

# On the design and implementation of Modular Explicits

Samuel Vivien <sup>1</sup>    Didier Rémy <sup>2</sup>    Gabriel Scherer <sup>2</sup>

<sup>1</sup>INRIA & PSL

<sup>2</sup>INRIA

27 Janvier, 2026



## Un peu d'histoire

À son introduction dans OCaml, le langage des modules était stratifié, au dessus du langage noyau

## Un peu d'histoire

À son introduction dans OCaml, le langage des modules était stratifié, au dessus du langage noyau

Depuis OCaml 3.12, l'interaction entre les deux niveaux est rendue possible

## Un peu d'histoire

À son introduction dans OCaml, le langage des modules était stratifié, au dessus du langage noyau

Depuis OCaml 3.12, l'interaction entre les deux niveaux est rendue possible

- Encapsulation d'un module dans une expression: `(module  $M : S$ )`
- Utilisation d'une expression de module: `let module  $\mathcal{X} = (\mathbf{val} m) \mathbf{in} a[\mathcal{X}]$`

## Un peu d'histoire

À son introduction dans OCaml, le langage des modules était stratifié, au dessus du langage noyau

Depuis OCaml 3.12, l'interaction entre les deux niveaux est rendue possible

- Encapsulation d'un module dans une expression: `(module M : S)`
- Utilisation d'une expression de module: `let module X = (val m) in a[X]`

Les modules deviennent des expressions de première classe:

```
module M =  
  (val (if condition then (module M1 : S) else (module M2 : S)))
```

L'implémentation peut être choisie dynamiquement.

## Restriction sur la signatures des modules de première classe

Il faut une annotation!

```
let m = (module Int : Q)
```

## Restriction sur la signatures des modules de première classe

Il faut une annotation!

```
let m = (module Int : Q)
```

Elle peut-être donnée par le contexte

```
let m : (module Q) = (module Int)
```

## Restriction sur la signatures des modules de première classe

Il faut une annotation!

```
let m = (module Int : Q)
```

Elle peut-être donnée par le contexte

```
let m : (module Q) = (module Int)
```

mais pas omise:

```
let m = (module Int)
```

## Restriction sur la signatures des modules de première classe

Il faut une annotation!

```
let m = (module Int : Q)
```

Elle peut-être donnée par le contexte

```
let m : (module Q) = (module Int)
```

mais pas omise:

```
let m = (module Int)
```

Doit être une signature nommée

```
let m =  
  (module Int : sig type t = int end)
```

```
module type I = sig type t = int .. end  
let m = (module Int : I)
```

## Restriction sur la signatures des modules de première classe

Il faut une annotation!

```
let m = (module Int : Q)
```

Elle peut-être donnée par le contexte

```
let m : (module Q) = (module Int)
```

mais pas omise:

```
let m = (module Int)
```

Doit être une signature nommée

```
let m =  
  (module Int : sig type t = int end)
```

```
module type I = sig type t = int .. end  
let m = (module Int : I)
```

Éventuellement avec des contraintes:

```
let m =  
  (module S : Set with type t = int)
```

## Limitations sur le typage des modules de première classe

```
module type Add = sig type t val add : t → t → t end
```

## Limitations sur le typage des modules de première classe

```
module type Add = sig type t val add : t → t → t end
```

```
let double (module M : Add) x = M.add x x
```

## Limitations sur le typage des modules de première classe

```
module type Add = sig type t val add : t → t → t end
```

```
let double (module M : Add) x = M.add x x
```

*Error: This expression has type 'a but an expression was expected of type M.t  
The type constructor M.t would escape its scope*

## Limitations sur le typage des modules de première classe

```
module type Add = sig type t val add : t → t → t end
```

```
let double (module M : Add) x = M.add x x
```

*Error: This expression has type 'a but an expression was expected of type M.t*

*The type constructor M.t would escape its scope*

```
val double : (module Add) → M.t → M.t
```

# Limitations sur le typage des modules de première classe

```
module type Add = sig type t val add : t → t → t end
```

```
let double (module M : Add) x = M.add x x
```

*Error: This expression has type 'a but an expression was expected of type M.t*

*The type constructor M.t would escape its scope*

```
val double : (module Add) → M.t → M.t
```

```
val double : (module M : Add) → M.t → M.t
```

# Modular Explicits

On introduit un nouveau type *flèche* dans le langage des expressions :

**(module**  $\mathcal{X} : S$ )  $\rightarrow \tau$

où  $\mathcal{X}$  peut apparaître dans  $\tau$ .

# Modular Explicits

On introduit un nouveau type *flèche* dans le langage des expressions :

$(\mathbf{module} \mathcal{X} : S) \rightarrow \tau$                       où  $\mathcal{X}$  peut apparaître dans  $\tau$ .

Limitations (syntaxiques):

- Un paramètre doit être un module pour être dépendant.

**fun** (**module**  $\mathcal{X} : S$ )  $\rightarrow$  ...                      ✓

**fun** ( $a$ , (**module**  $\mathcal{X} : S$ ))  $\rightarrow$  ...                      ✗

**fun** ( $m : (\mathbf{module} S)$ )  $\rightarrow$  ...                      ✗

# Modular Explicits

On introduit un nouveau type *flèche* dans le langage des expressions :

**(module  $\mathcal{X} : S$ )**  $\rightarrow \tau$                       où  $\mathcal{X}$  peut apparaître dans  $\tau$ .

Limitations (syntaxiques):

- Un paramètre doit être un module pour être dépendant.

**fun (module  $\mathcal{X} : S$ )**  $\rightarrow \dots$                       ✓

**fun ( $a$ , (module  $\mathcal{X} : S$ ))**  $\rightarrow \dots$                       ✗

**fun ( $m :$  (module  $S$ ))**  $\rightarrow \dots$                       ✗

- L'argument d'une fonction dépendante doit être un module

$f$  (module  $M$ )                      ✓

**let  $m =$  (module  $M : S$ ) in  $f m$**                       ✗

## Règle de typage

AppD

$$\frac{\Gamma \vdash f : (\mathbf{module} \mathcal{X} : Q) \rightarrow \tau \quad \Gamma \vdash M : Q}{\Gamma \vdash f (\mathbf{module} M) : \tau[\mathcal{X} \leftarrow M]}$$

# Règle de typage

AppD

$$\frac{\Gamma \vdash f : (\mathbf{module} \mathcal{X} : Q) \rightarrow \tau \quad \Gamma \vdash M : Q \quad \Gamma \vdash M \rightsquigarrow P}{\Gamma \vdash f (\mathbf{module} M) : \tau[\mathcal{X} \leftarrow P]}$$

# Règle de typage

$$\text{AppD} \quad \frac{\Gamma \vdash f : (\text{module } \mathcal{X} : Q) \rightarrow \tau \quad \Gamma \vdash M : Q \quad \Gamma \vdash M \rightsquigarrow P}{\Gamma \vdash f (\text{module } M) : \tau[\mathcal{X} \leftarrow P]}$$

Cependant, on n'a pas toujours besoin de respecter la condition  $M \rightsquigarrow P$ .

```
let _ = f (module ME)
(* equivalent a *)
let _ = let module X = ME in f (module X)
```

# Règle de typage

$$\text{AppD} \quad \frac{\Gamma \vdash f : (\text{module } \mathcal{X} : Q) \rightarrow \tau \quad \Gamma \vdash M : Q \quad \Gamma \vdash M \rightsquigarrow P}{\Gamma \vdash f (\text{module } M) : \tau[\mathcal{X} \leftarrow P]}$$

Cependant, on n'a pas toujours besoin de respecter la condition  $M \rightsquigarrow P$ .

```
let _ = f (module ME)
(* equivalent a *)
let _ = let module X = ME in f (module X)
```

```
let _ = f (module struct type t = int end)
```

# Règle de typage

$$\text{AppD} \quad \frac{\Gamma \vdash f : (\text{module } \mathcal{X} : Q) \rightarrow \tau \quad \Gamma \vdash M : Q \quad \Gamma \vdash M \rightsquigarrow P}{\Gamma \vdash f (\text{module } M) : \tau[\mathcal{X} \leftarrow P]}$$

Cependant, on n'a pas toujours besoin de respecter la condition  $M \rightsquigarrow P$ .

```
let _ = f (module ME)
(* equivalent a *)
let _ = let module X = ME in f (module X)
```

```
let _ = f (module struct type t = int end)
```

```
let _ = f (module struct type t = A | B end)
```

## Règle de typage

AppD

$$\frac{\Gamma \vdash f : (\mathbf{module} \mathcal{X} : Q) \rightarrow \tau \quad \Gamma \vdash M : Q \quad \Gamma \vdash M \rightsquigarrow P}{\Gamma \vdash f (\mathbf{module} M) : \tau[\mathcal{X} \leftarrow P]}$$

# Règle de typage

$$\text{AppD} \quad \frac{\Gamma \vdash f : (\text{module } \mathcal{X} : Q) \rightarrow \tau \quad \Gamma \vdash M : Q \quad \Gamma \vdash M \rightsquigarrow P}{\Gamma \vdash f (\text{module } M) : \tau[\mathcal{X} \leftarrow P]}$$

Ce n'est pas possible de deviner un type dépendant.

## Règle de typage

$$\text{AppD} \quad \frac{\Gamma \vdash f : \forall \epsilon. (\text{module } \mathcal{X} : Q) \rightarrow^{\epsilon} \tau \quad \Gamma \vdash M : Q \quad \Gamma \vdash M \rightsquigarrow P}{\Gamma \vdash f (\text{module } M) : \tau[\mathcal{X} \leftarrow P]}$$

Ce n'est pas possible de deviner un type dépendant.

On réutilise le mode de principalité comme pour :

- Les méthodes polymorphes
- La surcharge de constructeurs
- Les arguments polymorphes

Un type flèche est connu si son nœud est polymorphe.

# Unification avec types dépendants

# Unification avec types dépendants

L'unification fusionne les deux types afin de propager les informations.

## Unification avec types dépendants

L'unification fusionne les deux types afin de propager les informations.

Et quand on unifie

$$(\mathbf{module} \ \mathcal{X}_1 : S_1) \rightarrow \tau_1$$

et

$$(\mathbf{module} \ \mathcal{X}_2 : S_2) \rightarrow \tau_2$$

on le fait à  $\alpha$ -équivalence près.

## Unification avec types dépendants

L'unification fusionne les deux types afin de propager les informations.

Et quand on unifie

$$(\mathbf{module} \ \mathcal{X}_1 : S_1) \rightarrow \tau_1$$

et

$$(\mathbf{module} \ \mathcal{X}_2 : S_2) \rightarrow \tau_2$$

on le fait à  $\alpha$ -équivalence près.

On doit donc unifier  $\mathcal{X}_1$  avec  $\mathcal{X}_2$ .

## Exemple utilisant modular explicit

## Exemple utilisant modular explicit

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

module ShowInt = struct type t = int ... end

module ShowList (X : Show) = struct
  type t = X.t list
  let show = ...
end
```

## Exemple utilisant modular explicit

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

module ShowInt = struct type t = int ... end

module ShowList (X : Show) = struct
  type t = X.t list
  let show = ...
end

let show (module X : Show) (v : X.t) = X.show v
(* val show : (module X : Show) → X.t → string *)
```

## Exemple utilisant modular explicit

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

module ShowInt = struct type t = int ... end

module ShowList (X : Show) = struct
  type t = X.t list
  let show = ...
end

let show (module X : Show) (v : X.t) = X.show v
(* val show : (module X : Show) → X.t → string *)

let () = print_string (show (module ShowList(ShowInt)) [2; 3; 5])
```

# Émulation de *type classe* en OCaml

En Haskell :

```
show :: Show t => t -> string
```

# Émulation de *type classe* en OCaml

En Haskell :

```
show :: Show t => t -> string
```

Avec modular explicit :

```
val show : (module S : Show) → S.t → string
```

# Émulation de *type classe* en OCaml

En Haskell :

```
show :: Show t => t -> string
```

Avec modular explicit :

```
val show : (module S : Show) → S.t → string
```

Mais aussi possible sans modular explicit :

```
val show : (module Show with type t = 'a) → 'a → string
```

# Émulation de *type classe* en OCaml

En Haskell :

```
show :: Show t => t -> string
```

Avec modular explicit :

```
val show : (module S : Show) → S.t → string
```

Mais aussi possible sans modular explicit :

```
val show : (module Show with type t = 'a) → 'a → string
```

Mais modular explicit est une étape vers modular implicits

```
val show : {S : Show} → S.t → string
```

# Système F dans modular explicit

$$\lambda x : \forall \alpha. \alpha \rightarrow \alpha. x x$$

# Système F dans modular explicit

$$\lambda x : \forall \alpha. \alpha \rightarrow \alpha. x x$$

```
let f x = x x
```

# Système F dans modular explicit

$$\lambda x : \forall \alpha. \alpha \rightarrow \alpha. x x$$

```
let f x = x x
```

```
(* with -rectypes *)
```

```
val f : ('a → 'a) as 'a → 'a
```

# Système F dans modular explicit

$$\lambda x : \forall \alpha. \alpha \rightarrow \alpha. x x$$

```
let f x = x x
```

# Système F dans modular explicit

$$\lambda x : \forall \alpha. \alpha \rightarrow \alpha. x x$$

```
let f x = x x
```

```
module type Type = sig type t end
let f (x : (module A : Type) → A.t → A.t) =
  let module T =
    struct type t = (module A : Type) → A.t → A.t end
  in
  x (module T) x
```

# L'expressivité de $F^\omega$ dans OCaml

# L'expressivité de $F^\omega$ dans OCaml

## Lambda calcul

$$\begin{aligned} \llbracket x \rrbracket &\triangleq x \\ \llbracket e \ e' \rrbracket &\triangleq \llbracket e \rrbracket \llbracket e' \rrbracket \\ \llbracket \lambda x : \sigma. e \rrbracket &\triangleq \lambda x : \sigma. \llbracket e \rrbracket \end{aligned}$$

L'expressivité de  $F^\omega$  dans OCamlLambda calcul  
Système F`module type Type = sig type t end`
$$\begin{aligned} \llbracket x \rrbracket &\triangleq x \\ \llbracket e e' \rrbracket &\triangleq \llbracket e \rrbracket \llbracket e' \rrbracket \\ \llbracket \lambda x : \sigma. e \rrbracket &\triangleq \lambda x : \llbracket \sigma \rrbracket^b. \llbracket e \rrbracket \\ \llbracket \Lambda \alpha : \star. e \rrbracket &\triangleq \text{fun (module } \alpha : \text{Type) } \rightarrow \llbracket e \rrbracket \\ \llbracket e \sigma \rrbracket &\triangleq \llbracket e \rrbracket (\text{module } \llbracket \sigma \rrbracket) \end{aligned}$$

# L'expressivité de $F^\omega$ dans OCaml

Lambda calcul  
 Système F

**module type** Type = **sig type** t **end**

$$\begin{aligned} \llbracket x \rrbracket &\triangleq x \\ \llbracket e \ e' \rrbracket &\triangleq \llbracket e \rrbracket \llbracket e' \rrbracket \\ \llbracket \lambda x : \sigma. e \rrbracket &\triangleq \lambda x : \llbracket \sigma \rrbracket^b. \llbracket e \rrbracket \end{aligned}$$

$$\begin{aligned} \llbracket \sigma \rightarrow \sigma' \rrbracket^b &\triangleq \llbracket \sigma \rrbracket^b \rightarrow \llbracket \sigma' \rrbracket^b \\ \llbracket \forall \alpha : \star. \sigma \rrbracket^b &\triangleq \llbracket \text{module } \alpha : \text{Type} \rrbracket \rightarrow \llbracket \sigma \rrbracket^b \\ \llbracket \alpha \rrbracket^b &\triangleq \alpha.t \end{aligned}$$

$$\begin{aligned} \llbracket \Lambda \alpha : \star. e \rrbracket &\triangleq \llbracket \text{fun (module } \alpha : \text{Type) } \rightarrow \llbracket e \rrbracket \rrbracket \\ \llbracket e \ \sigma \rrbracket &\triangleq \llbracket e \rrbracket (\llbracket \sigma \rrbracket) \end{aligned}$$

# L'expressivité de $F^\omega$ dans OCaml

Lambda calcul  
 Système F

**module type** Type = **sig type** t **end**  
**module** T(**type**  $\alpha$ ) = **struct type** t =  $\alpha$  **end**

$\llbracket x \rrbracket \triangleq x$   
 $\llbracket e e' \rrbracket \triangleq \llbracket e \rrbracket \llbracket e' \rrbracket$   
 $\llbracket \lambda x : \sigma. e \rrbracket \triangleq \lambda x : \llbracket \sigma \rrbracket^b. \llbracket e \rrbracket$   
 $\llbracket \Lambda \alpha : \star. e \rrbracket \triangleq \mathbf{fun (module } \alpha : \mathbf{Type) } \rightarrow \llbracket e \rrbracket$   
 $\llbracket e \sigma \rrbracket \triangleq \llbracket e \rrbracket (\mathbf{module } \llbracket \sigma \rrbracket)$

$\llbracket \sigma \rightarrow \sigma' \rrbracket^b \triangleq \llbracket \sigma \rrbracket^b \rightarrow \llbracket \sigma' \rrbracket^b$   
 $\llbracket \forall \alpha : \star. \sigma \rrbracket^b \triangleq (\mathbf{module } \alpha : \mathbf{Type}) \rightarrow \llbracket \sigma \rrbracket^b$   
 $\llbracket \sigma : \star \rrbracket^b \triangleq \llbracket \sigma \rrbracket.t$   
 $\llbracket \alpha \rrbracket \triangleq \alpha$   
 $\llbracket \sigma : \star \rrbracket \triangleq \mathbf{T(type } \llbracket \sigma \rrbracket^b)$

# L'expressivité de $F^\omega$ dans OCaml

Lambda calcul

**Système F**

$F^\omega$

**module type** Type = **sig type** t **end**  
**module** T(**type**  $\alpha$ ) = **struct type** t =  $\alpha$  **end**

$\llbracket x \rrbracket \triangleq x$   
 $\llbracket e e' \rrbracket \triangleq \llbracket e \rrbracket \llbracket e' \rrbracket$   
 $\llbracket \lambda x : \sigma . e \rrbracket \triangleq \lambda x : \llbracket \sigma \rrbracket^b . \llbracket e \rrbracket$

$\llbracket \Lambda \alpha : \kappa . e \rrbracket \triangleq \mathbf{fun} \ (\mathbf{module} \ \alpha : \llbracket \kappa \rrbracket) \ \rightarrow \llbracket e \rrbracket$   
 $\llbracket e \ \sigma \rrbracket \triangleq \llbracket e \rrbracket \ (\mathbf{module} \ \llbracket \sigma \rrbracket)$

$\llbracket \star \rrbracket \triangleq \mathbf{Type}$   
 $\llbracket \kappa \rightarrow \kappa' \rrbracket \triangleq \mathbf{functor} \ (\_ : \llbracket \kappa \rrbracket) \ \rightarrow \llbracket \kappa' \rrbracket$

$\llbracket \sigma \rightarrow \sigma' \rrbracket^b \triangleq \llbracket \sigma \rrbracket^b \rightarrow \llbracket \sigma' \rrbracket^b$   
 $\llbracket \forall \alpha : \kappa . \sigma \rrbracket^b \triangleq (\mathbf{module} \ \alpha : \llbracket \kappa \rrbracket) \ \rightarrow \llbracket \sigma \rrbracket^b$   
 $\llbracket \sigma : \star \rrbracket^b \triangleq \llbracket \sigma \rrbracket . t$

$\llbracket \alpha \rrbracket \triangleq \alpha$   
 $\llbracket \sigma : \star \rrbracket \triangleq \mathbf{T}(\mathbf{type} \ \llbracket \sigma \rrbracket^b)$   
 $\llbracket \lambda \alpha : \kappa . \sigma \rrbracket \triangleq \mathbf{functor} \ (\alpha : \llbracket \kappa \rrbracket) \ \rightarrow \llbracket \sigma \rrbracket$   
 $\llbracket \sigma \ \sigma' \rrbracket \triangleq \llbracket \sigma \rrbracket \ (\llbracket \sigma' \rrbracket)$

## Polymorphisme de kind d'ordre supérieur en Haskell

```
cfmap :: Monad m => (a -> b) -> Collection m a -> Collection m b
cfmap f (Collection a) = Collection (fmap f a)
```

## Polymorphisme de kind d'ordre supérieur en Haskell

```
cfmap :: Monad m => (a -> b) -> Collection m a -> Collection m b
cfmap f (Collection a) = Collection (fmap f a)
```

```
module type Monad = sig
  type 'a t
  val map : ('a → 'b) → 'a t → 'b t
  ...
end
```

## Polymorphisme de kind d'ordre supérieur en Haskell

```
cfmap :: Monad m => (a -> b) -> Collection m a -> Collection m b
cfmap f (Collection a) = Collection (fmap f a)
```

```
module type Monad = sig
  type 'a t
  val map : ('a → 'b) → 'a t → 'b t
  ...
end

let cfmap (module M : Monad) (module A : Type) (module B : Type)
  (f : A.t → B.t) (a : A.t M.t) : B.t M.t = M.map f a

let res = cfmap (module Option) (module Int) (module String)
  string_of_int (Some 3)
```

## Polymorphisme de kind d'ordre supérieur en Haskell

```
cfmap :: Monad m => (a -> b) -> Collection m a -> Collection m b  
cfmap f (Collection a) = Collection (fmap f a)
```

```
module type Monad = sig  
  type 'a t  
  val map : ('a → 'b) → 'a t → 'b t  
  ...  
end  
  
let cfmap (module M : Monad)  
  (f : 'a → 'b) (a : 'a M.t) : 'b M.t = M.map f a  
  
let res = cfmap (module Option)  
  string_of_int (Some 3)
```

# Et on peut encoder Modular Explicit dans OCaml 5.4

# Et on peut encoder Modular Explicit dans OCaml 5.4

`(module  $\mathcal{X} : Q$ )  $\rightarrow \tau \triangleq$  (module (functor ( $\mathcal{X} : Q$ )  $\rightarrow$  sig val value :  $\tau$  end))`

# Et on peut encoder Modular Explicit dans OCaml 5.4

$$\begin{aligned}(\text{module } \mathcal{X} : Q) \rightarrow \tau &\triangleq (\text{module (functor } (\mathcal{X} : Q) \rightarrow \text{sig val value : } \tau \text{ end})) \\ \text{fun (module } \mathcal{X} : Q) \rightarrow a &\triangleq (\text{module (functor } (\mathcal{X} : Q) \rightarrow \text{struct let value = } a \text{ end})) \\ a (\text{module } M) &\triangleq (\text{val } a)(M).\text{value}\end{aligned}$$

## Et on peut encoder Modular Explicit dans OCaml 5.4

$$\begin{aligned}(\text{module } \mathcal{X} : Q) \rightarrow \tau &\triangleq (\text{module (functor } (\mathcal{X} : Q) \rightarrow \text{sig val value : } \tau \text{ end})) \\ \text{fun (module } \mathcal{X} : Q) \rightarrow a &\triangleq (\text{module (functor } (\mathcal{X} : Q) \rightarrow \text{struct let value = } a \text{ end})) \\ a (\text{module } M) &\triangleq (\text{val } a)(M).\text{value}\end{aligned}$$

Et en utilisant la thèse de Clément Blaudeau on peut alors encoder tout ça dans  $F^\omega$ .

## Et on peut encoder Modular Explicit dans OCaml 5.4

$$\begin{aligned}(\text{module } \mathcal{X} : Q) \rightarrow \tau &\triangleq (\text{module (functor } (\mathcal{X} : Q) \rightarrow \text{sig val value : } \tau \text{ end})) \\ \text{fun (module } \mathcal{X} : Q) \rightarrow a &\triangleq (\text{module (functor } (\mathcal{X} : Q) \rightarrow \text{struct let value = } a \text{ end})) \\ a (\text{module } M) &\triangleq (\text{val } a)(M).\text{value}\end{aligned}$$

Et en utilisant la thèse de Clément Blaudeau on peut alors encoder tout ça dans  $F^\omega$ .

On a triché en n'utilisant pas de chemin mais c'est pour des raisons pragmatiques que la restriction existe, pas de sureté.

## Exemple

```
module type S = sig  
  type _ t  
  val singleton : 'a → 'a t  
end
```

```
let f el (module X : S) = X.singleton el
```

```
val f : 'a → (module X : S) → 'a X.t
```

## Exemple

```
module type S = sig
  type _ t
  val singleton : 'a → 'a t
end
```

```
let f el (module X : S) = X.singleton el
```

```
val f : 'a → (module X : S) → 'a X.t
```

v.s.

```
module type S2 = sig
  type t1
  module F : functor (X : S) →
    sig val value : t1 X.t end
end
```

```
let f (type a) (el : a) =
  (module struct
    type t1 = a
    module F = functor (X : S) →
      struct
        let value = X.singleton el
      end
  end : S2 with type t1 = a)
```

## Polymorphic parameters vs Modular explicits

Modular explicits devrait arriver dans OCaml 5.5 en même temps que Polymorphic parameters.

## Polymorphic parameters vs Modular explicits

Modular explicits devrait arriver dans OCaml 5.5 en même temps que Polymorphic parameters.

```
let f (x : 'a. 'a → 'a) = ...  
let f (x : (module A : Type) → A.t → A.t) = ...
```

## Polymorphic parameters vs Modular explicits

Modular explicits devrait arriver dans OCaml 5.5 en même temps que Polymorphic parameters.

```
let f (x : 'a. 'a → 'a) = ...  
let f (x : (module A : Type) → A.t → A.t) = ...
```

```
let f_2 (x : 'a. 'a → 'b. 'b → 'a * 'b) = ...
```

```
let f_2 (x : (module A : Type) → A.t →  
          (module B : Type) → B.t → A.t * B.t) = ...
```

## Extensions futures

D'autres extensions sont encore nécessaires afin de rendre les modules plus ergonomiques:

## Extensions futures

D'autres extensions sont encore nécessaires afin de rendre les modules plus ergonomiques:

- Signatures paramétriques :  $(t \text{ Eq})$  à la place de  $(\text{Eq with type } t = t)$

## Extensions futures

D'autres extensions sont encore nécessaires afin de rendre les modules plus ergonomiques:

- Signatures paramétriques :  $(t \text{ Eq})$  à la place de  $(\text{Eq with type } t = t)$
- Foncteur avec un type comme argument : **functor** (**type** a)  $\rightarrow \dots$

## Extensions futures

D'autres extensions sont encore nécessaires afin de rendre les modules plus ergonomiques:

- Signatures paramétriques :  $(t \text{ Eq})$  à la place de  $(\text{Eq with type } t = t)$
- Foncteur avec un type comme argument : **functor (type a) → ...**
- Modular implicits

## Modular implicits

Cf : Leo White, Frédéric Bour & Jeremy Yallop : Modular Implicits (2014!)

```
module type Show = sig
  type t
  val show : t → string
end
module implicit ShowInt = struct type t = int ... end
module implicit ShowList (X : Show) = struct
  type t = X.t list
  let show = ...
end
```

# Modular implicits

Cf : Leo White, Frédéric Bour & Jeremy Yallop : Modular Implicits (2014!)

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

module implicit ShowInt = struct type t = int ... end

module implicit ShowList (X : Show) = struct
  type t = X.t list
  let show = ...
end
```

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

# Modular implicits

Cf : Leo White, Frédéric Bour & Jeremy Yallop : Modular Implicits (2014!)

```
module type Show = sig
  type t
  val show : t → string
end
module implicit ShowInt = struct type t = int ... end
module implicit ShowList (X : Show) = struct
  type t = X.t list
  let show = ...
end
```

```
let show {X : Show} (v : X.t) = X.show v
let () = print_string (show [2; 3; 5])
```

## Modular implicits

Cf : Leo White, Frédéric Bour & Jeremy Yallop : Modular Implicits (2014!)

```
module type Show = sig
  type t
  val show : t → string
end
module implicit ShowInt = struct type t = int ... end
module implicit ShowList (X : Show) = struct
  type t = X.t list
  let show = ...
end
```

```
let show {X : Show} (v : X.t) = X.show v
let () = print_string (show [2; 3; 5])
let () = print_string (show {ShowIntList2} [2; 3; 5])
```

# Conclusion

Modular explicit est une nouvelle fonctionnalité qui n'améliore pas l'expressivité mais qui améliore grandement la concision.

Et même si cette fonctionnalité est plus importante que prévue elle reste un point d'étape.

## Des questions ?

PR : #13275

Pour tester : <https://github.com/samsa1/modular-compiler-variants>

```
opam repo add modular-variants \  
  git+https://github.com/samsa1/modular-compiler-variants.git  
  
opam switch create 5.2.0+modular-explicits \  
  --repos modular-variants, default
```

<https://hal.science/hal-05428136>



```
module type Eq = sig
  type t
  val eq : t → t → bool
end

module type Ord = sig
  type t
  val cmp : t → t → int
  module Eq : Eq with type t = t
end

module type Ord' = sig
  type t
  val cmp : t → t → int
  include Eq with type t := t
end
```