Modular implicits

Leo White Frédéric Bour

August 2, 2014

We propose a system for ad-hoc polymorphism in OCaml based on using modules as type-directed implicit parameters.

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.

Type classes in Haskell [4] have proved an effective mechanism for supporting ad-hoc polymorphism. They use a type-directed implicit parameter passing mechanism to implement constrained polymorphism. Implicits in Scala [3] provide similar capabilities to type classes via direct support for type-directed implicit parameter passing. Chambart et al. have proposed [1] adding support for type-directed implicit parameters to OCaml to support ad-hoc polymorphism.

Taking inspiration from "Modular Type Classes" by Dreyer et al. [2] we describe the design and implementation of a system for ad-hoc polymorphism in OCaml based on passing implicit *module* parameters to functions based on their *module type*. We argue that using modules as the basis for our implicits, rather than regular types, is a natural fit for implementing ad-hoc polymorphism. It also allows us to support advanced type class features such as associated types and constructor classes.

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, we can write the generic print function as follows:

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

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

The type of `print` is written:

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

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

```
print [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 `print` function.

3 Implicit declarations

3.1 Implicit modules

Our proposal also 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 printing integers could be defined:

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

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

3.2 Implicit parameters

In addition to implicit module declarations the system also considers any implicit parameters currently in scope (e.g. `S` inside the body of the function `print` above) to be available for selection as an implicit parameter.

3.3 Implicit functors

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

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

4 Searching for implicits

4.1 Ambiguity

It is important 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.

In cases where the search for an implicit argument would be ambiguous users can instead pass the implicit parameters explicitly. For example:

```
print (implicit ShowInt) 3
```

4.2 Backtracking

Unlike Haskell type classes, our system considers functor arguments when deciding if a search is ambiguous. For example, the ambiguity of the following implicit declarations:

```
implicit module DisplayFoo
  : Display with type t = foo

implicit functor DisplayOfShow (S : Show)
  : Display with type t = S.t
```

depends on whether or not there exists an implicit module with type `Show with type t = foo`.

In standard Haskell, these definitions are always considered ambiguous, or using the `OverlappingInstances` extension they are always considered unambiguous. By checking the existence of a module for argument `S` of `DisplayOfShow`, our system is able to decide ambiguity more accurately.

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

4.3 Termination

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

5 Future work

5.1 Constructor classes

Our current prototype does not support constructor classes. For example:

```
(implicit M : Monad) -> 'a -> 'a M.t
```

is not a valid type at present, since `M.t` is a type constructor. However, extending the system to support constructor classes seems both achievable and highly desirable.

6 Acknowledgements

We would like to thank Jeremy Yallop and Anil Madhavapeddy for helpful discussions.

References

[1] Pierre Chambart and Grégoire Henry. Experiments in generic programming. In *OCaml Users and Developers Workshop*, 2012.

[2] Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. Modular type classes. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 63–70, New York, NY, USA, 2007. ACM.

[3] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, New York, NY, USA, 2010. ACM.

[4] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.