m (fun x -> R_ret (f x))
   end)
```

As with higher-rank polymorphism, there is no suitable elaboration of overloaded functions involving higher-kinded polymorphism into functions with first-class module parameters, since higher-kinded polymorphism is not supported in OCaml's core language.

### 2.4.3 First-class functors and type inference

Type inference for higher-rank and higher-kinded polymorphism is undecidable in the general case, and so type systems which support such polymorphism require type annotations. For instance, annotations are required on all first-class functor parameters, and on recursive definitions of recursive functors. The same requirements apply to functions with implicit module arguments.

For example, the following function will not type-check if the `sh` parameter is not annotated with its type:

```
let show_three sh =
  sh {ShowInt} 3
```

Instead, `show_three` must be defined as follows:

```
let show_three (sh : {S : Show} -> S.t -> string)  =
  sh {ShowInt} 3
```

Requiring type annotations means that type inference is not *order independent* – if the body of `show_three` were type-checked before its parameter list then inference would fail. To maintain predictability of type inference, some declarative guarantees are made about the order of type-checking; for example, a variable's binding will always be checked before its uses. If type inference of a program only succeeds due to an ordering between operations which is not ensured by these guarantees then the OCaml compiler will issue a warning.

# 3 Modular implicits by example

The combination of the implicit resolution mechanism and the integration with the module language leads to a system which can support a wide range of programming patterns. We demonstrate this with a selection of example programs.

## 3.1 Defining overloaded functions

Some overloaded functions, such as `show` from Figure 1, simply project a member of the implicit module argument. However, it is also common to define an overloaded function in terms of an existing overloaded function. For example, the following `print` function composes the standard OCaml function `print_string` with the overloaded function `show` to print a value to standard output:

```
let print {S: Show} (v: S.t) =
  print_string (show v)
```

It is instructive to consider the details of resolution for the call to `show` in the body of `print`. As described in Section 2.2, resolution of the implicit argument to `show` involves generating constraints for the types in the signature `Show`, unifying with the context to refine the constraints, and then searching for a module `M` which matches the signature of `Show` and satisfies the constraints.

Since there is a single type `t` in the signature `Show`, resolution begins with the constraint set

```
t = 'a
```

and gives the variable `show` the type `'a -> string`. Unification with the ascribed type of the parameter `v` instantiates `'a`, refining the constraint

```
t = S.t
```

Since the type `S.t` is an abstract member of the implicit module parameter `S`, the search for a matching module returns `S` as the unique implicit module which satisfies the constraint.

The ascription on the parameter `v` plays an essential role in this process. Without the ascription, resolution would involve searching for an implicit module of type `Show` satisfying the constraint `t = 'a`. Since any implicit module matching the signature `Show` satisfies this constraint, regardless of the definition of `t`, the resolution procedure will fail with an ambiguity error if there are multiple implicit modules in scope matching `Show`.

## 3.2 Instance constraints

Haskell's instance constraints make it possible to restrict the set of instantiations of type parameters when defining overloaded functions. For example, here is an instance of the `Show` class for the pair constructor `(,)`, which is only available when there are also an instance of `Show` for the type parameters `a` and `b`:

```
instance (Show a, Show b) => Show (a, b) where
   show (x, y) = "(" ++ show x ++ "," ++ show y ++ ")"
```

With modular implicits, instance constraints become parameters to implicit module bindings:

```
implicit module Show_pair {A: Show} {B: Show} = struct
   type t = A.t * B.t
   let show (x, y) = "(" ^ A.show x ^ "," ^ B.show y ^ ")"
end
```

It is common for the types of implicit module parameters to be related to the type of the whole, as in this example, where the parameters each match `Show` and the result has type `Show with type t = A.t * B.t`. However, neither instance constraints nor implicit module parameters require that the parameter and the result types are related. Here is the definition of an implicit module `Complex_cartesian`, which requires only that the parameters have implicit module bindings of type `Num`, not of type `Complex`:

```
implicit module Complex_cartesian {N: Num} = struct
  type t = N.t complex_cartesian
  let conj { re; im } = { re; im = N.negate im }
end
```

(We leave the reader to deduce the definitions of the `complex_cartesian` type and of the `Num` signature.)

## 3.3 Inheritance

Type classes in Haskell provide support for *inheritance*. For example, the `Ord` type class is defined as inheriting from the `Eq` type class:

```
class Eq a where
  (==) :: a -> a -> Bool

class Eq a => Ord a where
  compare :: a -> a -> Ordering
```

This means that instances of `Ord` can only be created for types which have an instance of `Eq`. By declaring `Ord` as inheriting from `Eq`, functions can use both `==` and `compare` on a type with a single constraint that the type have an `Ord` instance.

### 3.3.1 The "diamond" problem

It is tempting to try to implement inheritance with modular implicits by using the structural subtyping provided by OCaml's modules. For example, one might try to define `Ord` and `Eq` as follows:

```
module type Eq = sig
  type t
  val equal : t -> t -> bool
end

let equal {E : Eq} x y = E.equal x y

module type Ord = sig
  type t
  val equal : t -> t -> bool
  val compare : t -> t -> int
end

let compare {O : Ord} x y = O.compare x y
```

which ensures that any module which can be used as an implicit `Ord` argument can also be used as an implicit `Eq` argument. For example, a single module can be created for both equality and comparison of integers:

```
implicit module OrdInt = struct
  type t = int
  let equal = Int.equal
  let compare = Int.compare
end
```

However, an issue arises when trying to implement implicit functors for type constructors using this scheme. For example, we might want to define the following two implicit functors:

```
implicit module EqList {E : Eq} = struct
  type t = E.t list
  let equal x y = List.equal E.equal x y
end

implicit module OrdList {O : Ord} = struct
  type t = O.t list
  let equal x y = List.equal O.equal x y
  let compare x y = List.compare O.compare x y
end
```

which implement `Eq` for lists of types which implement `Eq`, and implement `Ord` for lists of types which implement `Ord`.

The issue arises when we wish to resolve an `Eq` instance for a list of a type which implements `Ord`. For example, we might wish to apply the `equal` function to lists of ints:

```
equal [1; 2; 3] [4; 5; 6]
```

The implicit argument in this call is ambiguous: we can use either `EqList(OrdInt)` or `OrdList(OrdInt)`.

This is a kind of "diamond" problem: we can restrict `OrdInt` to an `Eq` and then lift it using `EqList`, or we can lift `OrdInt` using `OrdList` and then restrict the result to an `Eq`.

In Haskell, the problem is avoided by canonicity – it doesn't matter which way around the diamond we go, we know that the result will be the same.

### 3.3.2 Module aliases

OCaml provides special support for *module aliases* [5]. A module can be defined as an alias for another module:

```
module L = List
```

This defines a new module whose type is the singleton type "`= List`". In other words, the type of `L` guarantees that it is equal to `List`. This equality allows types such as `Set(List).t` and `Set(L).t` to be considered equal.

Since `L` is statically known to be equal to `List`, we do not consider an implicit argument to be ambiguous if `L` and `List` are the only possible choices.

In our proposal we extend module aliases to support implicit functors. For example,

```
implicit module ShowL {S : Show} = ShowList{S}
```

creates a module alias. This means that `ShowL(ShowInt)` is an alias for `ShowList(ShowInt)`, and its type guarantees that the two modules are equal.

In order to maintain coherence we must require that all implicit functors be pure. If `ShowList` performed side-effects then two separate applications of it would not necessarily be equal. We ensure this using the standard OCaml value restriction. This is a very conservative approximation of purity, but we do not expect it to be too restrictive in practice.

### 3.3.3 Inheritance with module aliases

Using module aliases we can implement inheritance using modular implicits. Our `Ord` example is encoded as follow:

```
module type Eq = sig
  type t
  val equal : t -> t -> bool
end

let equal {E : Eq} x y = E.equal x y

module type Ord = sig
  type t
  module Eq : Eq with type t = t
  val compare : t -> t -> int
end

let compare {O : Ord} x y = O.compare x y

implicit module EqOrd {O : Ord} = O.Eq

implicit module EqInt = struct
  type t = int
  let equal = Int.equal
end

implicit module OrdInt = struct
  type t = int
  module Eq = EqInt
  let compare = Int.compare
end

implicit module EqList {E : Eq} = struct
  type t = E.t list
  let equal x y = List.equal E.equal x y
end
```

```
implicit module OrdList {O : Ord} = struct
  type t = O.t list
  module Eq = EqList{O.Eq}
  let compare x y = List.compare O.compare x y
end
```

The basic idea is to represent inheritance by including a submodule of the inherited type, along with an implicit functor to extract that submodule. By wrapping the inherited components in a module they can be aliased.

The two sides of the "diamond" are now `EqList(EqOrd(OrdInt))` or `EqOrd(OrdList(OrdInt))`, both of which are aliases for `EqList(EqInt)` so there is no ambiguity.

### 3.4 Constructor classes

Since OCaml's modules support type members which have type parameters, modular implicits naturally support *constructor classes* [7] – i.e. functions whose implicit instances are indexed by parameterised type constructors. For example, here is a definition of a `Functor` module type, together with implicit instances for the parameterised types `list` and `option`:

```
module type Functor = sig
  type +'a t
  val map : ('a -> 'b) -> 'a t -> 'b t
end

let map {F: Functor} (f : 'a -> 'b) (c : 'a F.t) = F.map f c

implicit module Functor_list = struct
  type 'a t = 'a list
  let map = List.map
end

implicit module Functor_option = struct
  type 'a t = 'a option
  let map f = function
     None -> None
   | Some x -> Some (F x)
end
```

The choice to translate implicits into first-class functors makes elaboration for implicit modules with parameterised types straightforward. Here is the elaborated code for `map`:

```
let map =
  (module functor (F: Functor) -> struct
      let value (f : 'a -> 'b) (c : 'a F.t) = F.map f c
    end)
```

### 3.5 Multi-parameter type classes

Most of the examples we have seen so far involve resolution of implicit modules with a single type member. However, nothing in the design of modular implicits restricts resolution to a single type. The module signature inclusion relation on which resolution is based supports modules with an arbitrary number of type members (and indeed, with many other components, such as modules and module types).

Here is an example illustrating overloading with multiple types. The `Widen` signature includes two type members, `slim` and `wide`, and a coercion function `widen` for converting from the former to the latter. The two implicit modules, `Widen_int_float` and `Widen_opt`, respectively implement conversion from a `ints` to `floats`, and lifting of widening to options. The final line illustrates the instantiation of a widening function from `int option` to `float option`, based on the three implicit modules.

```
module type Widen =
sig
  type slim
  type wide
  val widen : slim -> wide
end

let widen {C:Widen} (v: C.slim) : C.wide = C.widen v

implicit module Widen_int_float =
struct
  type slim = int
  type wide = float
  let widen = Pervasives.float
end

implicit module Widen_opt{A: Widen} =
struct
  type slim = A.slim option
  type wide = A.wide option
  let widen = function
     None -> None
   | Some v -> Some (A.widen v)
end

let v : int option = widen (Some 3)
```

In order to find a suitable implicit argument for the call to `widen` on the last line, the resolution procedure first generates fresh types variables for `slim` and `wide`

```
slim = 'a
wide = 'b
```

and replaces the corresponding names in the type of the variable `widen`:

```
widen : 'a -> 'b
```

Unifying this last type with the type supplied by the context (i.e. the type of the argument the ascribed result type) reveals that `'a` should be equal to `int option` and `'b` should be equal to `float option`. The search for a suitable argument must therefore find a module of type `Widen` with the following constraints:

```
slim = int option
wide = float option
```

The implicit functor `Widen_option` is suitable if a modules `A` can be found such that A has type `Widen` with the constraints

```
slim = int
wide = float
```

The implicit module `Widen_int_int` satisfies these constraints, and the search is complete.

The instantiated call shows the implicit module argument constructed by the resolution procedure:

```
let v : int option = widen {Widen_option(Widen_int_float)} (Some 3)
```

### 3.6 Associated types

Since OCaml modules can contain abstract types, searches can be existentially quantified. For example, we can ask for a type which can be shown

```
Show
```

rather than how to show a specific type

```
Show with type t = int
```

The combination of signatures with multiple type members and support for existential searches gives us similar features to Haskell's associated types [1]. We can search for a module based on a subset of the types it contains and the search will fill-in the remaining types for us. For example, here is a module type `Array` for arrays with a type `t` of arrays and a type `elem` of array elements, together with a function `create` for creating arrays:

```
module type Array = sig
  type t
  type elem
  [...]
end

val create : {A : Array} -> int -> A.elem -> A.t
```

The `create` function can be used without specifying the array type being created:

```
let x = create 5 true
```

This will search for an implicit `Array with type elem = bool`. When one is found `x` will correctly be given the associated `t` type. This allows different array types to be used for different element types (e.g. bit vectors to represent arrays of bools).

### 3.7 Backtracking

The Haskell system of type classes ignores instance constraints when determining whether two instances are ambiguous. For example, the following two instance constraints are always considered ambiguous:

```
instance Floating n => Complex (Complex_Cartesian n)
instance Integral n => Complex (Complex_Cartesian n)
```

In contrast, modular implicits only considers those implicit functors for which suitable arguments are in scope as candidates for instantiation. For example, the following two implicit functors are not inherently ambiguous:

```
implicit module Complex_cartsian_floating {N: Floating}
  : Complex with type t = N.t complex_cartesian
implicit module Complex_cartsian_integral {N: Integral}
  : Complex with type t = N.t complex_cartesian
```

The `Complex_cartesian_floating` and `Complex_cartesian_integral` modules only give rise to ambiguity if constraint generation (Section 2.2.1) determines that the type `t` of the `Complex` signature should be instantiated to `s complex_cartesian` where there are instances of both `Floating` and `Integral` in scope for `s`:

```
implicit module Floating_s : Floating with type t = s
implicit module Integral_s : Integral with type t = s
```

Taking functor arguments into account during resolution is a form of backtracking. The resolution procedure considers both `Complex_cartesian_integral` and `Complex_cartesian_floating` as candidates for instantiation and attempts to find suitable arguments for both. The resolution is only ambiguous if both implicit functors can be applied to give implicit modules of the appropriate type.

### 3.8 Local instances

The `let implicit` construct described in Section 2.1 makes it possible to define implicit modules whose scope is limited to a particular expression. The following example illustrates how these local implicit modules can be used to select alternative behaviours when calling overloaded functions.

Here is a signature `Ord`, for types which support comparison:

```
module type Ord = sig
  type t
  val cmp : t -> t -> int
end
```

The `Ord` signature makes a suitable type for the implicit argument of a `sort` function:

```
val sort : {O: Ord} -> O.t list -> O.t list
```

Each call to `sort` constructs a suitable value for `Ord` from the implicit modules and functors in scope. Two possible orderings for `int` are:

```
module Ord_int = struct
  type t = int
  let cmp l r = Pervasives.compare l r
end
```

```
module Ord_int_rev = struct
 type t = int
  let cmp l r = Pervasives.compare r l
end
```

Either ordering can be used with `sort` by passing the argument explicitly:

```
sort {Ord_int} items
```

or

```
sort {Ord_int_rev} items
```

Explicitly passing implicit arguments bypasses the resolution mechanism altogether. It is occasionally useful to combine overriding of implicit modules for particular types with automatic resolution for other types. For example, if the following implicit module definition is in scope then `sort` can be used to sort lists of pairs of integers:

```
implicit module Ord_pair {A: Ord} {B: Ord} = struct
  type t = A.t * B.t
  let cmp (x1, x2) (y1, y2) =
    let c = A.cmp x1 y1 in
    if c <> 0 then c else B.cmp x2 y2
end
```

Suppose that we want to use `Ord_pair` together with both the regular and reversed integer comparisons to sort a list of pairs. One approach is to construct and pass entire implicit arguments explicitly:

```
sort {Ord_pair(Int_ord_rev)(Int_ord_rev)} items
```

Alternatively (and equivalently), local implicit module bindings for `Ord` and `Ord_int_rev` make it possible to override the behaviour at ints while using the automatic resolution behaviour to locate and use the `Ord_pair` functor:

```
let sort_both_ways items =
  let ord =
    let implicit module Ord = Ord_int in
      sort items
  in
  let rev =
    let implicit module Ord = Ord_int_rev in
      sort items
  in
    ord, rev
```

In Haskell, which lacks both local instances and a way of explicitly instantiating type class dictionary arguments, neither option is available, and programmers are advised to define library functions in pairs, with one function (such as `sort`) that uses type classes to instantiate arguments automatically, and one function (such as `sortBy`) that accepts a regular argument in place of a dictionary:

```
sort :: Ord a => [a] -> [a]
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
```

### 3.9 Structural matching

As Section 2.2.2 explains, picking a suitable implicit argument involves a module which matches a constrained signature. In contrast to Haskell's type classes, matching is therefore defined structurally (in terms of the names and types of module components) rather than nominally (in terms of the name of the signature). Structural matching allows the caller of an overloaded function to determine which part of a signature is required rather than requiring the definer of a class to anticipate which overloaded functions are most suitable for grouping together.

It is not difficult to find situations where structural matching is useful. The following signature for types which support basic arithmetic, with members for zero and one, and for addition and subtraction:

```
module type Num = sig
  type t
  val zero : t
  val one : t
  val ( + ) : t -> t -> t
  val ( * ) : t -> t -> t
end
```

The following implicit modules implement `Num` for the types `int` and `float`, using functions from OCaml's standard library:

```
implicit module Num_int = struct
  type t = int
  let zero = 0
  let one = 1
  let ( + ) = Pervasives.( + )
  let ( * ) = Pervasives.( * )
end

implicit module Num_float = struct
  type t = float
  let zero = 0.0
  let one = 1.0
  let ( + ) = Pervasives.( +. )
  let ( * ) = Pervasives.( *. )
end
```

The `Num` signature makes it possible to define a variety of arithmetic functions. However, in some cases `Num` offers more than necessary. For example, defining an overloaded function `sum` to compute the sum of a list of values requires only `zero` and `+`, not `one` and `*`. Using `Num` as the implicit signature for `sum` would make unnecessarily exclude types (such as strings) which have a notion of addition but which do not support multiplication.

Defining more constrained signatures makes it possible to define more general functions. Here is a signature `Add` which includes only those elements of `Num` involved in addition:

```
module type Add = sig
  type t
  val zero : t
```

```
  val ( + ) : t -> t -> t
end
```

Using `Add` we can define a `sum` which works for any type that has an implicit module with definitions of `zero` and `plus`:

```
let sum {A: Add} (l : A.t list) =
  List.fold_left A.( + ) A.zero l
```

The existing implicit modules `Num_int` and `Num_float` can be used with `sum`, since they both match `Add`. The following module, `Add_string`, also matches `Add`, making it possible to use `sum` either for summing a list of numbers or for concatenating a list of strings:

```
implicit module Add_string
  : Add with type t = string =
struct
  type t = string
  let zero = ""
  let ( + ) = Pervasives.( ^ ) (* concatenation *)
end
```

In other cases it may be necessary to use some other part of the `Num` interface. The following function computes an inner product for any type with an implicit module that matches `Num`:

```
let dot {N: Num} (l1 : N.t list) (l2 : N.t list) =
  sum (List.map2 N.( * ) l1 l2)
```

This time it would not be sufficient to use `Add` for the type of the implicit argument, since `dot` uses both multiplication and addition. However, `Add` still has a role to play: the implicit argument of `sum` uses the implicit argument `N` with type `Add`. Since the `Num` signature is a subtype of `Add` according to the rules of OCaml's module system, the argument can be passed through directly to `sum`. Here is the elaboration of `dot`, showing how the `sum` functor being unpacked, bound to `F`, then applied to the implicit argument `N`:

```
let dot =
  (module functor (N: Num) -> struct
       let value = fun (l1 : N.t list) (l2 : N.t list) ->
          let module F = (val sum) in
          let module F' = F(N) in
          F'.value (List.map2 N.( * ) l1 l2)
     end)
```

An optimising compiler might lift the unpacking and application of `sum` outside the body of the function, in order to avoid repeating the work each time the list arguments are supplied.

## 4 Canonicity

In Haskell, a type class has at most one instance per type within a program. For example, defining two instances of `Show` for the type `Int` or for the type constructor `Maybe` is not permitted. We call this property *canonicity*.

Haskell relies on canonicity to maintain coherence, whereas canonicity cannot be preserved by our system due to OCaml's support for modular abstraction.

### 4.1 Inference, coherence and canonicity

A key distinction between type classes and implicits is that, with type classes, constraints on a function's type can be inferred based on the use of other constrained functions in the function's definitions. For example, if a `show_twice` function uses the `show` function:

```
show_twice x = show x ++ show x
```

then Haskell will infer that `show_twice` has type `Show a => a -> String`.

This inference raises issues for coherence in languages with type classes. For example, suppose we have the following instance:

```
instance Show a => Show [a] where
    show l = showList l
```

and consider the function:

```
show_as_list x = show [x]
```

There are two valid types which could be inferred for this function:

```
show_as_list :: Show [a] => a -> String
```

or

```
show_as_list :: Show a => a -> String
```

In the second case, the `Show [a]` instance has been used to reduce the constraint to `Show a`.

The choice between these two types changes where the `Show [a]` constraint is resolved. In the first case it will be resolved at *calls* to `show_as_list`. In the second case it has been resolved at the *definition* of `show_as_list`.

If type class instances are canonical then it does not matter where a constraint is resolved, as there is only a single instance to which it could be resolved. Thus, with canonicity, the inference of `show_as_list`'s type cannot affect the dynamic semantics of the program, and coherence is preserved.

However, if type class instances are not canonical then where a constraint is resolved can affect which instance is chosen, which in turn changes the dynamic semantics of the program. Thus, without canonicity, the inference of `show_as_list`'s type can affect the dynamic semantics of the program, breaking coherence.

### 4.2 Canonicity and abstraction

It would not be possible to preserve canonicity in OCaml because type aliases can be made abstract. Consider the following example:

```
module F (X : Show) = struct
  implicit module S = X
end

implicit module ShowInt = struct
  type t = int
  let show = string_of_int
end
```

```
F(struct
  type t = int
  let show _ = "An int"
end)
```

The functor `F` defines an implicit `Show` module for the abstract type `X.t`, whilst the implicit module `ShowInt` is for the type `int`. However, `F` is later applied to a module where `t` is an alias for `int`. This violates canonicity but this violation is hidden by abstraction.

Whilst it may seem that such cases can be detected by peering through abstractions, this is not possible in general and defeats the entire purpose of abstraction. Fundamentally, canonicity is not a modular property and cannot be respected by a language with full support for modular abstraction.

### 4.3 Canonicity as a feature

Besides maintaining coherence, canonicity is sometimes a useful feature in itself. The canonical example for the usefulness of canonicity is the `union` function for sets in Haskell. The `Ord` type class defines an ordering for a type[4]:

```
class Ord a where
  (<=) :: a -> a -> Bool
```

This ordering is used to create sets implemented as binary trees:

```
data Set a
empty :: Set a
insert :: Ord a => a -> Set a -> Set a
delete :: Ord a => a -> Set a -> Set a
```

The `union` function computes the union of two sets:

```
union :: Ord a => Set a -> Set a -> Set a
```

Efficiently implementing this union requires both sets to have been created using the same ordering. This property is ensured by canonicity, since there is only one instance of `Ord a` for each `a`, and all sets of type `Set a` must have been created using it.

### 4.4 An alternative to canonicity as a feature

In terms of modular implicits, Haskell's `union` function would have type:

```
val union: {O : Ord} -> O.t set -> O.t set -> O.t set
```

but without canonicity it is not safe to give `union` this type since there is no guarantee that all sets of a given type are using the same ordering.

The issue is that the `set` type is only parametrised by the type of its elements, when it should really be also parametrised by the ordering used to create it. Traditionally, this problem is solved in OCaml by using applicative functors:

```
module Set (O : Ord) : sig
  type elt
```

[4] Some details of `Ord` are omitted for simplicity

```
  type t
  val empty : t
  val add : elt -> t -> t
  val remove : elt -> t -> t
  val union : t -> t -> t
  [...]
end
```

When applied to an `Ord` argument `O`, the `Set` functor produces a module containing the following functions:

```
val empty : Set(O).t
val add : elt -> Set(O).t -> Set(O).t
val remove : elt -> Set(O).t -> Set(O).t
val union : Set(O).t -> Set(O).t -> Set(O).t
```

The same approach transfers to modular implicits, giving our polymorphic set operations the following types:

```
val empty : {O : Ord} -> Set(O).t
val add : {O : Ord} -> O.t -> Set(O).t -> Set(O).t
val remove : {O : Ord} -> O.t -> Set(O).t -> Set(O).t
val union : {O : Ord} -> Set(O).t ->
              Set(O).t -> Set(O).t
```

The type for sets is now `Set(O).t` which is parametrised by the ordering module `O`, ensuring that `union` is only applied to sets using the same ordering.

# 5 Order independence and compositionality

Two properties enjoyed by traditional ML type-systems are *order independence* and *compositionality*. This section describes how modular implicits affect these properties.

## 5.1 Order independence

Type inference is *order independent* when the order in which expressions are type-checked does not affect whether type inference succeeds. Traditional ML type inference is order independent, however some of OCaml's advanced features, including first-class functors, cause order dependence.

As described in Section 2.2, type checking implicit applications has two aspects:

1. Inferring the types which constrain the implicit argument
2. Resolving the implicit argument using the modules and functors in the implicit scope.

These two aspects are inter-dependent: the order in which they are performed affects whether type inference succeeds.

### 5.1.1 Resolution depends on types

Consider the implicit application from line 24 of our `Show` example (Figure 1):

```
show 5
```

Resolving the implicit argument `S` requires first generating the constraint `t = int`. Without this constraint the argument would be ambiguous – it could be `ShowInt`, `ShowFloat`, `ShowList(ShowFloat)`, etc. This constraint can only come from type-checking the non-implicit argument `5`.

This demonstrates that resolution depends on type inference, and so some type inference must be done before implicit arguments are resolved.

### 5.1.2 Types depend on resolution

Given that resolution depends on type inference, we might be tempted to perform resolution in a second pass of the program, after all type inference has finished. However, although it is not immediately obvious, types also depend on resolution.

Consider the following code:

```
module type Sqrtable = sig
  type t
  val sqrt : t -> t
end

let sqrt {S : Sqrtable} x = S.sqrt x

implicit module SqrtFloat = struct
  type t = float
  let sqrt x = sqrt_float x
end

let sqrt_twice x = sqrt (sqrt x)
```

The `sqrt_twice` function contains two calls to `sqrt`, which has an implicit argument of module type `Sqrtable`. There are no constraints on these implicit parameters as `x` has an unknown type; however, there is only one `Sqrtable` module in scope so the resolution is still unambiguous. By resolving `S` to `SqrtFloat` we learn that `x` in fact has type `float`.

This demonstrates that types depend on resolution, and so resolution must be done before some type inference. In particular, it is important that resolution is performed before generalisation is attempted on any types which depend on resolution because type variables cannot be unified after they have been generalised.

### 5.1.3 Resolution depends on resolution

Since resolution depends on types, and types can depend on resolution, it follows that one argument's resolution can depend on another argument's resolution.

Following on from the previous example, consider the following code:

```
module type Summable = sig
  type t
  val sum : t -> t -> t
end
```

```
let double {S : Summable} x = S.sum x x

implicit module SumInt = struct
  type t = int
  let sum x y = x + y
end

implicit module SumFloat = struct
  type t = float
  let sum x y = x +. y
end

let sqrt_double x = sqrt (double x)
```

Here there are two implicit applications: one of `sqrt` and one of `double`. As before, the arguments of these functions have no constraints since `x`'s type is unknown. If the resolution of `double`'s implicit argument is attempted without constraint it will fail as ambiguous, since either `SumInt` or `SumFloat` could be used. However, if `sqrt`'s implicit argument is first resolved to `SqrtFloat` then we learn that the return type of the call to `double` is float. This allows `double`'s implicit argument to unambiguously be resolved as `SumFloat`.

This demonstrates that resolutions can be depend on other resolutions, and so the order in which resolutions are attempted will affect which programs will type-check successfully.

### 5.1.4 Predictable inference

In the presence of order dependence, inference can be kept predictable by providing some declarative guarantees about the order of type-checking, and disallowing programs whose type inference would only succeed due to an ordering between operations which is not ensured by these guarantees. This is the approach OCaml takes with its other order-dependent features[5].

Taking the same approach with modular implicits involves two choices about the design:

1. When should implicit resolution happen relative to type inference?
2. In what order should implicit arguments be resolved?

The dependence of resolution on type inference is much stronger than the dependence of type inference on resolution: delaying type inference until after resolution would lead to most argument resolutions being ambiguous.

In order to perform as much inference as possible before attempting resolution, resolution is delayed until the point of generalisation. Technically, resolution could be delayed until a generalisation is reached which directly depends on a type involved in that resolution. However, we take a more predictable approach and resolve all the implicit arguments in an expression whenever the result of that expression is generalised.

In practice, this means that implicit arguments are resolved at the nearest enclosing let binding. For example, in this code:

```
let f g x =
  let z = [g (show 5) (show 4.5); x] in
```

[5]OCaml emits a warning rather than out-right disallowing programs which depend on an unspecified ordering

```
  g x :: z
```

the implicit arguments of both calls to `show` will be resolved after the entire expression

```
[g (show 5) (show 4.5); x]
```

has been type-checked, but before the expression

```
g x :: z
```

has been type-checked.

Our implementation of modular implicits makes very few guarantees about the order of resolution of implicit arguments within a given expression. It is guaranteed that implicit arguments of the same function will be resolved left-to-right, and that implicit arguments to a function will be resolved before any implicit arguments within other arguments to that function.

These guarantees mean that the example of dependent resolutions:

```
let sqrt_double x = sqrt (double x)
```

will resolve without ambiguity, but that the similar expression:

```
let double_sqrt x = double (sqrt x)
```

will result in an ambiguous resolution error. This can be remedied either by adding a type annotation:

```
let double_sqrt x : float = double (sqrt x)
```

or by lifting the argument into its own `let` expression to force its resolution:

```
let double_sqrt x =
  let s = sqrt x in
    double s
```

## 5.2 Compositionality

Compositionality refers to the ability to combine two independent well-typed program fragments to produce a program fragment that is also well typed. In OCaml, this property holds of top-level definitions up to renaming of identifiers.

Requiring that implicit arguments be unambiguous means that renaming of identifiers is no longer sufficient to guarantee two sets of top-level definitions can be composed. For example,

```
implicit module ShowInt1 = struct
  type t = int
  let show x = "ShowInt1: " ^ (int_of_string x)
end

let x = show 5
```

and

```
implicit module ShowInt2 = struct
  type t = int
  let show x = "ShowInt2: " ^ (int_of_string x)
end

let y = show 6
```

cannot be safely combined because the call to `show` in the definition of `y` would become ambiguous. In order to ensure that two sets of definitions can safely compose they must not contain overlapping implicit module declarations.

However, whilst compositionality of top-level definitions is lost, compositionality of modules is maintained. Any two well-typed module definitions can be combined to produce a well-typed program. This is an important property, as it allows support for separate compilation without the possibility of errors at link time.

# 6 Related work

There is a large literature on systematic approaches to ad-hoc polymorphism, Kaes [10] being perhaps the earliest example. We restrict our attention here to a representative sample.

## 6.1 Type classes

Haskell type classes [18] are the classic formalised method for ad-hoc polymorphism. They have been replicated in a number of other programming languages (e.g. Agda's instance arguments [3], Rust's traits [13]).

The key difference between approaches based on type class and approaches based on implicits is that type class constraints can be inferred, whilst implicit parameters must be defined explicitly. Haskell maintains coherence, in the presence of such inference, by ensuring that type class instances are canonical.

Canonicity is not possible in a language which supports modular abstraction (such as OCaml), and so type classes are not always a viable choice. Canonicity is also not always desirable: the restriction to a single instance per type is not compositional and can force users to create additional types to work around it.

The decision to infer constraints also influences other design choices. For example, whereas modular implicits instantiate implicit arguments only at function application sites, the designers of type classes take the dual approach of only generalizing constrained type variables at function abstraction sites [8, Section 4.5.5]. Both restrictions have the motivation of avoiding unexpected work – in Haskell, adding constraints to non-function bindings can cause a loss of sharing, whereas in OCaml, inserting implicit arguments at sites other than function calls can cause side effects to take place in the evaluation of apparently effect-free code.

Other differences between our proposal and type classes include support for backtracking during parameter resolution – allowing for more precise detection of ambiguity, and resolution based on any type defined within the module rather than on a single specific type.

## 6.2 Implicits

Scala implicits [14] are a major inspiration for this work. They provide implicit parameters on functions, which are selected from the scope of the call site based on their type. In Scala these parameters have normal Scala types, whilst we propose using module types. Scala's object system has many properties in common with a module system, so advanced features such as associated types are still possible despite Scala's implicits being based on normal types.

Scala's implicits have a more complicated notion of scope than our proposal. This seems to be aimed at fitting implicits into Scala's object-oriented approach: for example allowing implicits to be searched

for in companion objects of the class of the implicit parameter. This makes it more difficult to answer the question "Where is the implicit parameter coming from?", in turn making it more difficult to reason about code. Our proposal simply uses lexical scope when searching for an implicit parameter.

Scala supports overlapping implicit instances. If an implicit parameter is resolved to more than one definition, rather than give an ambiguity error, a complex set of rules gives an ordering between definitions, and a most specific definition will be selected. An ambiguity error is only given if multiple definitions are considered equally specific. This can be useful, but makes reasoning about implicit parameters more difficult: to know which definition is selected you must know all the definitions in the current scope. Our proposal always gives an ambiguity error if multiple implicit modules are available.

In addition to implicit parameters, Scala also supports implicit conversions. If a method is not available on an object's type the implicit scope is searched for a function to convert the object to a type on which the method is available. This feature greatly increases the complexity of finding a method's definition, and is not supported in our proposal.

Chambart et al. have proposed [2] adding support for implicits to OCaml using core OCaml types for implicit parameters. Our proposal instead uses module types for implicit parameters. This allows our system to support more advanced features including associated types and higher kinds. The module system also seems a more natural fit for ad-hoc polymorphism due to its direct support for signatures.

The implicit calculus [15] provides a minimal and general calculus of implicits which could serve as a basis for formalising many aspects of our proposal.

Coq's type classes [16] are similar to implicits. They provide implicit dependent record parameters selected based on their type.

### 6.3 Canonical structures

In addition to type classes, Coq also supports a mechanism for ad-hoc polymorphism called *canonical structures* [12]. Type classes and implicits provide a mechanism to resolve a value based on type information. Coq, being dependently typed, already uses unification to resolve values from type information, so canonical structures support ad-hoc polymorphism by providing additional ad-hoc rules that are applied during unification.

Like implicits, canonical structures do not require canonicity, and do not operate on a single specific type: ad-hoc unification rules are created for every type or term defined in the structure. Canonical structures also support backtracking of their search due to the backtracking built into Coq's unification.

### 6.4 Concepts

Gregor et al. [6] describe *concepts*, a system for ad-hoc polymorphism in C++[6].

C++ has traditionally used simple overloading to support ad-hoc polymorphism restricted to monomorphic uses. C++ also supports parametric polymorphism through templates. However, overloading within templates is re-resolved after template instantiation. This means that the combination of overloading and templates provides full ad-hoc polymorphism. Delaying a significant part of type checking until template instantiation increases compilation times and makes error message more difficult to understand.

Concepts provide a disciplined mechanism for full ad-hoc polymorphism through an approach similar to type classes and implicits. Like type classes, a new kind of type is used to constrain parametric type variables. New concepts are defined using a `concept` construct. Classes with the required members

[6]This should not be confused with more recent "concepts lite" proposal, due for inclusion in the next C++ standard

of a concept automatically have an instance for that concept, and further instances can be defined using the `concept_map` construct. Like implicits, concepts cannot be inferred and are not canonical.

Concepts allow overlapping instances, using C++'s complex overloading rules to resolve ambiguities. Concept maps can override the default instance for a type. These features can be useful, but make reasoning about implicit parameters more difficult. Our proposal requires all implicit modules to be explicit and always gives an ambiguity error if multiple matching implicit modules are available.

F#'s static constraints [17] are similar to concepts without support for concept maps.

### 6.5 Modular type classes

Dreyer et al. [4] describe *modular type classes*, a type class system which uses ML module types as type classes and ML modules as type class instances. This system sticks closely to the design of Haskell type classes. In particular it infers type class constraints, and gives ambiguity errors at the point when modules are made implicit.

In order to maintain coherence in the presence of inferred constraints and without canonicity, the system includes a number of undesirable restrictions:

- Modules may only be made implicit at the top-level; they cannot be introduced within a module or a value definition.
- Only module definitions are permitted at the top-level; all value definitions must be contained within a sub-module.
- All top-level module definitions must have an explicit signature.

These restrictions essentially split the language into an outer layer that consists only of module definitions and an inner layer within each module definition. Within the inner layer instances are canonical and constraints are inferred. In the outer layer instances are not canonical and all types must be given explicitly; there is no type inference.

In order to give ambiguity errors at the point where modules are made implicit, one further restriction is required: all implicit modules must define a type named `t` and resolution is always done based on this type.

By basing our design on implicits rather than type classes we avoid such restrictions. Our proposal also includes higher-rank implicit parameters, higher-kinded implicit parameters and resolution based on multiple types. These are not included in the design of modular type classes.

Wehr et al. [19] give a comparison and translation between modules and type classes. This translation does not consider the implicit aspect of type classes, but does illustrate the relationship between type class features (e.g. associated types) and module features (e.g. abstract types).

## 7 Future work

This paper gives only an informal description of the type system and resolution procedure. Giving a formal description is left as future work.

We have created a prototype implementation of our proposal based on OCaml 4.02.0. Further work is needed to bring this prototype up to production quality.

Further work is also needed to answer more practical questions: How well do modular implicits scale to large codebases? How best to design libraries using implicits? How efficient is implicit resolution on real world codebases?

**Acknowledgements**

We would like to thank Stephen Dolan and Anil Madhavapeddy for helpful discussions, Andrew Kennedy for bringing Canonical Structures to our attention, and the anonymous reviewers for their insightful comments.

# References

[1] Manuel M. T. Chakravarty, Gabriele Keller, Simon L. Peyton Jones & Simon Marlow (2005): *Associated types with class*. In Jens Palsberg & Martín Abadi, editors: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, ACM, pp. 1–13, doi:10.1145/1040305.1040306. Available at `http://doi.acm.org/10.1145/1040305.1040306`.

[2] Pierre Chambart & Grégoire Henry (2012): *Experiments in Generic Programming*. OCaml Users and Developers Workshop.

[3] Dominique Devriese & Frank Piessens (2011): *On the bright side of type classes: instance arguments in Agda*. In Manuel M. T. Chakravarty, Zhenjiang Hu & Olivier Danvy, editors: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, ACM, pp. 143–155, doi:10.1145/2034773.2034796. Available at `http://doi.acm.org/10.1145/2034773.2034796`.

[4] Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty & Gabriele Keller (2007): *Modular type classes*. In Martin Hofmann & Matthias Felleisen, editors: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, ACM, pp. 63–70, doi:10.1145/1190216.1190229. Available at `http://doi.acm.org/10.1145/1190216.1190229`.

[5] Jacques Garrigue & Leo White (2014): *Type-level module aliases: independent and equal*. ML Family Workshop.

[6] Douglas Gregor, Jaakko Järvi, Jeremy G. Siek, Bjarne Stroustrup, Gabriel Dos Reis & Andrew Lumsdaine (2006): *Concepts: linguistic support for generic programming in C++*. In Peri L. Tarr & William R. Cook, editors: *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, ACM, pp. 291–310, doi:10.1145/1167473.1167499. Available at `http://doi.acm.org/10.1145/1167473.1167499`.

[7] Mark P. Jones (1995): *A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism*. *J. Funct. Program.* 5(1), pp. 1–35, doi:10.1017/S0956796800001210. Available at `http://dx.doi.org/10.1017/S0956796800001210`.

[8] Simon Peyton Jones, editor (2002): *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/. Available at `http://haskell.org/definition/haskell98-report.pdf`.

[9] Simon Peyton Jones, Mark Jones & Erik Meijer (1997): *Type classes: exploring the design space*. In: *Haskell workshop*, 1997.

[10] Stefan Kaes (1988): *Parametric Overloading in Polymorphic Programming Languages*. In Harald Ganzinger, editor: *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*, *Lecture Notes in Computer Science* 300, Springer, pp. 131–144, doi:10.1007/3-540-19027-9_9. Available at `http://dx.doi.org/10.1007/3-540-19027-9_9`.

[11] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy & Jérôme Vouillon (2014): *The OCaml system release 4.02: Documentation and user's manual*. Interne, Inria. Available at `https://hal.inria.fr/hal-00930213`.

[12] Assia Mahboubi & Enrico Tassi (2013): *Canonical Structures for the Working Coq User*. In Sandrine Blazy, Christine Paulin-Mohring & David Pichardie, editors: *Interactive Theorem Proving*, *Lecture Notes in Computer Science* 7998, Springer Berlin Heidelberg, pp. 19–34, doi:10.1007/978-3-642-39634-2_5. Available at http://dx.doi.org/10.1007/978-3-642-39634-2_5.

[13] *The Rust programming language*. http://www.rust-lang.org.

[14] Bruno C. d. S. Oliveira, Adriaan Moors & Martin Odersky (2010): *Type classes as objects and implicits*. In William R. Cook, Siobhán Clarke & Martin C. Rinard, editors: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, ACM, pp. 341–360, doi:10.1145/1869459.1869489. Available at http://doi.acm.org/10.1145/1869459.1869489.

[15] Bruno C. d. S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee & Kwangkeun Yi (2012): *The implicit calculus: a new foundation for generic programming*. In Jan Vitek, Haibo Lin & Frank Tip, editors: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, ACM, pp. 35–44, doi:10.1145/2254064.2254070. Available at http://doi.acm.org/10.1145/2254064.2254070.

[16] Matthieu Sozeau & Nicolas Oury (2008): *First-Class Type Classes*. In Otmane Aït Mohamed, César A. Muñoz & Sofiène Tahar, editors: *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, *Lecture Notes in Computer Science* 5170, Springer, pp. 278–293, doi:10.1007/978-3-540-71067-7_23. Available at http://dx.doi.org/10.1007/978-3-540-71067-7_23.

[17] Don Syme, Luke Hoban, Tao Liu, Dmitry Lomov, James Margetson, Brian McNamara, Joe Pamer, Penny Orwick, Daniel Quirk, Chris Smith et al. (2005): *The F# 3.0 Language Specification*.

[18] Philip Wadler & Stephen Blott (1989): *How to Make ad-hoc Polymorphism Less ad-hoc*. In: *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, ACM Press, pp. 60–76, doi:10.1145/75277.75283. Available at http://doi.acm.org/10.1145/75277.75283.

[19] Stefan Wehr & Manuel M. T. Chakravarty (2008): *ML Modules and Haskell Type Classes: A Constructive Comparison*. In G. Ramalingam, editor: *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, *Lecture Notes in Computer Science* 5356, Springer, pp. 188–204, doi:10.1007/978-3-540-89330-1_14. Available at http://dx.doi.org/10.1007/978-3-540-89330-1_14.

[20] Jeremy Yallop & Leo White (2014): *Lightweight Higher-Kinded Polymorphism*. In Michael Codish & Eijiro Sumii, editors: *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, *Lecture Notes in Computer Science* 8475, Springer, pp. 119–135, doi:10.1007/978-3-319-07151-0_8. Available at http://dx.doi.org/10.1007/978-3-319-07151-0_8.