Modular implicits

Leo White Frédéric Bour

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. Taking inspiration from Scala's implicits and "Modular Type Classes" by Dreyer et al. [3] 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.

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.

Ad-hoc polymorphism allows the dynamic semantics of a program to be affected by the types of values in that program. Since 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, the following property is important:

> 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 the fundamental property that must hold in a system for ad-hoc polymorphism.

1.1 Type classes

Type classes in Haskell [9] 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
```

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[1]:

```
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:

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

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

Constraints on a value's type can be inferred based on the use of other constrained values in the value'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`.

Haskell's coherence in the presence of inferred type constraints relies on type class instances being *canonical* – for each pair of type and type class there is at most one instance in the entire program. For example, there is at most one instance of `Show Int`, defining two such instances will result in a compiler error. Section 5.2 describes why this property cannot hold in OCaml.

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

1.2 Implicits

Implicits in Scala [7] 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:

```
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, given the following definition in scope:

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

`show` can be called on integers without specifying the second parameter – 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 value explicitly. The need for a value to have an implicit parameter cannot be inferred from the value'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 emit an ambiguity error.

1.3 Modular type classes

Dreyer et al. [3] 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 value can be inferred from the value's definition. Unlike traditional type classes, this system cannot ensure that type class instances are canonical (see Section 5.2). Maintaining coherence in the presence of constraint inference without canonicity requires a number of undesirable restrictions, which are discussed in Section 6.5.

1.4 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 value's implicit parameters must be given explicitly, we are able to avoid the undesirable restrictions of modular type classes.

The rest of this paper is structured as follows: Section 2 describes the definition of functions with implicit module parameters, Section 3 describes how module definitions are made available for selection as implicit parameters, Section 4 describes the search procedure for resolving implicit parameters, Section 5 discusses why modular implicits cannot be canonical, Section 6 discusses related work, and Section 7 discusses future work.

2 Implicit parameters

We propose adding a new kind of function parameter `(implicit M : S)` where `M` is an identifier and `S` is a module type. For example, the generic show function is written as follows:

```
module type Show = sig
  type t
  val show : t -> string
end

let show (implicit S : Show) x = S.show x
```

The type of `show` is written:

```
(implicit S : Show) -> S.t -> string
```

When the `show` function is applied the system attempts to fill-in the implicit `S` parameter by finding a module with the correct type. For example,

```
show [1.2; 2.3; 4.5]
```

will cause the compiler to search for a module with type `Show with type t = float list`, and pass it as the implicit argument to the `show` function.

Note that implicit parameters must always be added to a value explicitly. The need for a value to have an implicit parameter cannot be inferred from the value's definition. Section 5.1 discusses why this is the case.

2.1 Higher-rank implicit parameters

Note that functions with implicit parameters can themselves be used as parameters to functions. For example, the following function:

```
let show_3 (sh : (implicit S : Show) -> S.t -> string) =
  sh 3
```

takes a function `sh` which expects an implicit parameter and applies it too `3`. This can be applied to `show`:

```
show_3 show
```

In the above example the value of the implicit parameter to `show` is looked up in the scope of the *body* of `show_3`. This makes it quite different from the following example:

```
let show_3 (sh : int -> string) = sh 3
[...]
show_3 (fun (x : int) -> show x)
```

where the implicit parameter to `show` is looked up in the scope of the *call* to `show_3`.

This feature means that implicit parameters provide support for higher-rank polymorphism. For example, we can define the following function:

```
let show_stuff (sh : (implicit S : Show) -> S.t -> string) =
  sh 3 ^ " " ^ sh "hello"
```

This function takes a function that expects an implicit parameter `S` of type `Show`, and uses it once with `S.t` equal to `int` and once with `S.t` equal to `string`. This requires `sh` to be polymorphic in `S.t`, which, since `s` is a parameter, constitutes higher-rank polymorphism.

It is worth noting that OCaml already provides some support for higher-rank polymorphism through polymorphic record fields and methods.

2.2 Higher-kinded implicit parameters

Implicit parameters also provide support for higher-kinded polymorphism. For example, given a module type definition for monads:

```
module type Monad = sig
  type 'a t
  val return : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t
end
```

We can define functions that are polymorphic in the monad they operate on:

```
let return (implicit M : Monad) x = M.return x
let bind (implicit M : Monad) m k = M.bind m k
```

which have the following types:

```
val return : (implicit M : Monad) -> 'a -> 'a M.t
val bind : (implicit M : Monad) -> 'a M.t ->
             ('a -> 'b M.t) -> 'b M.t
```

These types are polymorphic in the higher-kinded type constructor `M.t`.

This corresponds to constructor classes in Haskell [5], and would introduce higher-kinded polymorphism into OCaml's core language. Previously, higher-kinded polymorphism was only supported directly in OCaml's verbose module language or indirectly through an encoding based on defunctionalisation [11], so this feature is a genuine improvement in OCaml's expressivity.

2.3 Explicit implicit parameters

As with Scala implicits, searching for an implicit value can fail due to ambiguity (Section 4.1). In such cases, the implicit parameter can be given explicitly using the following syntax:

```
show (implicit ShowInt) 6
```

where `ShowInt` is a module which matches the signature `Show with type t = int`.

3 Implicit declarations

3.1 Implicit modules

Our proposal introduces a new form of module declaration, which makes the module available for use as an implicit parameter. These implicit declarations are regular module definitions annotated with the `implicit` keyword. For example, an implicit module for showing integers could be defined:

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

This declaration both declares a regular module `ShowInt` and makes it available for use in implicit arguments.

Note that only "top-level" implicit modules are considered in scope: if module `N` contains an implicit sub-module `M` then `M` will not be in scope unless `N` has been opened. For convenience, we provide the syntax:

```
open implicit N
```

which brings the implicit sub-modules modules of `N` into the implicit scope.

3.2 Implicit parameters

In addition to implicit module declarations the system also considers any implicit parameters currently in scope. For example

```
let print (implicit S : Show) (x : S.t) =
  print_string (show x)
```

will use the implicit parameter `S` as the implicit parameter of the `show` function. Note that there is guaranteed to be no ambiguity here because the type `S.t` is abstract.

3.3 Implicit functors

Modules for use as implicit arguments can also be constructed using functors. For example, an implicit functor for showing lists could be defined:

```
implicit functor ShowList (S:Show) = struct
  type t = S.t list
  let show l = string_of_list S.show l
end
```

This means that if the system is looking for an implicit module with type `Show with type t = int list`, then it can instead look for an implicit module with type `Show with type t = int`, and apply `ShowList` to it to create the required module.

Such functors can be used multiple times whilst resolving a single implicit parameter, for example

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

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

4 Resolving implicit parameters

4.1 Ambiguity

In order to ensure coherence, we must ensure that uses of implicit parameters are not ambiguous. If there are two implicit modules with type `S` in scope then using a function of type `(implicit M : S) -> ...` is an error. It is not sufficient that the search for implicits finds a solution; it must also ensure that the solution is unique.

4.2 Backtracking

Unlike Haskell type classes, our system considers functor arguments when deciding if a resolution is ambiguous. For example, given a module type and function for printing:

```
module type Print = sig
  type t
  val print : t -> unit
end

let print (implicit P : Print) x = P.print x
```

with the following implicit definitions in scope:

```
implicit module PrintFloatList = struct
  type t = float list
  let print l = print_list print_float l
end

implicit functor PrintShowList (S : Show) = struct
  type t = S.t list
  let print (l : t) =
```

```
    print_list (fun x -> print_string (show x)) l
end
```

Then the ambiguity of `print [1.5]` depends on whether or not there exists an implicit module with type `Show with type t = float`.

In Haskell, these definitions are always considered ambiguous or, with overlapping instances enabled, they are always considered unambiguous. By checking the existence of an implicit module of type `Show with type t = float` to use as the argument `S` of `PrintShowList`, our system is able to decide ambiguity more accurately.

Handling such definitions can be seen as a form of backtracking. The `PrintShowList` functor is selected first but if an implicit module for its argument `S` cannot be found the search backtracks and instead selects `PrintFloatList`.

4.3 Termination

Implicit functors can be used multiple times whilst resolving a single implicit parameter so we must be careful of non-termination in the resolution procedure. For example, a functor which tries to define how to show a type in terms of how to show that type:

```
implicit functor ShowIt (S:Show) = struct
  type t = S.t
  let show = show
end
```

is obviously not well-founded.

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, our 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.

To prevent non-terminating resolution, each recursive use of an implicit functor must be on a module type that is strictly smaller than the module type of the previous use. Strictly smaller is defined pointwise on type definitions within the module type: all type definitions must be smaller or equal, and at least one type definition must be strictly smaller.

For example, if we use a functor to resolve a module of type `Show with type t = int list` then any recursive use of that functor must be to resolve a module of type `Show with type t = int` because that is the only module type which is strictly smaller than the original.

Non-terminating resolutions are really a subclass of ambiguous resolutions: if the resolution procedure does not terminate then we do not know whether there may be multiple possible resolutions of an implicit parameter. For this reason, failure to meet the termination restrictions must be treated as an error, we cannot simply ignore the non-terminating possibilities and continue to look for an alternative resolution.

4.4 Existentials

OCaml modules can contain abstract types. This means that searches can be existentially quantified – we can ask for a type which can be shown (`Show`) rather than how to show a specific type (`Show with type t = int`).

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, consider a module type for arrays and a function to create such arrays:

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

val create : (implicit A:Array) -> A.elem -> int -> A.t
```

where `t` is the type of an array and `elem` is the type of an array element. `create` can be used without specifying the array type being created:

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

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 specialised array types for different element types (e.g. bit vectors to represent arrays of bools).

5 Canonicity

In Haskell you can only define a single instance of a type class for each type. In our system this would be equivalent to only allowing a single implicit module declaration for a given module type within a given program. 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.

5.1 Inference, coherence and canonicity

A key distinction between type classes and implicits is that, with type classes, constraints on a value's type can be inferred based on the use of other constrained values in the value'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, given the following instance:

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

Consider the function:

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

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

```
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.

5.2 Canonicity and abstraction

It would not be possible to preseerve 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.

5.3 A benefit of canonicity

An often cited example of why canonicity is useful is the `union` function for sets in Haskell. The `Ord` type class defines an ordering for a type[2]:

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

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

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 gives 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 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.

5.4 An alternative to canonicity

In terms of our system, Haskell's `union` function would have type:

```
val union: (implicit O : Ord) -> O.t set -> O.t set -> O.t set
```

but without canonicity it is not safe for us to give `union` this type since we cannot ensure 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 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
  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
```

We can use the same approach with implicits by giving our polymorphic set operations the following types:

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

6 Related work

There is a large literature on systematic approaches to ad-hoc polymorphism, Kaes [6] being perhaps the earliest example. We restrict our attention here to a representative sample.

6.1 Type classes

Haskell type classes [9] are the classic formalised method for ad-hoc polymorphism. They have been replicated in a number of other programming languages (e.g. Coq's type classes, Agda's instance arguments, Rust's traits).

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.

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 [7] are an obvious 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 much 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 regular 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 much 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 [8] provides a minimal and general calculus of implicits which could serve as a basis for formalising many aspects of our proposal.

6.3 Canonical structures

In addition to type classes, Coq also supports a mechanism for ad-hoc polymorphism called *canonical structures*. 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.

Canonical structures rely on canonicity, but unlike type classes they 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. [4] describe *concepts*, a system for ad-hoc polymorphism in C++[3].

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. Since much type-checking is delayed until template instantiation, C++ has very slow compilation times and very confusing error messages.

Concepts provide a disciplined approach to 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, and instances of a concept are 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. Concepts also attempt to infer instances for types based on their definition if the required `concept_map` is not defined. These features can be useful, but make reasoning about implicit parameters much more difficult. Our proposal requires all implicit modules to be explicit and always gives an ambiguity error if multiple implicit modules are available.

6.5 Modular type classes

Dreyer et al. [3] 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

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

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. Whilst 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 model on implicits rather than type classes we avoid such restrictions, and the result is a cleaner design.

Our proposal also includes higher-rank implicit parameters, higher-kinded implicit parameters and resolution based on multiple types (the equivalent of multi-parameter type classes). These are not included in the design of modular type classes.

Wehr et al. [10] 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 focuses on the design of modular implicits and the properties of the resolution procedure. It does not describe the details of the type system, and giving a formal description of this type system is left as future work. In particular, the typing of higher-rank and higher-kinded implicit parameters, whilst related to previously published type systems, is novel enough to be worthy of a full formal treatment.

A prototype implementation based on OCaml 4.02.0 has been created and is available through the OPAM package manager[4]. Future work is needed to bring this prototype up to production quality.

Acknowledgements

We would like to thank Jeremy Yallop, Stephen Dolan and Anil Madhavapeddy for helpful discussions and Andrew Kennedy for bringing Canonical Structures to our attention.

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*,

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

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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.