Overloading

```
val (=) : {E:EQ} → E.t → E.t → bool
```

Uses for overloading

Equality, **comparison**, **hashing**

```
val (=): 'a → 'a → bool
```

Arithmetic

```
val (+): int → int → int
val (+.): float → float → float
val add : int64 → int64 → int64
...
```

Printing

Why does overloading matter?

Parametric overloading helps **preserve abstraction**

```
module S = Set.Make(String)
S.of_list ["a"; "b"] = S.of_list ["b"; "a"]
   ⇝ false
```

Overloading helps to **abstract over "ad-hoc" behaviour**

e.g. sum a list of numbers

It's **tedious** to do work that the compiler could do

e.g. construct and apply a pretty-printing function

Polymorphism: ad-hoc vs parametric

Parametric polymorphism: uniform behaviour at every type

e.g. behaviour of `map` does not vary with element type

Ad-hoc polymorphism: behaviour varies according to type

e.g. behaviour of `print` should be different for `int` and `bool`

(But today's approach is *compatible* with parametricity.)

Example: arithmetic

```
module type NUM = sig
  type t
  val zero : t
  val add : t → t → t
end
```

Interface

```
implicit module Num_int = struct
  type t = int
  let zero = 0
  let add x y = x + y
end
```

Implicit modules

```
let sum: {N:NUM} → N.t list → N.t =
 fun {N:NUM} l →
  fold_left N.add N.zero l
```

Implicit parameters
(introduction)

```
sum [1;2;3]
sum [1.0;2.0;3.0]
```

Implicit arguments
(elimination)

Implicits, implicit and explicit

Overloaded functions are **parameterised** by per-type behaviour.

```
sum [1;2;3]                sum {Num_int} [1;2;3]

sum [1.0;2.0;3.0]          sum {Num_float} [1.0;2.0;3.0]
```

Implicit functors: overloading for parameterised types

Implicit modules with implicit arguments:

```
implicit module Num_pair{A:NUM}{B:NUM} = struct
  type t = A.t * B.t
  let zero = (A.zero, B.zero)
  let add (a1, b1) (a2, b2) =
    (A1.add a1 a2, B1.add b1 b2)
end
```

```
sum [(10, 1.0); (20, 2.0)]

sum [(100, (10, 1.0)); (200, (20, 2.0))]
```

```
sum {Num_pair{Num_int}{Num_float}}
  [(10, 1.0); (20, 2.0)]
```

Implicits functors: inheritance

FRACTIONAL extends NUM:

```
module type FRACTIONAL = sig
  type t
  module Num : NUM with type t = t
  val div : t → t → t
end
```

Fractional_int extends Num_int:

```
implicit module Fractional_int = struct
  type t = int
  module Num = Num_int
  let div n d = n / d
end
```

Extracting NUM from FRACTIONAL:

```
implicit module Num_fractional{F:FRACTIONAL} = F.Num
```

Mixing NUM and FRACTIONAL:

```
div (add x y) x
```

Finding a match

Calling a function with an implicit argument:

```
add 3 4
```

Beginning the search

```
add {?:NUM with type t = 'a} (3 : 'a) (4 : 'a)
```

Constraining the search

```
add {?:NUM with type t = int} (3 : int) (4 : int)
```

Need `implicit` module that is an **instance** of `NUM with type t = int`

the same types & values

abstract types instantiated

polymorphic types constrained

(+ matching submodules, exception members, etc.)

Ambiguity?

Avoidable ambiguity

```
module type SHOW =
  sig type t val show : t → string end

implicit module Show_bool =
  struct type t = bool let show = string_of_bool end

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

let show {S:SHOW} (x: S.t) = S.show x
let print x = show x
```

Avoidable ambiguity

```
module type SHOW =
  sig type t val show : t → string end

implicit module Show_bool =
  struct type t = bool let show = string_of_bool end

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

let show {S:SHOW} (x: S.t) = S.show x
let print x = show x
```

Solution 1 (**generalize**): `let print {S:SHOW} (x:S.t)= show x`

Solution 2 (**specialize**): `let print (x:int)= show x`

Genuine ambiguity

```
module type SHOW =
  sig type t val show : t → string end

implicit module Show_bool = struct
 type t = bool
 let show = string_of_bool
end

implicit module Show_boolean = struct
 type t = bool
 let show s = Printf.printf "%b" s
end

let show {S:SHOW} (x: S.t) = S.show x
let print (x : bool) = show x
```

Resolving ambiguity: possible heuristics

Pick the **most recent definition**:

```
implicit module Show_bool = struct type t = bool ...
implicit module Show_boolean = struct type t = bool
    ...
```

(`Show_boolean` has priority)

Pick the **best match**:

... but what does "best" mean?

Something else?

~~Ambiguity~~

Ambiguity at the point of **definition**: ✓

```
implicit module Show_bool₁ = struct type t = bool ...
implicit module Show_bool₂ = struct type t = bool ...
```

Ambiguity at the point of **use**: ✗

```
show false
```

Elaboration

Elaboration

Elaboration: translate OCaml+implicits into OCaml

Aims:

understand implicits in terms of existing constructs
simplify implementation

Steps:

1. Resolve and instantiate implicit arguments
2. Translate implicits to packages

Preliminary: packages ("first-class modules")

Package types

```
type t = (module S)
```

Modules ⇝ packages (intro)

```
let p = (module M:S)
```

Packages ⇝ modules (elim)

```
module M = (val p)
```

Sugar: package patterns

```
fun (module M : S) → e
```

sugars

```
fun (m : (module S)) →
    let module M = (val m)
    in e
```

Extra: package constraints

```
(module M:S with type s = 'a
            and type t = int)
```

Preliminary: packages ("first-class modules")

Package types

```
type t = (module S)
```

Modules $\rightsquigarrow$ **packages** (intro)

```
let p = (module M:S)
```

Packages $\rightsquigarrow$ **modules** (elim)

```
module M = (val p)
```

Extra: package constraints

```
(module M:S with type s = 'a
                and type t = int)
```

Sugar: package patterns

```
fun (module M : S) → e
```

sugars

```
fun (m : (module S)) →
   let module M = (val m)
   in e
```

Preliminary: packages ("first-class modules")

Package types

```
type t = (module S)
```

Modules ⇝ packages (intro)

```
let p = (module M:S)
```

Packages ⇝ modules (elim)

```
module M = (val p)
```

Sugar: package patterns

```
fun (module M : S) → e
```

sugars

```
fun (m : (module S)) →
   let module M = (val m)
   in e
```

Extra: package constraints

```
(module M:S with type s = 'a
            and type t = int)
```

First-class modules in action

The `write_c` function accepts a *first-class functor*:

```
module type BINDINGS = functor (F:FOREIGN) → sig end

val write_c: formatter → (module BINDINGS) → unit
```

FFI bindings are defined using a second-class functor:

```
module Bindings(F: FOREIGN) = struct
 open F
 let puts = foreign "puts"
              (string @→ returning int)
end
```

`(module -)` packages the functor:

```
write_c std_formatter (module Bindings)
```

Elaboration, step 1: instantiate explicit arguments

Implicit arguments become explicit arguments:

```
sum [1;2;3]
⇝
   sum {Num_int} [1;2;3]
```

```
sum [[1];[2];[3]]
⇝
   sum {Num_list{Num_int}} [[1];[2];[3]]
```

Elaboration, step 2a: functions into functors (introduction)

Functions with implicit arguments become functor packages:

```
let sum: {N:NUM} → N.t list → N.t =
 fun {N:NUM} l →
  fold_left N.add N.zero l
```

⇝

```
module type SUM_TYPE =
  functor(N:NUM) → sig val v : N.t list → N.t end

let sum : (module SUM_TYPE) =
  (module functor (N:NUM) →
    struct
      let v = fun l → fold_left N.add N.zero l
    end)
```

Elaboration, step 2b: functions into functors (elimination)

Implicit applications become functor package applications:

```
sum {Num_int} [1;2;3]
```

$\rightsquigarrow$

```
let module Sum = (val sum)(Num_int) in
  Sum.v [1;2;3]
```

Implicits and higher kinds

Implicits and higher kinds

Generalizing `map` to arbitrary "container" types:

```
fmap succ [1; 2; 3]
    ⇝ [2; 3; 4]

fmap succ (Some 6)
    ⇝ Some 7
```

```
replace () [1; 2; 3]
    ⇝ [(); (); ()]

replace () (Some "a")
    ⇝ (Some ())
```

Question: what's the **type** of `fmap`? It should generalize

```
val map_list : ('a → 'b) → 'a list → 'b list
val map_option : ('a → 'b) → 'a option → 'b option
```

Implicits and higher kinds (continued)

```
module type FUNCTOR = sig
  type +'a t
  val fmap : ('a → 'b) → 'a t → 'b t
end
```

```
val fmap: {F:FUNCTOR} → ('a → 'b) → 'a F.t → 'b F.t
```

Implicits and higher kinds (continued)

```
module type FUNCTOR = sig
  type +'a t
  val fmap : ('a → 'b) → 'a t → 'b t
end
```

```
let fmap {F:FUNCTOR} f x = F.fmap
let replace {F:FUNCTOR} x c = fmap (fun _ → x) c
```

```
implicit module Functor_option = struct
  type 'a t = 'a option
  let fmap f = function
   | None → None
   | Some v → Some (f v)
end
```

Summary

Implicits support **parametric ad-hoc behaviour**

Ambiguity is prohibited

Implicits elaborate into **first-class functors**

Implicits support **higher-kinded polymorphism**

Next time: monads etc.

$$\gg\!=$$