Modular implicits

Leo White Frédéric Bour

Ad-hoc polymorphism

Ad-hoc polymorphism occurs when a function is defined over several different types, acting in a different way for each type.

```
4 + 9
4.5 + 9.5

print [true; false]
print (Some 8.4)
```

Type classes

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

instance Show Int where
  show = showInt

instance Show Float where
  show = showFloat
```

Type classes

```
> show 7
"7"

> show 4.5
"4.5"
```

Coherence

```
> instance Show Int where
    show = "An Int"

<interactive>:2:10:
    Duplicate instance declarations:
      instance Show Int -- Defined at <interactive>:2:10
      instance Show Int -- Defined in ‘GHC.Show’
```

Abstract type equalities

```
module M : sig
  type t
end = struct
  type t = int
end
```

Abstract type equalities

```
module F (X : sig type t val show : t -> string end) =
struct
  instance Show X.t where
    show = X.show
end

instance Show int where
    show = string_of_int

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

Scala implicits

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

def show[T](x: T)(implicit s: Showable[T]) = s.show(x)

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

show(7)
```

Scala implicits

```
implicit object IntShowable2 extends Showable[Int] {
  def show(x: Int) = x.toString
}

show(7)
```

```
error: ambiguous implicit values:
 both object IntShowable2 in object $iw of type
   object IntShowable2
 and object IntShowable in object $iw of type
   object IntShowable
 match expected type Showable[Int]
       show(7)
```

Modular implicits

Implicit *module* parameters to functions chosen by their *module type*.

Implicit parameters

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

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

Implicit parameters

The type of `show` is written:

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

Implicit modules

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

implicit module ShowFloat = struct
  type t = float
  let show = string_of_float
end
```

Implicit modules

```
# show 4;;
- : string = "4"

# show 4.6;;
- : string = "4.6"
```

Implicit parameters

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

```
# let print x =
    print_string (show x);;

Characters 30-34:
    print_string (show x)
                  ^^^^
Error: Ambiguous implicit S: ShowFloat and ShowInt
are both solutions.
```

Implicit scope

```
type foo = Foo

module M = struct
  implicit module ShowFoo = struct
    type t = foo
    let show Foo = "Foo"
  end
end

let () = print Foo (* Error *)
```

Implicit scope

```
type foo = Foo

module M = struct
  implicit module ShowFoo = struct
    type t = foo
    let show Foo = "Foo"
  end
end

open M

let () = print Foo
```

Implicit scope

```
type foo = Foo

module M = struct
  implicit module ShowFoo = struct
    type t = foo
    let show Foo = "Foo"
  end
end

open implicit M

let () = print Foo
```

Implicit functors

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

Implicit functors

```
# show [1; 2; 3];;
- : string = "[ 1, 2, 3 ]"

# show [[5.5]; [1.2; 3.4]];;
- : string = "[ [ 5.5 ], [ 1.2, 3.4 ] ]"
```

Ambiguity

```
implicit module ShowInt1 = struct
  type t = int
  let show = string_of_int
end

implicit module ShowInt2 = struct
  type t = int
  let show _ = "An int"
end
```

Ambiguity

```
# show 9

Characters 0-4:
  show 9
  ^^^^^^
Error: Ambiguous implicit S: ShowInt2 and ShowInt1
are both solutions.
```

Explicit implicit arguments

```
let f x = show (implicit ShowInt) x
```

Termination

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



# show 9

Characters 0-4:
  show 9
  ^^^^^^
Error: Termination check failed when searching for
implicit S.
```

Constructor classes

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

let return (implicit M : Monad) x = M.return x

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

Constructor classes

```
implicit module MonadList = struct
  type 'a t = 'a list
  let return x = [x]
  let bind m k =
    List.fold_right (fun x acc -> k x @ acc) m []
end

# let l = [5; 6; 7] >>= fun x -> return (x + 1);;
val l : int MonadList.t = [6; 7; 8]
```

Constructor classes

```
let when_ (implicit M : Monad) p s : unit M.t =
  if p then s else return ()
```

Higher-rank polymorphism

```
let poly (f : (implicit S : Show)
                              -> S.t -> string) =
    f 5 ^ f 5.5

# poly
    (fun (implicit S : Show) (x : S.t) ->
       "Show: " ^ show x ^ " ");;
- : string = "Show: 5 Show: 5.5 "
```

Associated types

```
module type Graph = sig
  type t
  type vertex
  type edge
  val empty : t
  val add_edge : t -> vertex -> vertex -> t
  val from : t -> vertex -> edge list
end

let empty (implicit G : Graph) () =
  G.empty
let add_edge (implicit G : Graph) g f t =
  G.add_edge g f t
let from (implicit G : Graph) g x =
  G.from g x
```

Associated types

```
implicit module IntGraph =
  MkGraph(struct type t = int end)

# let x = from (add_edge (empty ()) 1 3) 1;;
val x : IntGraph.edge list =
  [{IntGraph.from = 1; to_ = 3}]
```

Status

Working prototype based on OCaml 4.02

- Install it using the OCaml Package Manager (OPAM):

```
$ opam switch 4.02.0+modular-implicits
```

- Try it online (all compiled to javascript and running in the browser):
  http://andrewray.github.io/iocamljs/modimp.html
- When you (inevitably) find bugs, report them to
  http://github.com/ocamllabs/ocaml-modular-implicits

Coherence through functors

```
type ('a, 'b) set

module Mk(O : Ord) = struct
  type o
end

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