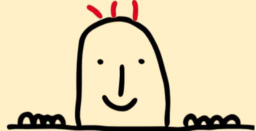
A crash course on the OCaml 🐫 **module language**

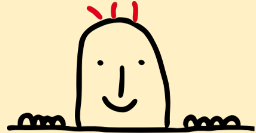
Xavier Van de Woestyne - **Tarides** - https://xvw.lol/en



Thanks to **Stefan Wehr**, my reviewer, for his patience with my *chaotic approach* 🫣 !

A crash course on the OCaml 🐫 **module language**

Xavier Van de Woestyne - **Tarides** - https://xvw.lol/en



Thanks to **Stefan Wehr**, my reviewer, for his patience with my *chaotic approach* 🫣 !

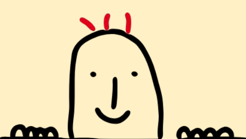
A crash course on the OCaml 🐫 **module language**

Xavier Van de Woestyne - **Tarides** - https://xvw.lol/en

We are heavily involved in the **maintenance** and **development** of the OCaml language (and its ecosystem).

If you have any projects, **let's chat!**

(You can call us **Well-subtyped**, the OCaml consultants)



Thanks to **Stefan Wehr**, my reviewer, for his patience with my *chaotic approach* 🫣 !

A crash course on the OCaml 🐫 **module language**

Xavier Van de Woestyne - **Tarides** - https://xvw.lol/en

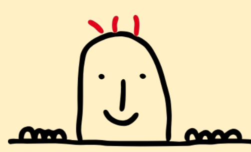
We have experience in **software design**, **distributed systems**, **formal methods**, **build systems**, **editor tooling** and **virtualization**.

...

We are heavily involved in the **maintenance** and **development** of the OCaml language (and its ecosystem).

If you have any projects, **let's chat!**

(You can call us **Well-subtyped**, the OCaml consultants)



Thanks to **Stefan Wehr**, my reviewer, for his patience with my *chaotic approach* 🫣 !

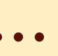
A crash course on the OCaml 🐫 **module language**

Xavier Van de Woestyne - **Tarides** - https://xvw.lol/en

We have experience in **software design**, **distributed systems**, **formal methods**, **build systems**, **editor tooling** and **virtualization**.

We are heavily involved in the **maintenance** and **development** of the OCaml language (and its ecosystem).

If you have any projects, **let's chat!**



Hannes will talk about it seriously at **5pm**!

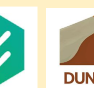
Thanks to **Stefan Wehr**, my reviewer, for his patience with my *chaotic approach* 🫣 !

Double mention to Mirage, which was made possible in its early days by the **flexibility of the module language**.

A crash course on the OCaml 🐫 **module language**

Xavier Van de Woestyne - **Tarides** - https://xvw.lol/en

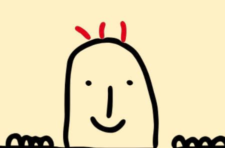
We have experience in **software design**, **distributed systems**, **formal methods**, **build systems**, **editor tooling** and **virtualization**.

...


We are heavily involved in the **maintenance** and **development** of the OCaml language (and its ecosystem).

If you have any projects, **let's chat!**

Hannes will talk about it seriously at **5pm**!



Why this presentation & warnings

The **OCaml Module Language is very cool!**
A very old subject and **a lot of exciting recent work**

Why this presentation & warnings

The **OCaml Module Language is very cool!**
A very old subject and **a lot of exciting recent work**

Why this presentation & warnings

This is **not a tutorial on OCaml** *(unfortunately)*, so
we won't dwell on syntax

(and I will not try to convince you that abstraction is good, we already know that).

THE GOAL IS TO SHARE MY PASSION!

The **OCaml Module Language is very cool!**
A very old subject and **a lot of exciting recent work**

Why this presentation & warnings

This is **not a tutorial on OCaml** *(unfortunately)*, so we won't dwell on syntax

(and I will not try to convince you that abstraction is good, we already know that).

THE GOAL IS TO SHARE MY PASSION!

We will briefly overview the module language and look at some use cases, before concluding with a look at work in progress.

The **OCaml Module Language is very cool!**
A very old subject and **a lot of exciting recent work**

Where I began my apprenticeship on the modules

Why this presentation & warnings

This is **not a tutorial on OCaml** *(unfortunately)*, so we won't dwell on syntax

(and I will not try to convince you that abstraction is good, we already know that).

THE GOAL IS TO SHARE MY PASSION!

We will briefly overview the module language and look at some use cases, before concluding with a look at work in progress.

Understanding and Evolving
the ML Module System

Derek Dreyer

May 2005
CMU-CS-05-131

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

🐫OCaml

An **ML language**, **Strict**,,**Functional**, **Imperative**, with a
powerful type system (ADTs, GADTs and row polymorphism),
Type inference, **Advanced module system**,
OOP with structural subtyping
and **user defined effect** (as core language feature)
Safe, Expressive and Efficient !

An **ML language**, **Strict**,,**Functional**, **Imperative**, with a **powerful type system** (ADTs, GADTs and row polymorphism), **Type inference**, **Advanced module system**, **OOP** with structural subtyping and **user defined effect** (as core language feature) **Safe, Expressive and Efficient !**

Carine & **Sudha** are presenting a tutorial on OCaml Multicore at **1pm**!

An **ML language**, **Strict**,,**Functional**, **Imperative**, with a **powerful type system** (ADTs, GADTs and row polymorphism), **Type inference**, **Advanced module system**, **OOP** with structural subtyping and **user defined effect** (as core language feature) **Safe, Expressive and Efficient !**

Carine & **Sudha** are presenting a tutorial on OCaml Multicore at **1pm**!

In general, I use OCaml for all of my projects. **Not because I like OCaml**, just because it is a very decent choice for **a lot of things!**

Expressivity, Safety, Elegance, Good Tooling, Good Libraries **and great success stories**!

An **ML language**, **Strict**,, **Functional**, **Imperative**, with a **powerful type system** (ADTs, GADTs and row polymorphism), **Type inference**, **Advanced module system**, **OOP** with structural subtyping and **user defined effect** (as core language feature)
Safe, Expressive and Efficient !

Carine & **Sudha a**re presenting a tutorial on OCaml Multicore at **1pm**!

You don't believe me?
Paul-Elliot presents a component of its full-feature presentation software, written entirely in OCaml at **11:05 am**!

In general, I use OCaml for all of my projects.
Not because I like OCaml, just because it is a very decent choice for **a lot of things!**

Expressivity, Safety, Elegance, Good Tooling, Good Libraries **and great success stories**!

An **ML language**, **Strict**,,**Functional**, **Imperative**, with a **powerful type system** (ADTs, GADTs and row polymorphism), **Type inference**, **Advanced module system**, **OOP** with structural subtyping and **user defined effect** (as core language feature) **Safe, Expressive and Efficient !**

Carine & **Sudha** are presenting a tutorial on OCaml Multicore at **1pm**!

You don't believe me?
Paul-Elliot presents a component of its full-feature presentation software, written entirely in OCaml at **11:05 am**!

In general, I use OCaml for all of my projects.
Not because I like OCaml, just because it is a very decent choice for **a lot of things!**

Expressivity, Safety, Elegance, Good Tooling, Good Libraries **and great success stories**!

A very **sweet spot** between **Haskell** and **Rust** (or **OxCaml**)!
And you can add more safety *because of a good tooling*.

An **ML language**, **Strict**,,**Functional**, **Imperative**, with a
powerful type system (ADTs, GADTs and row polymorphism),
Type inference, **Advanced module system**,
OOP with structural subtyping
and **user defined effect** (as core language feature)
Safe, Expressive and Efficient !

Carine & **Sudha** are presenting a tutorial on
OCaml Multicore at **1pm**!

You don't believe me?
Paul-Elliot presents a component of
its full-feature presentation software,
written entirely in OCaml at **11:05 am**!

In general, I use OCaml for all of my projects.
Not because I like OCaml, just because it is a very decent choice
for **a lot of things!**

Expressivity, Safety, Elegance, Good Tooling, Good Libraries
and great success stories!

A very **sweet spot** between **Haskell** and **Rust** (or **OXCaml**)!
And you can add more safety *because of a good tooling.*

You don't believe me?
Nicolas talks about test generation
based on formal specifications at
1.50pm!

An **ML language**, **Strict**,, **Functional**, **Imperative**, with a **powerful type system** (ADTs, GADTs and row polymorphism), **Type inference**, **Advanced module system**, **OOP** with structural subtyping and **user defined effect** (as core language feature) **Safe, Expressive and Efficient !**

Let's talk about that

Carine & **Sudha** are presenting a tutorial on OCaml Multicore at **1pm**!

You don't believe me?
Paul-Elliot presents a component of its full-feature presentation software, written entirely in OCaml at **11:05 am**!

In general, I use OCaml for all of my projects. **Not because I like OCaml**, just because it is a very decent choice for **a lot of things!**

Expressivity, Safety, Elegance, Good Tooling, Good Libraries **and great success stories**!

A very **sweet spot** between **Haskell** and **Rust** (or **OxCaml**)! And you can add more safety *because of a good tooling.*

You don't believe me?
Nicolas talks about test generation based on formal specifications at **1.50pm**!

Modularity

Modularity is the **decomposition** of a system into **components** with **well-defined interfaces** that can be developed, **understood**, and **reused independently**.

Enabling:

- **independent development**
- **reasoning** (structure)
- **reusability**

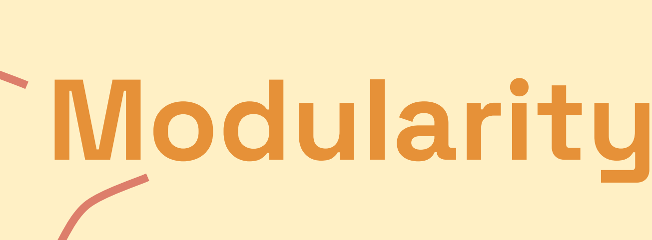
Using **Interfaces**, **Abstraction**, **Namespaces** and **Composition**

Modularity is the **decomposition** of a system into **components** with **well-defined interfaces** that can be developed, **understood**, and **reused independently**.

Enabling:

- **independent development**
- **reasoning** (structure)
- **reusability**

Using **Interfaces**, **Abstraction**, **Namespaces** and **Composition**



Many different approaches to modularity (in differents programming languages)

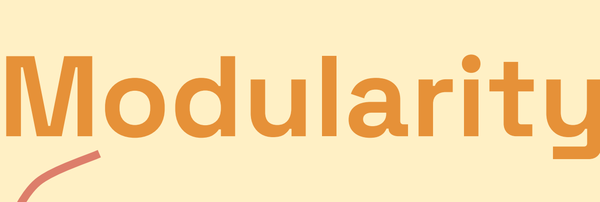
- Using function
- Libraries/packages
- Build Systems (*compilation units*)
- Objects
- Type Classes
- Module System

Modularity is the **decomposition** of a system into **components** with **well-defined interfaces** that can be developed, **understood**, and **reused independently**.

Enabling:

- **independent development**
- **reasoning** (structure)
- **reusability**

Using **Interfaces**, **Abstraction**, **Namespaces** and **Composition**



Many different approaches to modularity (in differents programming languages)

- Using function
- Libraries/packages
- Build Systems (*compilation units*)
- Objects
- Type Classes
- Module System

The **OCaml** (and **SML**) approach!

Modules in OCaml

A module can consist of:

- A collection of types
- A collection of variable bindings
- A set of top-level effects
- A collection of sub-modules

Can be ***sealed*** *by an interface (without an explicit interface, as OCaml is mainly inferred, the typechecker can deduce the interface).*

Modules in OCaml

A module can consist of:

- A collection of types
- A collection of variable bindings
- A set of top-level effects
- A collection of sub-modules

*Can be **sealed** by an interface (without an explicit interface, as OCaml is mainly inferred, the typechecker can deduce the interface).*

Modules are used to describe several things:

- A business unit/A collection of function
- A data-structure

Modules in OCaml

A module can consist of:

- A collection of types
- A collection of variable bindings
- A set of top-level effects
- A collection of sub-modules

*Can be **sealed** by an interface (without an explicit interface, as OCaml is mainly inferred, the typechecker can deduce the interface).*

Every OCaml file **.ml** is an **implementation** and every **.mli** is and **interface**

This enables separate and incremental compilation (making the compilation of ambitious projects very efficient).

But for the presentation, we will focus on modules as a language and not as a compilation tool.

Modules are used to describe several things:

- A business unit/A collection of function
- A data-structure

A first example

```
module Stack = struct
 type 'a t = 'a list

 let empty = []
 let add x stack = x :: stack
 let pop = function x :: _ -> Some x | [] -> None
end
```

A first example

```
module Stack = struct
 type 'a t = 'a list

 let empty = []
 let add x stack = x :: stack
 let pop = function x :: _ -> Some x | [] -> None
end
```

Outside of the module, we can use the **dot-notation** to access the module members:

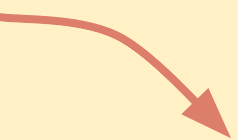
- Stack.empty
- Stack.add 10 (Stack.empty)

A first example

```
module Stack = struct
 type 'a t = 'a list

 let empty = []
 let add x stack = x :: stack
 let pop = function x :: _ -> Some x | [] -> None
end
```

Inferred into



```
module Stack : sig
 type 'a t = 'a list

 val empty : 'a list
 val add : 'a -> 'a list -> 'a list
 val pop : 'a list -> 'a option
end
```

A first example

```
module Stack = struct
 type 'a t = 'a list

 let empty = []
 let add x stack = x :: stack
 let pop = function x :: _ -> Some x | [] -> None
end
```

Inferred into

Our representation is **leaking** !

```
module Stack : sig
 type 'a t = 'a list

 val empty : 'a list
 val add : 'a -> 'a list -> 'a list
 val pop : 'a list -> 'a option
end
```

Implementation detail

Let’s fix that using an Ascription

```
module Stack : sig
  type 'a t

  val empty : 'a t
  val add : 'a -> 'a t -> 'a t
  val pop : 'a t -> 'a option
end = struct
  type 'a t = 'a list

  let empty = []
  let add x stack = x :: stack
  let pop = function x :: _ -> Some x | [] -> None
end
```

We usually split signatures inside `mli` files.

We **hide** the implementation of **t** **making it abstract.**

We can also **deal with encapsulation**

(and it is a great place to put documentation)

Abstraction allows us to enforce invariants

```
module Positive_int : sig
  type t
  val make : int -> t option
end = struct
  type t = int
  let is_positive x = x >= 0
  let make x =
    if is_positive x then Some x
    else None
end
```

Since **t** is abstract, **we can only use make to build a t value.**

(we can also use **private** to make types *read-only*)

Hidden in the **API**

An interesting symmetry

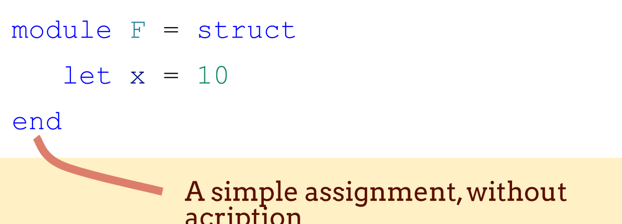
```
let f = 10
```

A simple assignment, without ascription

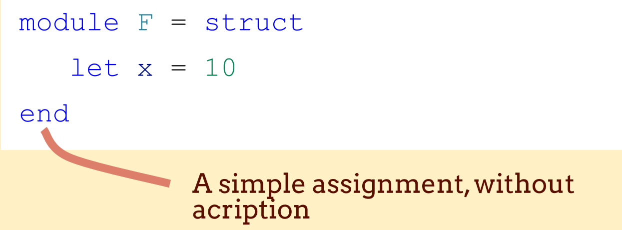
An interesting symmetry

```
let f = 10
```

A simple assignment, without ascription



An interesting symmetry




```
let f : int = 10
```

A simple assignment, **with** ascription

An interesting symmetry

```
module F = struct
   let x = 10
end
```

A simple assignment, without acription

```
let f : int = 10
```

A simple assignment, **with** ascription

```
module F : sig val x : int end =
struct
   let x = 10
end
```

A simple assignment, **with** ascription

An interesting symmetry

```
module F = struct
    let x = 10
end
```

A simple assignment, without acription

```
let f : int = 10
```

A simple assignment, **with** ascription

```
module F : sig val x : int end =
struct
    let x = 10
end
```

A simple assignment, **with** ascription

An ML language generally includes **two** different languages: **the language of values** and t**he language of modules.**

The two languages share similarities (but also differences).

Dissecting module components

```
module F : sig val x : int end =
struct
    let x = 10
end
```

Dissecting module components

```
module F : sig val x : int end =
struct
    let x = 10
end
```

A module **expression**

Dissecting module components

A module **path**

```
module F : sig val x : int end =
struct
    let x = 10
end
```

A module **expression**

Dissecting module components

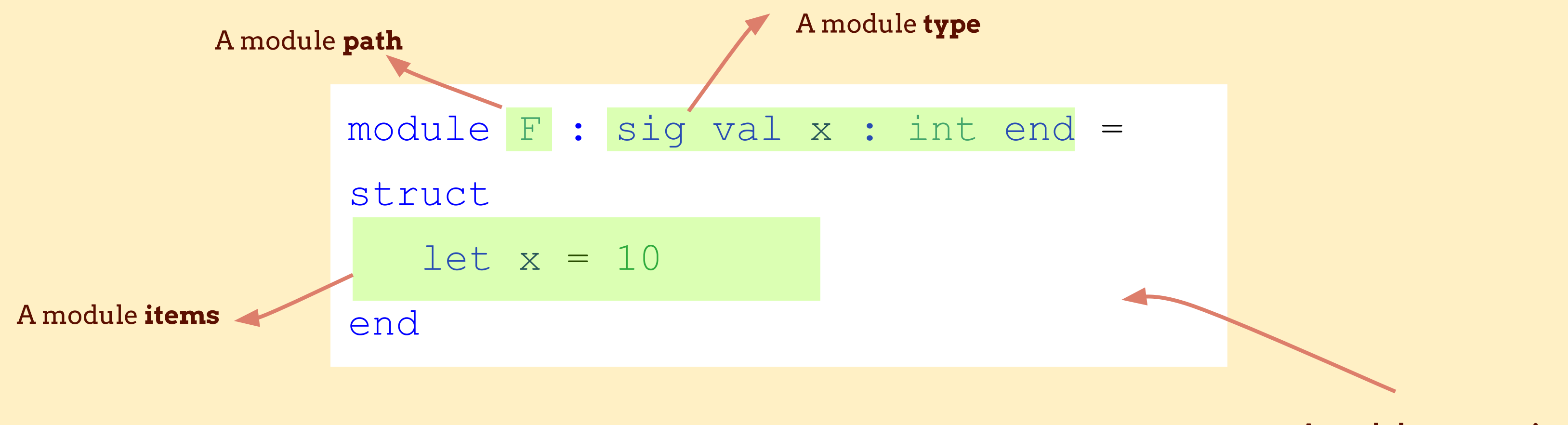
A module **path**

A module **type**

```
module F : sig val x : int end =
struct
    let x = 10
end
```

A module **expression**

Dissecting module components



Reusing interfaces

One of the roles of **modularity is reuse**. Can signatures be **reused**?

```
module Float = struct
 type t = float
 let plus x y = x +. y
 let minus x y = x -. y
 let prod x y = x *. y
 let div x y = x /. y
end
```

```
module Int = struct
 type t = int
 let plus x y = x + y
 let minus x y = x - y
 let prod x y = x * y
 let div x y = x / y
end
```

As with the language of values, where we have **user-defined types.**

Types can be described in the language of modules.

Sharing interfaces

One of the roles of **modularity is reuse**. Can signatures be **shared**?

```
module type Num = sig
 type t
 val plus : t -> t -> t
 val minus : t -> t -> t
 val prod : t -> t -> t
 val div : t -> t -> t
end
```

```
module Float: Num = struct
 type t = float
 let plus x y = x +. y
 let minus x y = x -. y
 let prod x y = x *. y
 let div x y = x /. y
end
```

```
module Int: Num = struct
 type t = int
 let plus x y = x + y
 let minus x y = x - y
 let prod x y = x * y
 let div x y = x / y
end
```

Sharing interfaces

One of the roles of **modularity is reuse**. Can signatures be **shared**?

```
module type Num = sig
 type t
 val plus : t -> t -> t
 val minus : t -> t -> t
 val prod : t -> t -> t
 val div : t -> t -> t
end
```

But **HEY**, it change the semantics of my program. The type **t** is now **abstract!**

```
module Float: Num = struct
 type t = float
 let plus x y = x +. y
 let minus x y = x -. y
 let prod x y = x *. y
 let div x y = x /. y
end
```

```
module Int: Num = struct
 type t = int
 let plus x y = x + y
 let minus x y = x - y
 let prod x y = x * y
 let div x y = x / y
end
```

Sharing interfaces

One of the roles of **modularity is reuse**. Can signatures be **shared**?

But **HEY**, it change the semantics of my program. The type **t** is now **abstract!**

```
module type Num = sig
 type t
 val plus : t -> t -> t
 val minus : t -> t -> t
 val prod : t -> t -> t
 val div : t -> t -> t
end
```

Propagate type equalities (can act on multiple types, and modules **with module F = …**) and can be destructive using **:=**

```
module Float: Num with type t = float
= struct
 type t = float
 let plus x y = x +. y
 let minus x y = x -. y
 let prod x y = x *. y
 let div x y = x /. y
end
```

```
module Int: Num with type t = int
 = struct
 type t = int
 let plus x y = x + y
 let minus x y = x - y
 let prod x y = x * y
 let div x y = x / y
end
```

Sharing interfaces

One of the roles of **modularity is reuse**. Can signatures be **shared**?

We can also **recover** a module type from a module to **lift a module into a module type**.

```
module type TList =
    module type of List
```

But **HEY**, it change the semantics of my program. The type **t** is now **abstract!**

```
module type Num = sig
 type t
 val plus : t -> t -> t
 val minus : t -> t -> t
 val prod : t -> t -> t
 val div : t -> t -> t
end
```

Propagate type equalities (can act on multiple types, and modules **with module F = ...**) and can be destructive using **:=**

```
module Float: Num with type t = float
= struct
 type t = float
 let plus x y = x +. y
 let minus x y = x -. y
 let prod x y = x *. y
 let div x y = x /. y
end
```

```
module Int: Num with type t = int
 = struct
 type t = int
 let plus x y = x + y
 let minus x y = x - y
 let prod x y = x * y
 let div x y = x / y
end
```

So we have :

- **values** (structures)
- **types** (signatures and module-types)

So we have :

- **values** (structures)
- **types** (signatures and module-types)

The next step is obviously: Function !

So we have :

- **values** (structures)
- **types** (signatures and module-types)

The next step is obviously: Function !

In module language, functions are called ... **functors**.

(which **have little to do with category theory**, even though there are somewhat ad hoc justifications for them on the internet.)

So we have :

- **values** (structures)
- **types** (signatures and module-types)

The next step is obviously: Function !

In module language, functions are called ... **functors**.

(which **have little to do with category theory**, even though there are somewhat ad hoc justifications for them on the internet.)

A **Functor** is a function in the **module level** from **structure to structure**

```
module type S = sig
 type t
 val compare : t -> t -> t
end
```

```
module type S = sig
 type t
 val compare : t -> t -> t
end
```

```
module Infix = functor (M : S) -> struct
 let ( > ) x y = M.compare x y > 0
 let ( >= ) x y = M.compare x y >= 0
 let ( < ) x y = M.compare x y < 0
 let ( <= ) x y = M.compare x y <= 0
end
```

```
module type S = sig
 type t
 val compare : t -> t -> t
end
```

```
module Infix = functor (M : S) -> struct
 let ( > ) x y = M.compare x y > 0
 let ( >= ) x y = M.compare x y >= 0
 let ( < ) x y = M.compare x y < 0
 let ( <= ) x y = M.compare x y <= 0
end
```

```
module type S = sig
  type t
  val compare : t -> t -> t
end
```

```
module Infix = functor (M : S) -> struct
  let ( > ) x y = M.compare x y > 0
  let ( >= ) x y = M.compare x y >= 0
  let ( < ) x y = M.compare x y < 0
  let ( <= ) x y = M.compare x y <= 0
end
```

```
module U = Infix (Int)
```

```
module type S = sig
 type t
 val compare : t -> t -> t
end
```

```
module Infix = functor (M : S) -> struct
 let ( > ) x y = M.compare x y > 0
 let ( >= ) x y = M.compare x y >= 0
 let ( < ) x y = M.compare x y < 0
 let ( <= ) x y = M.compare x y <= 0
end
```

Module are structurally subtype
(so we can give a **bigger module**
(a subtype))

```
module U = Infix (Int)
```

```
module type S = sig
 type t
 val compare : t -> t -> t
end
```

```
module Infix = functor (M : S) -> struct
 let ( > ) x y = M.compare x y > 0
 let ( >= ) x y = M.compare x y >= 0
 let ( < ) x y = M.compare x y < 0
 let ( <= ) x y = M.compare x y <= 0
end
```

Module are structurally subtype (so we can give a **bigger module** (a subtype))

```
module U = Infix (Int)
```

The **U** module now has the following signature

```
sig
 val ( > ) : int -> int -> bool
 val ( >= ) : int -> int -> bool
 val ( < ) : int -> int -> bool
 val ( <= ) : int -> int -> bool
end
```

```
module type S = sig
 type t
 val compare : t -> t -> t
end
```

```
module Infix = functor (M : S) -> struct
 let ( > ) x y = M.compare x y > 0
 let ( >= ) x y = M.compare x y >= 0
 let ( < ) x y = M.compare x y < 0
 let ( <= ) x y = M.compare x y <= 0
end
```

Module are structurally subtype
(so we can give a **bigger module**
(a subtype))

```
module U = Infix (Int)
```

The **U** module now has the following signature

```
sig
 val ( > ) : int -> int -> bool
 val ( >= ) : int -> int -> bool
 val ( < ) : int -> int -> bool
 val ( <= ) : int -> int -> bool
end
```

Functors can also be annotated:

```
module F (A : S) (B : S2) : S3 = ...
module F = function (A : S) (B : S2) -> (struct
 ...
end : S3)
```

```
module type S = sig
 type t
 val compare : t -> t -> t
end
```

```
module Infix = functor (M : S) -> struct
 let ( > ) x y = M.compare x y > 0
 let ( >= ) x y = M.compare x y >= 0
 let ( < ) x y = M.compare x y < 0
 let ( <= ) x y = M.compare x y <= 0
end
```

can be simplified in:

```
module Infix (M : S) = struct ... end
```

Module are structurally subtype
(so we can give a **bigger module**
(a subtype))

```
module U = Infix (Int)
```

The **U** module now has the following signature

```
sig
 val ( > ) : int -> int -> bool
 val ( >= ) : int -> int -> bool
 val ( < ) : int -> int -> bool
 val ( <= ) : int -> int -> bool
end
```

Functors can also be annotated:

```
module F (A : S) (B : S2) : S3 = ...
module F = function (A : S) (B : S2) -> (struct
 ...
end : S3)
```

So we have :

- **values** (structures)
- **types** (signatures and module-types)

The next step is obviously: Function !

In module language, functions are called ... **functors**.

(which **have little to do with category theory**, even though there are somewhat ad hoc justifications for them on the internet.)

A **Functor** is a function in the **module level** from **structure to structure**

From structure to structure ? But if I want go from structure to signature ?

So we have :

- **values** (structures)
- **types** (signatures and module-types)

The next step is obviously: Function !

In module language, functions are called ... **functors**.

(which **have little to do with category theory**, even though there are somewhat ad hoc justifications for them on the internet.)

A **Functor** is a function in the **module level** from **structure to structure**

From structure to structure ? But if I want go from structure to signature ? We can't but:

```
module F = functor (M : S) ->
struct
 module type T = sig
   val x : M.t
 end
end
```

So we have :

- **values** (structures)
- **types** (signatures and module-types)

The next step is obviously: Function !

In module language, functions are called ... **functors**.

(which **have little to do with category theory**, even though there are somewhat ad hoc justifications for them on the internet.)

A **Functor** is a function in the **module level** from **structure to structure**

From structure to structure ? But if I want go from structure to signature ? We can't but:

```
module F = functor (M : S) ->
struct
 module type T = sig
   val x : M.t
 end
end
```

this example does demonstrate *a little bit* the **dependent nature** of the OCaml module system

Functors enable higher kinded polymorphism

Higher-kinded polymorphism **quantifies over type constructors**.

Functors enable higher kinded polymorphism

Higher-kinded polymorphism **quantifies over type constructors**.

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

module Monad (R: MONAD) = struct
 type 'a t = 'a R.t
 let return x = R.return x
 let bind x f = R.bind x f

 let map f x = bind x (fun x -> return (f x))
 let ( >>= ) x f = bind f x
 let join x = bind x (fun x -> x)
 (* etc. *)
end
```

Functors enable higher kinded polymorphism

Higher-kinded polymorphism **quantifies over type constructors**.

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

module Monad (R: MONAD) = struct
 type 'a t = 'a R.t
 let return x = R.return x
 let bind x f = R.bind x f

 let map f x = bind x (fun x -> return (f x))
 let ( >>= ) x f = bind f x
 let join x = bind x (fun x -> x)
 (* etc. *)
end
```

Functors enable higher kinded polymorphism

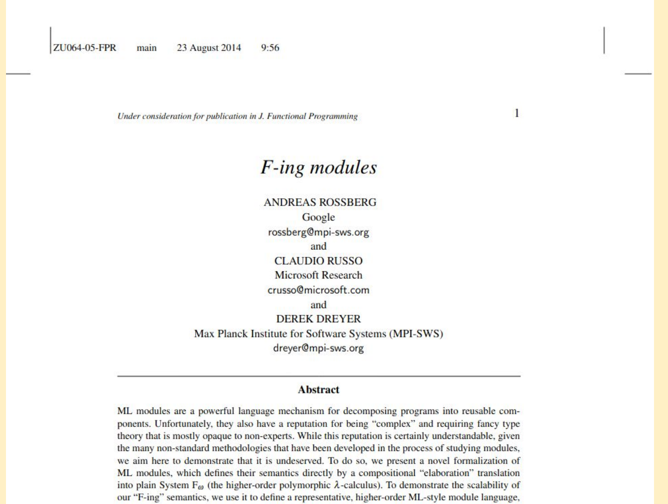
Higher-kinded polymorphism **quantifies over type constructors**.

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

module Monad (R: MONAD) = struct
  type 'a t = 'a R.t
  let return x = R.return x
  let bind x f = R.bind x f

  let map f x = bind x (fun x -> return (f x))
  let ( >>= ) x f = bind f x
  let join x = bind x (fun x -> x)
  (* etc. *)
end
```

In fact, the OCaml module language is a **typed lambda calculus** (System-F Omega), hence the presence of Kinds (and higher kinded types).


ZU064-05-FPR main 23 August 2014 9:56

Under consideration for publication in J. Functional Programming 1

F-ing modules

ANDREAS ROSSBERG
Google
rossberg@mpi-sws.org
and
CLAUDIO RUSSO
Microsoft Research
crusso@microsoft.com
and
DEREK DREYER
Max Planck Institute for Software Systems (MPI-SWS)
dreyer@mpi-sws.org

Abstract

ML modules are a powerful language mechanism for decomposing programs into reusable components. Unfortunately, they also have a reputation for being "complex" and requiring fancy type theory that is mostly opaque to non-experts. While this reputation is certainly understandable, given the many non-standard methodologies that have been developed in the process of studying modules, we aim here to demonstrate that it is undeserved. To do so, we present a novel formalization of ML modules, which defines their semantics directly by a compositional "elaboration" translation into plain System F_ω (the higher-order polymorphic λ-calculus). To demonstrate the scalability of our "F-ing" semantics, we use it to define a representative, higher-order ML-style module language,

So we have :

- **values** (structures)
- **types** (signatures and module-types)
- **Function** (functors)

So we have :

- **values** (structures)
- **types** (signatures and module-types)
- **Function** (functors)

So what's the difference with the value language?

So we have :

- **values** (structures)
- **types** (signatures and module-types)
- **Function** (functors)

So what's the difference with the value language?

Unlike value types, interactions with functions (functors) **require type annotations**. This is because, at the time SML and OCaml described their module languages, the theory was less advanced than it is now.

1ML – Core and Modules United (*F-ing First-class Modules*)

Andreas Rossberg
Google, Germany
rossberg@mpi-sws.org

Abstract

ML is two languages in one: there is the core, with types and expressions, and there are modules, with signatures, structures and functors. Modules form a separate, higher-order functional language on top of the core. There are both practical and technical reasons for this stratification; yet, it creates substantial duplication in syntax and semantics, and it reduces expressiveness. For example, selecting a module cannot be made a dynamic decision. Language extensions allowing modules to be packaged up as first-class values have been proposed and implemented in different variations.

ference [18, 3], and an advanced module system based on concepts from dependent type theory [17]. Although both have contributed to the success of ML, they exist in almost entirely distinct parts of the language. In particular, the convenience of type inference is available only in ML's so-called *core language*, whereas the *module language* has more expressive types, but for the price of being painfully verbose. Modules form a separate language layered on top of the core. Effectively, ML is two languages in one.

This stratification makes sense from a historical perspective. Modules were introduced for programming-in-the-large, when the core language already existed. The dependent type machinery that

So we have :

- **values** (structures)
- **types** (signatures and module-types)
- **Function** (functors)

So what's the difference with the value language?

Unlike value types, interactions with functions (functors) **require type annotations**. This is because, at the time SML and OCaml described their module languages, the theory was less advanced than it is now.

But hey, modules can introduce types, so **we have a mix of types and terms?** How are type equalities propagated in presence of **abstract types**?

1ML – Core and Modules United (*F-ing First-class Modules*)

Andreas Rossberg
Google, Germany
rossberg@mpi-sws.org

Abstract

ML is two languages in one: there is the core, with types and expressions, and there are modules, with signatures, structures and functors. Modules form a separate, higher-order functional language on top of the core. There are both practical and technical reasons for this stratification; yet, it creates substantial duplication in syntax and semantics, and it reduces expressiveness. For example, selecting a module cannot be made a dynamic decision. Language extensions allowing modules to be packaged up as first-class values have been proposed and implemented in different variations. However, they remedy expressiveness only to some extent, are syn-

ference [18, 3], and an advanced module system based on concepts from dependent type theory [17]. Although both have contributed to the success of ML, they exist in almost entirely distinct parts of the language. In particular, the convenience of type inference is available only in ML's so-called *core language*, whereas the *module language* has more expressive types, but for the price of being painfully verbose. Modules form a separate language layered on top of the core. Effectively, ML is two languages in one.

This stratification makes sense from a historical perspective. Modules were introduced for programming-in-the-large, when the core language already existed. The dependent type machinery that

```
module type S = sig
 type t
end
```

```
module type S = sig
 type t
end
```

```
module Make = functor (M: S) -> (
 struct
   type t = T of M.t
   let make x = T x
   let id x = x
 end : sig
   type t
   val make : M.t -> t
   val id : t -> t
end)
```

```
module type S = sig
 type t
end
```

```
module Make = functor (M: S) -> (
 struct
   type t = T of M.t
   let make x = T x
   let id x = x
 end : sig
   type t
   val make : M.t -> t
   val id : t -> t
end)
```

the type **t** is **abstract**

```
module type S = sig
 type t
end
```

```
module Make = functor (M: S) -> (
 struct
   type t = T of M.t
   let make x = T x
   let id x = x
 end : sig
   type t
   val make : M.t -> t
   val id : t -> t
end)
```

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

module A = Make (M)
module B = Make (M)

let a = A.make 10
let b = B.make 11
let c = A.id b
```

the type **t** is **abstract**

```
module type S = sig
 type t
end
```

```
module Make = functor (M: S) -> (
 struct
   type t = T of M.t
   let make x = T x
   let id x = x
 end : sig
   type t
   val make : M.t -> t
   val id : t -> t
end)
```

the type **t** is **abstract**

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

module A = Make (M)
module B = Make (M)

let a = A.make 10
let b = B.make 11
let c = A.id b
```

Here we mixing **A.t** and **B.t**
Does it typecheck?

it **depends on the semantic of functors**

```
module type S = sig
 type t
end
```

```
module Make = functor (M: S) -> (
 struct
   type t = T of M.t
   let make x = T x
   let id x = x
 end : sig
   type t
   val make : M.t -> t
   val id : t -> t
end)
```

the type **t** is **abstract**

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

module A = Make (M)
module B = Make (M)

let a = A.make 10
let b = B.make 11
let c = A.id b
```

Here we mixing **A.t** and **B.t**
Does it typecheck?

it **depends on the semantic of functors**

- If it succeed **functors are Applicative**
  (and it has nothing to do with a Strong Lax Monoid)
- If if fails **functors are Generative**

```
module type S = sig
 type t
end
```

```
module Make = functor (M: S) -> (
 struct
   type t = T of M.t
   let make x = T x
   let id x = x
 end : sig
   type t
   val make : M.t -> t
   val id : t -> t
end)
```

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

module A = Make (M)
module B = Make (M)

let a = A.make 10
let b = B.make 11
let c = A.id b
```

Applicative functors:

Same input = same type.

F(X) = F(X)

Generative functors:

Every application create **fresh types**.

F(X) ≠ F(X)

```ocaml
module type S = sig
 type t
end
```

```ocaml
module Make = functor (M: S) -> (
 struct
   type t = T of M.t
   let make x = T x
   let id x = x
 end : sig
   type t
   val make : M.t -> t
   val id : t -> t
end)
```

```ocaml
module M = struct  type t = int end

module A = Make (M)
module B = Make (M)

let a = A.make 10
let b = B.make 11
let c = A.id b
```

Applicative functors:

Same input = same type.

F(X) = F(X)

Generative functors:

Every application create **fresh types**.

F(X) ≠ F(X)

OCaml Modules

SML Modules

```
module type S = sig
 type t
end
```

```
module Make = functor (M: S) () -> (
 struct
   type t = T of M.t
   let make x = T x
   let id x = x
 end : sig
   type t
   val make : M.t -> t
   val id : t -> t
end)
```

Become Generative

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

module A = Make (M) ()
module B = Make (M) ()

let a = A.make 10
let b = B.make 11
let c = A.id b
```

Applicative functors:

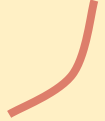
Same input = same type.

F(X) = F(X)

Generative functors:

Every application create **fresh types**.

F(X) ≠ F(X)



OCaml Modules

SML Modules

But also OCaml if we add **()**

Both semantics offer advantages that we will see in the examples.

Tracking type equalities is complicated

When can two modules be considered equivalent?

Tracking type equalities is complicated

When can two modules be considered equivalent?

```
module A = Make (struct type t = int end)
module B = Make (struct type t = int end)
let a = A.make 10
let b = B.make 11
let c = A.id b
```

```
module M = struct  type t = int end
module A = Make (M)
module B = Make (M)
let a = A.make 10
let b = B.make 11
let c = A.id b
```

Tracking type equalities is complicated

When can two modules be considered equivalent?

```
module A = Make (struct type t = int end)
module B = Make (struct type t = int end)
let a = A.make 10
let b = B.make 11
let c = A.id b
```

Fail to typecheck

the modules do not share the same path
(even if they are syntactically identical)

```
module M = struct  type t = int end
module A = Make (M)
module B = Make (M)
let a = A.make 10
let b = B.make 11
let c = A.id b
```

Succeed to typecheck

use Path to clarify module equalities

Tracking type equalities is complicated

When can two modules be considered equivalent?

```
module A = Make (struct type t = int end)
module B = Make (struct type t = int end)
let a = A.make 10
let b = B.make 11
let c = A.id b
```

Fail to typecheck

the modules do not share the same path
(even if they are syntactically identical)

```
module M = struct  type t = int end
module A = Make (M)
module B = Make (M)
let a = A.make 10
let b = B.make 11
let c = A.id b
```

Succeed to typecheck

use Path to clarify module equalities

And also, Applicative functors can hide generative functor...

So we have :

- **values** (structures)
- **types** (signatures and module-types)
- **Function** (functors)

So we have :

- **values** (structures)
- **types** (signatures and module-types)
- **Function** (functors)

The next step is obviously: Recursion!

So we have :

- **values** (structures)
- **types** (signatures and module-types)
- **Function** (functors)

The next step is obviously: Recursion!

OCaml allows you to describe recursive and mutually recursive modules, but their semantics are **incredibly complex** because they are **relatively poorly defined**.

So we have :

- **values** (structures)
- **types** (signatures and module-types)
- **Function** (functors)

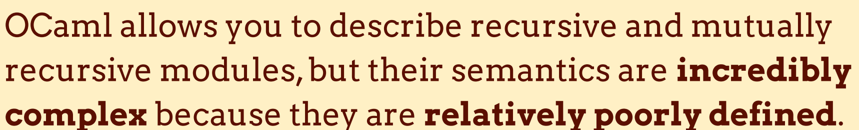
The next step is obviously: Recursion!

OCaml allows you to describe recursive and mutually recursive modules, but their semantics are **incredibly complex** because they are **relatively poorly defined**.

So we have :

- **values** (structures)
- **types** (signatures and module-types)
- **Function** (functors)

The next step is obviously: Recursion!




```ocaml
module A = struct
 (* not allowed *)
 type t = B.t
end

module B = struct
 type t = int
end
```

So we have :

- **values** (structures)
- **types** (signatures and module-types)
- **Function** (functors)

The next step is obviously: Recursion!

OCaml allows you to describe recursive and mutually recursiv[illegible]antics are **incredibly comple**[illegible]**ively poorly defined**.

```
module A = struct
 (* not allowed *)
 type t = B.t
end

module B = struct
 type t = int
end
```

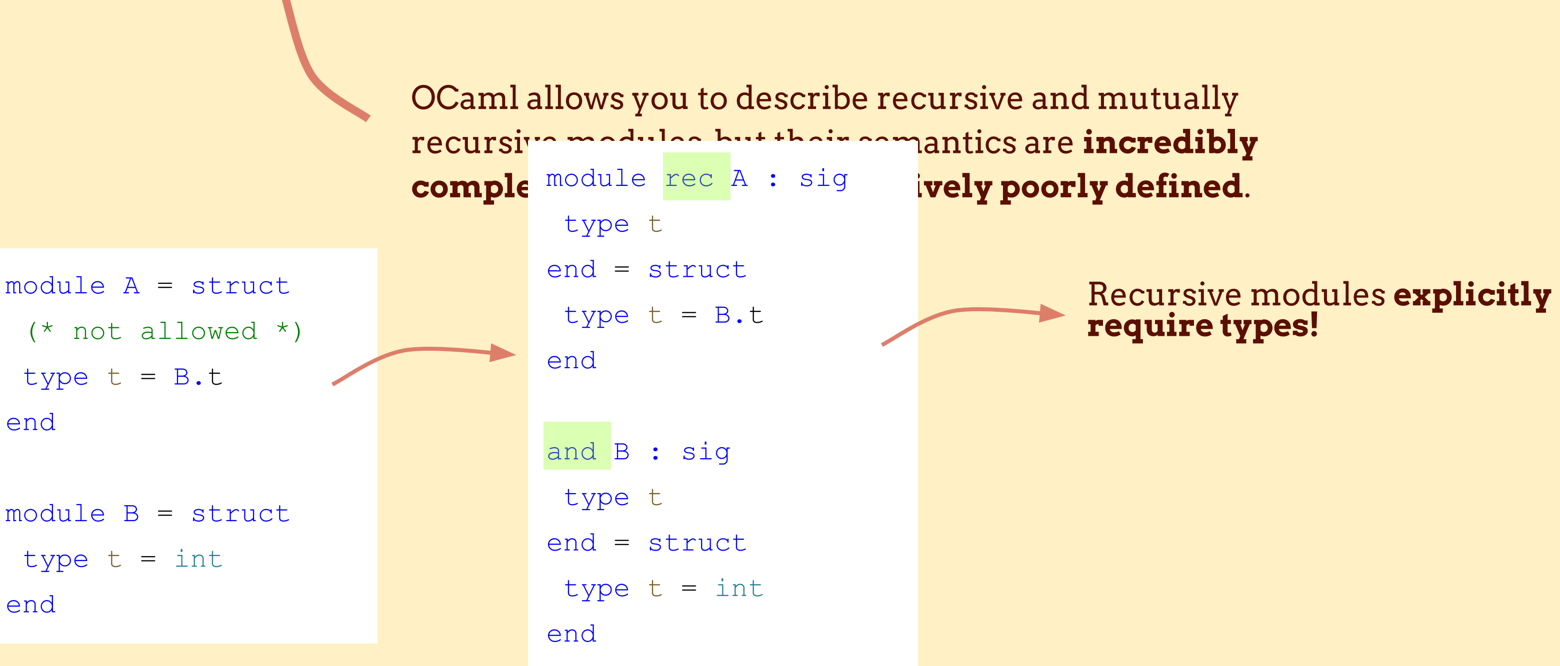
```
module rec A : sig
 type t
end = struct
 type t = B.t
end

and B : sig
 type t
end = struct
 type t = int
end
```

So we have :

- **values** (structures)
- **types** (signatures and module-types)
- **Function** (functors)

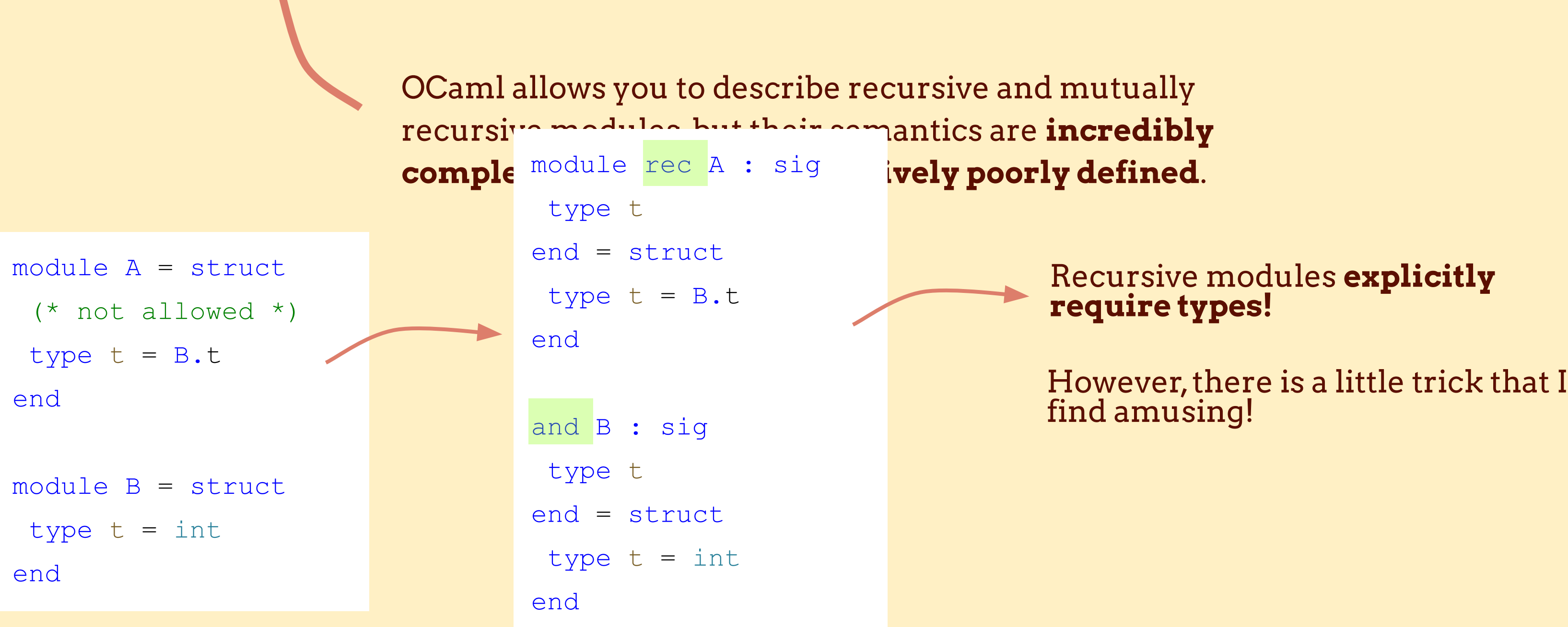
The next step is obviously: Recursion!



So we have :

- **values** (structures)
- **types** (signatures and module-types)
- **Function** (functors)

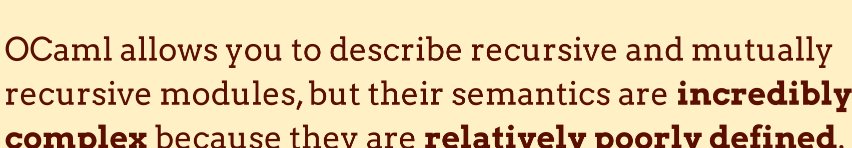
The next step is obviously: Recursion!



So we have :

- **values** (structures)
- **types** (signatures and module-types)
- **Function** (functors)

The next step is obviously: Recursion!



OCaml allows you to describe recursive and mutually recursive modules, but their semantics are **incredibly complex** because they are **relatively poorly defined**.

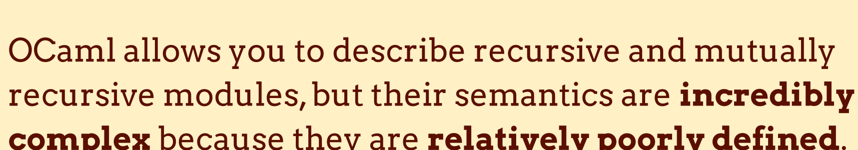
I would like to implement this signature by making the three types concrete.

```
module type S = sig
 type i
 type s
 type f
end
```

So we have :

- **values** (structures)
- **types** (signatures and module-types)
- **Function** (functors)

The next step is obviously: Recursion!



OCaml allows you to describe recursive and mutually recursive modules, but their semantics are **incredibly complex** because they are **relatively poorly defined**.

I would like to implement this signature by making the three types concrete.

```
module type S = sig
 type i
 type s
 type f
end
```

let's use the classical approach

```
module A :
 S with type i = int
    and type s = string
    and type f = float
= struct
 type i = int
 type s = string
 type f = float
end
```

So we have :

- **values** (structures)
- **types** (signatures and module-types)
- **Function** (functors)

The next step is obviously: Recursion!

OCaml allows you to describe recursive and mutually recursive modules, but their semantics are **incredibly complex** because they are **relatively poorly defined**.

I would like to implement this signature by making the three types concrete.

```
module type S = sig
 type i
 type s
 type f
end
```

let's use the classical approach

```
module A :
 S with type i = int
    and type s = string
    and type f = float
= struct
 type i = int
 type s = string
 type f = float
end
```

```
module type A =
 S with type i = int
    and type s = string
    and type f = float
module rec A : A = A
```

It works because the signature completely determines the module's contents

Module extension using inclusion

OCaml allows modules to be included in other modules.

What can be useful for extending an existing module:

Module extension using inclusion

OCaml allows modules to be included in other modules.
What can be useful for extending an existing module:

```
module My_list = struct
 include List
 let print_int_list =
   iter print_int
end
```

Module extension using inclusion

OCaml allows modules to be included in other modules.
What can be useful for extending an existing module:

```
module My_list = struct
 include List
 let print_int_list =
   iter print_int
end
```

My_list has now all definition of **List** and the weird function **print_int_list.**

This is why I can access to **iter** without full qualification.

Module extension using inclusion

OCaml allows modules to be included in other modules.
What can be useful for extending an existing module:

```
module My_list = struct
 include List
 let print_int_list =
   iter print_int
end
```

My_list has now all definition of **List** and the weird function **print_int_list.**

This is why I can access to **iter** without full qualification.

include takes **a module expression** so this is valid:

```
include struct ... end
```

Aside note on strengthening

Strengthening is one of the core mechanisms of the OCaml module type system. It is what allows the compiler to **recover type equalities from module paths**, making the module system usable in practice.

Aside note on strengthening

Strengthening is one of the core mechanisms of the OCaml module type system. It is what allows the compiler to **recover type equalities from module paths**, making the module system usable in practice.

When we use **module type of** (to recover the type of a module) we **lose strengthening**.

To make this module type reusable in many situations, it is intentionally not strengthened: abstract types and datatypes are not explicitly related with the types of the original module. For the same reason, module aliases in the inferred type are expanded.

Aside note on strengthening

Strengthening is one of the core mechanisms of the OCaml module type system. It is what allows the compiler to **recover type equalities from module paths**, making the module system usable in practice.

When we use **module type of** (to recover the type of a module) we **lose strengthening**.

To make this module type reusable in many situations, it is intentionally not strengthened: abstract types and datatypes are not explicitly related with the types of the original module. For the same reason, module aliases in the inferred type are expanded.

But in the case of **extension**, we probably want to **keep the strengthening.**

Fortunately, **there is a sequence of 6 OCaml keywords** that can be used to restore the strengthening:

Aside note on strengthening

Strengthening is one of the core mechanisms of the OCaml module type system. It is what allows the compiler to **recover type equalities from module paths**, making the module system usable in practice.

When we use **module type of** (to recover the type of a module) we **lose strengthening**.

To make this module type reusable in many situations, it is intentionally not strengthened: abstract types and datatypes are not explicitly related with the types of the original module. For the same reason, module aliases in the inferred type are expanded.

But in the case of **extension**, we probably want to **keep the strengthening.**

Fortunately, **there is a sequence of 6 OCaml keywords** that can be used to restore the strengthening:

```
module type MY_LIST = sig
 include module type of struct include List end
 val replace: 'a t -> ('a -> bool) -> 'a -> 'a t
end
```

Module Aliases and Opens

Opening consists of making definitions available in a scope without exporting them, as opposed to including them.

Module Aliases and Opens

Opening consists of making definitions available in a scope without exporting them, as opposed to including them.

```
module L = List (* I can access to List via L. *)

open List (* Globaly open List. *)

let f y =
   let open List in
   (* Here, List is available *)
   map (fun x -> x) y

let g y =
   List.(
       (* Here, List is available at
          the expression-level. *)
       map (fun x -> x) y
   )
```

Module Aliases and Opens

Opening consists of making definitions available in a scope without exporting them, as opposed to including them.

```
module L = List (* I can access to List via L. *)

open List (* Globaly open List. *)

let f y =
   let open List in
   (* Here, List is available *)
   map (fun x -> x) y

let g y =
   List.(
       (* Here, List is available at
          the expression-level. *)
       map (fun x -> x) y
   )
```

Like **include**, open take arbitrary module expression.

```
open struct ... end
```

Module Aliases and Opens

Opening consists of making definitions available in a scope without exporting them, as opposed to including them.

```
module L = List (* I can access to List via L. *)

open List (* Globaly open List. *)

let f y =
   let open List in
   (* Here, List is available *)
   map (fun x -> x) y

let g y =
   List.(
       (* Here, List is available at
          the expression-level. *)
      map (fun x -> x) y
   )
```

Like **include**, open take arbitrary module expression.

```
open struct ... end
```

Encapsulation without **mli**

```
open struct
  let x = 10 (* x exists but hidden*)
end
```

Module Aliases and Opens

Opening consists of making definitions available in a scope without exporting them, as opposed to including them.

```
module L = List (* I can access to List via L. *)

open List (* Globaly open List. *)

let f y =
   let open List in
   (* Here, List is available *)
   map (fun x -> x) y

let g y =
   List.(
      (* Here, List is available at
         the expression-level. *)
      map (fun x -> x) y
   )
```

Like **include**, open take arbitrary module expression.

```
open struct ... end
```

Encapsulation without **mli**

```
open struct
   let x = 10 (* x exists but hidden*)
end
```

That can **reach Anchoring issues**
Captured by the typechecker

```
open struct
 type t = A | B
end
let y = A
```

Interacting with the value-level

Even though we presented the value language and the module language as two distinct languages, OCaml has **two analogous mechanisms for interacting with the value language.**

Interacting with the value-level

Even though we presented the value language and the module language as two distinct languages, OCaml has **two analogous mechanisms for interacting with the value language.**

since 5.5.0

Module-dependent function

functions **parameterized over a module**

Interacting with the value-level

Even though we presented the value language and the module language as two distinct languages, OCaml has **two analogous mechanisms for interacting with the value language.**

since 5.5.0 Module-dependent function

functions **parameterized over a module**

```
let sort (module C: comparable) li =
 List.sort (C.compare) li
```

Interacting with the value-level

Even though we presented the value language and the module language as two distinct languages, OCaml has **two analogous mechanisms for interacting with the value language.**

since 5.5.0 Module-dependent function

functions **parameterized over a module**

```
let sort (module C: comparable) li =
 List.sort (C.compare) li
```

The module is
known statically

- Reduce the need for funcors (that can be heavy
- Allows a form of **Higher kinded polymorphism**

Interacting with the value-level

Even though we presented the value language and the module language as two distinct languages, OCaml has **two analogous mechanisms for interacting with the value language.**

since 5.5.0 Module-dependent function

functions **parameterized over a module**

```
let sort (module C: comparable) li =
 List.sort (C.compare) li
```

The module is
known statically

- Reduce the need for funcors (that can be heavy
- Allows a form of **Higher kinded polymorphism**

```
let map (module M: MONAD)
       (f: 'a -> 'b) (x: 'a M.t) : 'b M.t =
 M.bind x (fun x -> M.return (f x))
```

Interacting with the value-level

Even though we presented the value language and the module language as two distinct languages, OCaml has **two analogous mechanisms for interacting with the value language.**

since 5.5.0 Module-dependent function

functions **parameterized over a module**

```
let sort (module C: comparable) li =
 List.sort (C.compare) li
```

The module is
known statically

- Reduce the need for funcors (that can be heavy
- Allows a form of **Higher kinded polymorphism**

```
let map (module M: MONAD)
        (f: 'a -> 'b) (x: 'a M.t) : 'b M.t =
 M.bind x (fun x -> M.return (f x))
```

First-class module

build **dynamic module based on runtime value**

Interacting with the value-level

Even though we presented the value language and the module language as two distinct languages, OCaml has **two analogous mechanisms for interacting with the value language.**

since 5.5.0 Module-dependent function

functions **parameterized over a module**

```
let sort (module C: comparable) li =
 List.sort (C.compare) li
```

The module is **known statically**

- Reduce the need for funcors (that can be heavy
- Allows a form of **Higher kinded polymorphism**

```
let map (module M: MONAD)
        (f: 'a -> 'b) (x: 'a M.t) : 'b M.t =
 M.bind x (fun x -> M.return (f x))
```

First-class module

build **dynamic module based on runtime value**

```
let f x =
 let module T = struct
    let f () =
      print_endline x
 end
in (module T : S)
```

Interacting with the value-level

Even though we presented the value language and the module language as two distinct languages, OCaml has **two analogous mechanisms for interacting with the value language.**

since 5.5.0 Module-dependent function

functions **parameterized over a module**

```
let sort (module C: comparable) li =
 List.sort (C.compare) li
```

The module is **known statically**

- Reduce the need for funcors (that can be heavy
- Allows a form of **Higher kinded polymorphism**

```
let map (module M: MONAD)
      (f: 'a -> 'b) (x: 'a M.t) : 'b M.t =
 M.bind x (fun x -> M.return (f x))
```

First-class module

build **dynamic module based on runtime value**

```
let f x =
 let module T = struct
    let f () =
      print_endline x
 end
in (module T : S)
```

The module can be defined as dependent on **runtime** value.

(And we can also apply functor)

Interacting with the value-level

Even though we presented the value language and the module language as two distinct languages, OCaml has **two analogous mechanisms for interacting with the value language.**

since 5.5.0 Module-dependent function

functions **parameterized over a module**

```
let sort (module C: comparable) li =
 List.sort (C.compare) li
```

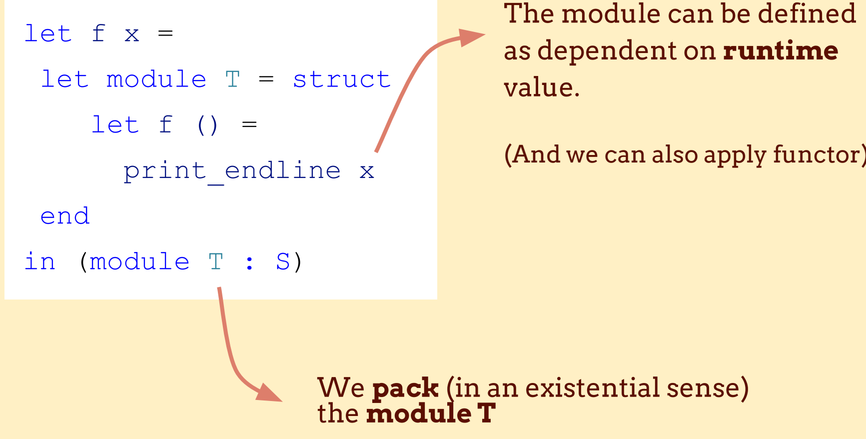
The module is **known statically**

- Reduce the need for funcors (that can be heavy
- Allows a form of **Higher kinded polymorphism**

```
let map (module M: MONAD)
        (f: 'a -> 'b) (x: 'a M.t) : 'b M.t =
 M.bind x (fun x -> M.return (f x))
```

First-class module

build **dynamic module based on runtime value**



Interacting with the value-level

Even though we presented the value language and the module language as two distinct languages, OCaml has **two analogous mechanisms for interacting with the value language.**

since 5.5.0 Module-dependent function

functions **parameterized over a module**

```
let sort (module C: comparable) li =
 List.sort (C.compare) li
```

The module is
known statically

- Reduce the need for funcors (that can be heavy
- Allows a form of **Higher kinded polymorphism**

```
let map (module M: MONAD)
      (f: 'a -> 'b) (x: 'a M.t) : 'b M.t =
 M.bind x (fun x -> M.return (f x))
```

First-class module

build **dynamic module based on runtime value**

Val **unpack** a *packed module*

```
module Fs =
(val (let kind = Sys.getenv_opt "CONTEXT" in
    match kind with
    | Some "in-test" -> (module Test : file_system)
    | _ -> (module Regular : file_system)))
```

The module is
known at runtime

Let’s look at some examples of how to use module language. For fun and profit!

Let's have some **fun** with a Prévert-style list of module use cases!

Shared behavior

Shared behavior

```
module type WITH_COMPARE = sig
 type t
 val compare : t -> t -> int
end
```

Shared behavior

```
module type WITH_COMPARE = sig
 type t
 val compare : t -> t -> int
end
```

```
module Comparable (M : WITH_COMPARE) = struct
 let ( > ) x y = M.compare x y > 0
 let ( >= ) x y = M.compare x y >= 0
 let ( < ) x y = M.compare x y < 0
 let ( <= ) x y = M.compare x y <= 0
 let min a b = if a > b then b else a
 let max a b = if a > b then a else b

 let clamp ~min:a ~max:b x =
   let small = min a b and big = max a b in
   min big (max small x)
end
```

Shared behavior

```
module type WITH_COMPARE = sig
 type t
 val compare : t -> t -> int
end
```

```
module Comparable (M : WITH_COMPARE) = struct
 let ( > ) x y = M.compare x y > 0
 let ( >= ) x y = M.compare x y >= 0
 let ( < ) x y = M.compare x y < 0
 let ( <= ) x y = M.compare x y <= 0
 let min a b = if a > b then b else a
 let max a b = if a > b then a else b

 let clamp ~min:a ~max:b x =
   let small = min a b and big = max a b in
   min big (max small x)
end
```

```
module Int = struct
 include Stdlib.Int
 include Comparable (Int)
end

module Int64 = struct
 include Stdlib.Int64
 include Comparable (Int64)
end

module Float= struct
 include Stdlib.Float
 include Comparable (Float)
end
```

Less **Error Prone**

Data-structures

We have already seen the example of the sack, but the standard library **has several data structures that are already functorized**,

for example:

Data-structures

We have already seen the example of the sack, but the standard library **has several data structures that are already functorized**,

for example:

```
(* A Set of integers. *)
module Int_set = Set.Make (Int)

(* A Map indexed over strings. *)
module String_map = Map.Make (String)

(* An Hash Table indexed over floats. *)
module Float_hash = Hashtbl.Make (Float)
```

Data-structures

We have already seen the example of the sack, but the standard library **has several data structures that are already functorized**,

for example:

```
(* A Set of integers. *)
module Int_set = Set.Make (Int)

(* A Map indexed over strings. *)
module String_map = Map.Make (String)

(* An Hash Table indexed over floats. *)
module Float_hash = Hashtbl.Make (Float)
```

Fits well with extensions and abstractions (*even with* higher-kinded polymorphism).

Dependency Injection

Dependency Injection

```
module type CONSOLE = sig
 val print : string -> unit
 val read : unit -> string
end
```

Dependency Injection

```
module type CONSOLE = sig
 val print : string -> unit
 val read : unit -> string
end
```

```
module Program (C : CONSOLE) = struct
 let run () =
   let () = C.print "Hello World\n" in
   let () = C.print "What is your name?:\n" in
   let name = C.read () in
   C.print ("Hello " ^ name)
end
```

Dependency Injection

```
module type CONSOLE = sig
 val print : string -> unit
 val read : unit -> string
end
```

```
module Program (C : CONSOLE) = struct
 let run () =
   let () = C.print "Hello World\n" in
   let () = C.print "What is your name?:\n" in
   let name = C.read () in
   C.print ("Hello " ^ name)
end
```

```
let program (module C : CONSOLE) =
 let () = C.print "Hello World\n" in
 let () = C.print "What is your name?:\n" in
 let name = C.read () in
 C.print ("Hello " ^ name)
```

Dependency Injection

```
module type CONSOLE = sig
 val print : string -> unit
 val read : unit -> string
end
```

```
module Program (C : CONSOLE) = struct
 let run () =
   let () = C.print "Hello World\n" in
   let () = C.print "What is your name?:\n" in
   let name = C.read () in
   C.print ("Hello " ^ name)
end
```

```
let program (module C : CONSOLE) =
 let () = C.print "Hello World\n" in
 let () = C.print "What is your name?:\n" in
 let name = C.read () in
 C.print ("Hello " ^ name)
```

can be simplified using a **Reader Monad** (for example)

Dependency Injection

```
module type CONSOLE = sig
 val print : string -> unit
 val read : unit -> string
end
```

```
module Program (C : CONSOLE) = struct
 let run () =
   let () = C.print "Hello World\n" in
   let () = C.print "What is your name?:\n" in
   let name = C.read () in
   C.print ("Hello " ^ name)
end
```

```
let program (module C : CONSOLE) =
 let () = C.print "Hello World\n" in
 let () = C.print "What is your name?:\n" in
 let name = C.read () in
 C.print ("Hello " ^ name)
```

can be simplified using a **Reader Monad** (for example)

But in OCaml, in fact, **we have other tools for that**:

- User Defined Effects
- Objects

Packing module and runtime decision

Packing module and runtime decision

```
module type file_system = sig
 val write_file : path -> string -> unit
 val read_file : path -> string option
end
```

Packing module and runtime decision

```
module Regular = struct
 let write_file = File.write
 let read_file = File.read
end

module Test = struct
 let fs = ref ["a.txt", "A"; "b.txt", "B"]
 let read_file path =
   List.assoc_opt path !fs

 let write_file path content =
   fs := List.map (fun (p, ctn) ->
       if p = path then p, content
       else p, ctn
     ) !fs
end
```

```
module type file_system = sig
 val write_file : path -> string -> unit
 val read_file : path -> string option
end
```

Packing module and runtime decision

```
module Regular = struct
 let write_file = File.write
 let read_file = File.read
end

module Test = struct
 let fs = ref ["a.txt", "A"; "b.txt", "B"]
 let read_file path =
   List.assoc_opt path !fs

 let write_file path content =
   fs := List.map (fun (p, ctn) ->
       if p = path then p, content
       else p, ctn
     ) !fs
end
```

```
module type file_system = sig
 val write_file : path -> string -> unit
 val read_file : path -> string option
end
```

```
module Fs=
(val (let kind = Sys.getenv_opt "CONTEXT" in
    match kind with
    | Some "in-test" -> (module Test : file_system)
    | _ -> (module Regular : file_system)))
```

Capability Tokens Generated by Functors

A very simple example of singleton-type using generative functors

Capability Tokens Generated by Functors

A very simple example of singleton-type using generative functors

```
module type CAP = sig
 type t
 val create : unit -> t
end

module Make_cap () : CAP =
struct
 type t = Uniq (* We do not care *)
 let create () = Uniq
end

module Ressource (C: CAP) = struct
 let access (_cap: C.t) =
   print_endline "succeed"
end
```

let's make it **generative**

Capability Tokens Generated by Functors

A very simple example of singleton-type using generative functors

```
module type CAP = sig
 type t
 val create : unit -> t
end

module Make_cap () : CAP =
struct
 type t = Uniq (* We do not care *)
 let create () = Uniq
end

module Ressource (C: CAP) = struct
 let access (_cap: C.t) =
   print_endline "succeed"
end
```

let's make it **generative**

```
module A = Make_cap ()
module B = Make_cap ()

let () =
 let module RA = Ressource (A) in
 RA.access (A.create ())

let () =
 let module RA = Ressource (A) in
 RA.access (B.create ())
```

Capability Tokens Generated by Functors

A very simple example of singleton-type using generative functors

```
module type CAP = sig
 type t
 val create : unit -> t
end

module Make_cap () : CAP =
struct
 type t = Uniq (* We do not care *)
 let create () = Uniq
end

module Ressource (C: CAP) = struct
 let access (_cap: C.t) =
   print_endline "succeed"
end
```

let's make it **generative**

```
module A = Make_cap ()
module B = Make_cap ()

let () =
 let module RA = Ressource (A) in
 RA.access (A.create ())

let () =
 let module RA = Ressource (A) in
 RA.access (B.create ())
```

Typechecking **will fail**

Algebraic structure

Algebraic structure

```
module type SEMIGROUP = sig
 type t
 val concat : t -> t -> t
end

module type MONOID = sig
 include SEMIGROUP
 val neutral : t
end
```

Algebraic structure

```
module type SEMIGROUP = sig
 type t
 val concat : t -> t -> t
end

module type MONOID = sig
 include SEMIGROUP
 val neutral : t
end
```

```
module Make_semigroup (S: SEMIGROUP) = struct
 include S
 let fold x xs = List.fold_left concat x xs
end

module Make_moinoid (M: MONOID) = struct
 include Make_semigroup (M)
 include M
 let reduce xs =
    List.fold_left concat neutral xs
end
```

Algebraic structure

```
module type SEMIGROUP = sig
 type t
 val concat : t -> t -> t
end

module type MONOID = sig
 include SEMIGROUP
 val neutral : t
end
```

```
let fold (module M: SEMIGROUP) (x: M.t) (xs: M.t list) =
 List.fold_left M.concat x xs

let reduce (module M: MONOID) (xs: M.t list) =
 List.fold_left M.concat M.neutral xs
```

```
module Make_semigroup (S: SEMIGROUP) = struct
 include S
 let fold x xs = List.fold_left concat x xs
end

module Make_moinoid (M: MONOID) = struct
 include Make_semigroup (M)
 include M
 let reduce xs =
    List.fold_left concat neutral xs
end
```

Algebraic structure

this a pretty new syntax 🥳

before that, it was heavy, with locally abstract types

```
module type SEMIGROUP = sig
 type t
 val concat : t -> t -> t
end

module type MONOID = sig
 include SEMIGROUP
 val neutral : t
end
```

```
let fold (module M: SEMIGROUP) (x: M.t) (xs: M.t list) =
 List.fold_left M.concat x xs

let reduce (module M: MONOID) (xs: M.t list) =
 List.fold_left M.concat M.neutral xs
```

```
module Make_semigroup (S: SEMIGROUP) = struct
 include S
 let fold x xs = List.fold_left concat x xs
end

module Make_moinoid (M: MONOID) = struct
 include Make_semigroup (M)
 include M
 let reduce xs =
    List.fold_left concat neutral xs
end
```

Algebraic structure

this a pretty new syntax 🥳

before that, it was heavy, with locally abstract types

```
module type SEMIGROUP = sig
 type t
 val concat : t -> t -> t
end

module type MONOID = sig
 include SEMIGROUP
 val neutral : t
end
```

```
let fold (module M: SEMIGROUP) (x: M.t) (xs: M.t list) =
 List.fold_left M.concat x xs

let reduce (module M: MONOID) (xs: M.t list) =
 List.fold_left M.concat M.neutral xs
```

```
module Make_semigroup (S: SEMIGROUP) = struct
 include S
 let fold x xs = List.fold_left concat x xs
end

module Make_moinoid (M: MONOID) = struct
 include Make_semigroup (M)
 include M
 let reduce xs =
    List.fold_left concat neutral xs
end
```

Since **module-dependent functions** are brand new, we still need to build a library **design culture**.

Tagless Final Encoding

Tagless Final Encoding

```
module type INT = sig
 type 'a t

 val int : int -> int t
 val add : int t -> int t -> int t
end
```

```
module type BOOL = sig
 type 'a t

 val bool : bool -> bool t
 val if_ : bool t -> 'a t -> 'a t -> 'a t
end
```

Tagless Final Encoding

```ocaml
module type INT = sig
 type 'a t

 val int : int -> int t
 val add : int t -> int t -> int t
end
```

```ocaml
module type BOOL = sig
 type 'a t

 val bool : bool -> bool t
 val if_ : bool t -> 'a t -> 'a t -> 'a t
end
```

```ocaml
module type LANG = sig
 include INT
 include BOOL with type 'a t := 'a t
end
```

Tagless Final Encoding

Tagless Final Encoding

```
module Eval : LANG = struct
 type 'a t = 'a
 let int x = x
 let bool b = b
 let add x y = x + y
 let if_ pred if_true if_false =
     if pred then if_true else if_false
end
```

Tagless Final Encoding

```
module Eval : LANG = struct
 type 'a t = 'a
 let int x = x
 let bool b = b
 let add x y = x + y
 let if_ pred if_true if_false =
     if pred then if_true else if_false
end
```

```
module Print : LANG = struct
 type 'a t = string
 let int x = string_of_int x
 let bool x = string_of_bool x
 let add x y = "(" ^ x ^ " + " ^ y ^ ")"
 let if_ b x y =
   "(if " ^ b ^ " then " ^ x ^ " else " ^ y ^ ")"
end
```

Tagless Final Encoding

```
module Eval : LANG = struct
 type 'a t = 'a
 let int x = x
 let bool b = b
 let add x y = x + y
 let if_ pred if_true if_false =
     if pred then if_true else if_false
end
```

```
module Print : LANG = struct
 type 'a t = string
 let int x = string_of_int x
 let bool x = string_of_bool x
 let add x y = "(" ^ x ^ " + " ^ y ^ ")"
 let if_ b x y =
   "(if " ^ b ^ " then " ^ x ^ " else " ^ y ^ ")"
end
```

```
module P (L : LANG) = struct
 let program =
   let open L in
   if_ (bool true)
       (add (int 1) (int 2))
     (add (int 4) (int 5))
end
```

First-class modules exaggeration

Let's try to find the type of an arbitrary expression (without type parameters)!

Simplified from **https://def.lakaban.net/2021-06-25-inferring-type-declarations-in-ocaml/**

First-class modules exaggeration

Let's try to find the type of an arbitrary expression (without type parameters)!
Simplified from **https://def.lakaban.net/2021-06-25-inferring-type-declarations-in-ocaml/**

```
let my_expression = 42


type t = (* to be filled *)
```

First-class modules exaggeration

Let's try to find the type of an arbitrary expression (without type parameters)!
Simplified from **https://def.lakaban.net/2021-06-25-inferring-type-declarations-in-ocaml/**

```
let my_expression = 42


type t = (* to be filled *)
```

```
module type T = sig type t end
```

First-class modules exaggeration

Let's try to find the type of an arbitrary expression (without type parameters)!
Simplified from **https://def.lakaban.net/2021-06-25-inferring-type-declarations-in-ocaml/**

```
let my_expression = 42


type t = (* to be filled *)
```

```
module type T = sig type t end
```

```
let ty (type a) (x : a) : (module T with type t = a) =
  (module struct type t = a end)
```

First-class modules exaggeration

Let's try to find the type of an arbitrary expression (without type parameters)!
Simplified from **https://def.lakaban.net/2021-06-25-inferring-type-declarations-in-ocaml/**

```
let my_expression = 42

type t = (* to be filled *)
```

```
module type T = sig type t end
```

This is a locally abstract type

```
let ty (type a) (x : a) : (module T with type t = a) =
  (module struct type t = a end)
```

First-class modules exaggeration

Let's try to find the type of an arbitrary expression (without type parameters)!
Simplified from **https://def.lakaban.net/2021-06-25-inferring-type-declarations-in-ocaml/**

```
let my_expression = 42
module Ty = (val ty my_expression)
type t = Ty.t
```

```
module type T = sig type t end
```

This is a locally abstract type

```
let ty (type a) (x : a) : (module T with type t = a) =
  (module struct type t = a end)
```

First-class modules exaggeration

Let's try to find the type of an arbitrary expression (without type parameters)!
Simplified from **https://def.lakaban.net/2021-06-25-inferring-type-declarations-in-ocaml/**

```
let my_expression = 42
module Ty = (val ty my_expression)
type t = Ty.t
```

We **Unpack our module**

```
module type T = sig type t end
```

This is a locally abstract type

```
let ty (type a) (x : a) : (module T with type t = a) =
  (module struct type t = a end)
```

First-class modules exaggeration

Let's try to find the type of an arbitrary expression (without type parameters)!
Simplified from **https://def.lakaban.net/2021-06-25-inferring-type-declarations-in-ocaml/**

```
let my_expression = 42
module Ty = (val ty my_expression)
type t = Ty.t
```

And we have **Ty.t** = **int**

```
module type T = sig type t end
```

This is a locally abstract type

```
let ty (type a) (x : a) : (module T with type t = a) =
  (module struct type t = a end)
```

First-class modules exaggeration

Let's try to find the type of an arbitrary expression (without type parameters)!
Simplified from **https://def.lakaban.net/2021-06-25-inferring-type-declarations-in-ocaml/**

```
let my_expression = 42
module Ty = (val ty my_e
type t = Ty.t
```

```
module type T = sig type t end
```

abstract type

```
= a) =
```

And we hav

```
include (
 (functor (M : sig module type T module X : T end) -> M.X)
  (struct
   let some_expr () = my_expression
   module type T0 = sig type my_type end
   module X = (val (
       (fun (type a) (_ : unit -> a) :
          (module T0 with type my_type = a) ->
          (module struct type my_type = a end))
       some_expr
     ))
   module type T = module type of X
  end))
```

Here is the author's version that brings **my_type** in the scope

Work In progress

Still an area of research, with recent
proposals currently being integrated.

Work In progress

Still an area of research, with recent proposals currently being integrated.

Avoiding Signature Avoidance in ML Modules with Zippers

Clément Blaudeau, Didier Rémy, Gabriel Radanne

▶ **To cite this version:**

Clément Blaudeau, Didier Rémy, Gabriel Radanne. Avoiding Signature Avoidance in ML Modules with Zippers. Proceedings of the ACM on Programming Languages, 2025, POPL (9), 10.1145/3704902. hal-04801582

Work In progress

Still an area of research, with recent proposals currently being integrated.

Signature Avoidance: losing some shared abstract types because of type inference

Avoiding Signature Avoidance in ML Modules with Zippers

Clément Blaudeau, Didier Rémy, Gabriel Radanne

▸ **To cite this version:**

Clément Blaudeau, Didier Rémy, Gabriel Radanne. Avoiding Signature Avoidance in ML Modules with Zippers. Proceedings of the ACM on Programming Languages, 2025, POPL (9), 10.1145/3704902 . hal-04801582

Work In progress

Still an area of research, with recent proposals currently being integrated.

Signature Avoidance: losing some shared abstract types because of type inference

Avoiding Signature Avoidance in ML Modules with Zippers

Clément Blaudeau, Didier Rémy, Gabriel Radanne

▸ **To cite this version:**

Clément Blaudeau, Didier Rémy, Gabriel Radanne. Avoiding Signature Avoidance in ML Modules with Zippers. Proceedings of the ACM on Programming Languages, 2025, POPL (9), 10.1145/3704902 . hal-04801582

Fulfilling OCaml Modules with Transparency

BLAUDEAU CLEMENT, Inria, France and Université Paris Cité, France
DIDIER RÉMY, Inria, France
GABRIEL RADANNE, Inria, France

ML modules come as an additional layer on top of a core language to offer large-scale notions of composition and abstraction. They largely contributed to the success of OCaml and SML. While modules are easy to write for common cases, their advanced use may become tricky. Additionally, despite a long line of works, their meta-theory remains difficult to comprehend, with involved soundness proofs. In fact, the module layer of OCaml does not currently have a formal specification and its implementation has some surprising behaviors.

Building on previous translations from ML modules to F^ω, we propose a type system, called M^ω, that covers a large subset of OCaml modules, including both applicative and generative functors, and extended with transparent ascription. This system produces signatures in an OCaml-like syntax extended with F^ω quantifiers. We provide a reverse translation from M^ω signatures to path-based source signatures along with a characterization of *signature avoidance* cases, making M^ω signatures well suited to serve as a new internal representation for a typechecker. The soundness of the type system is shown by elaboration in F^ω. We improve

Work In progress

Still an area of research, with recent proposals currently being integrated.

Opaque Ascription (current): Hides implementation details and keeps types abstract unless the signature specifies otherwise.

Transparent Ascription: Preserves type equalities from the implementation.

Signature Avoidance: losing some shared abstract types because of type inference

Avoiding Signature Avoidance in ML Modules with Zippers

Clément Blaudeau, Didier Rémy, Gabriel Radanne

▶ To cite this version:

Clément Blaudeau, Didier Rémy, Gabriel Radanne. Avoiding Signature Avoidance in ML Modules with Zippers. Proceedings of the ACM on Programming Languages, 2025, POPL (9), 10.1145/3704902 . hal-04801582

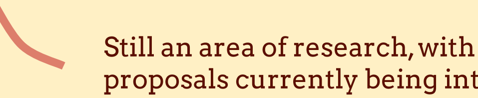
Fulfilling OCaml Modules with Transparency

BLAUDEAU CLEMENT, Inria, France and Université Paris Cité, France
DIDIER RÉMY, Inria, France
GABRIEL RADANNE, Inria, France

ML modules come as an additional layer on top of a core language to offer large-scale notions of composition and abstraction. They largely contributed to the success of OCaml and SML. While modules are easy to write for common cases, their advanced use may become tricky. Additionally, despite a long line of works, their meta-theory remains difficult to comprehend, with involved soundness proofs. In fact, the module layer of OCaml does not currently have a formal specification and its implementation has some surprising behaviors.

Building on previous translations from ML modules to F^ω, we propose a type system, called M^ω, that covers a large subset of OCaml modules, including both applicative and generative functors, and extended with transparent ascription. This system produces signatures in an OCaml-like syntax extended with F^ω quantifiers. We provide a reverse translation from M^ω signatures to path-based source signatures along with a characterization of *signature avoidance* cases, making M^ω signatures well suited to serve as a new internal representation for a typechecker. The soundness of the type system is shown by elaboration in F^ω. We improve

Work In progress



Still an area of research, with recent proposals currently being integrated.

If we have that… we can have

Fulfilling OCaml Modules with Transparency

BLAUDEAU CLEMENT, Inria, France and Université Paris Cité, France
DIDIER RÉMY, Inria, France
GABRIEL RADANNE, Inria, France

ML modules come as an additional layer on top of a core language to offer large-scale notions of composition and abstraction. They largely contributed to the success of OCaml and SML. While modules are easy to write for common cases, their advanced use may become tricky. Additionally, despite a long line of works, their meta-theory remains difficult to comprehend, with involved soundness proofs. In fact, the module layer of OCaml does not currently have a formal specification and its implementation has some surprising behaviors.

Building on previous translations from ML modules to F^ω, we propose a type system, called M^ω, that covers a large subset of OCaml modules, including both applicative and generative functors, and extended with transparent ascription. This system produces signatures in an OCaml-like syntax extended with F^ω quantifiers. We provide a reverse translation from M^ω signatures to path-based source signatures along with a characterization of *signature avoidance* cases, making M^ω signatures well suited to serve as a new internal representation for a typechecker. The soundness of the type system is shown by elaboration in F^ω. We improve over previous encodings of sealing *within applicative functors*, by the introduction of *transparent existential types*, a weaker form of existential types that can be lifted out of universal and arrow types. This shines a new light on the form of abstraction provided by applicative functors and brings their treatment much closer to those of generative functors.

CCS Concepts: • **Software and its engineering** → *Functional languages*; *Semantics*; **Modules / packages**; •

Work In progress

Still an area of research, with recent proposals currently being integrated.

If we have that… we can have

Fulfilling OCaml Modules with Transparency

BLAUDEAU CLEMENT, Inria, France and Université Paris Cité, France
DIDIER RÉMY, Inria, France
GABRIEL RADANNE, Inria, France

ML modules come as an additional layer on top of a core language to offer large-scale notions of composition and abstraction. They largely contributed to the success of OCaml and SML. While modules are easy to write for common cases, their advanced use may become tricky. Additionally, despite a long line of works, their meta-theory remains difficult to comprehend, with involved soundness proofs. In fact, the module layer of OCaml does not currently have a formal specification and its implementation has some surprising behaviors.

Building on previous translations from ML modules to F^ω, we propose a type system, called M^ω, that covers a large subset of OCaml modules, including both applicative and generative functors, and extended with transparent ascription. This system produces signatures in an OCaml-like syntax extended with F^ω quantifiers. We provide a reverse translation from M^ω signatures to path-based source signatures along with a characterization of *signature avoidance* cases, making M^ω signatures well suited to serve as a new internal representation for a typechecker. The soundness of the type system is shown by elaboration in F^ω. We improve over previous encodings of sealing *within applicative functors*, by the introduction of *transparent existential types*, a weaker form of existential types that can be lifted out of universal and arrow types. This shines a new light on the form of abstraction provided by applicative functors and brings their treatment much closer to those of generative functors.

CCS Concepts: • **Software and its engineering** → *Functional languages*; *Semantics*; **Modules / packages**; •

Adhoc Polymorphism

On the design and implementation of Modular Explicits

Samuel Vivien[*] Didier Rémy[†] Thomas Refis[‡] Gabriel Scherer[§]

Presented at the OCaml workshop 2024, September 7

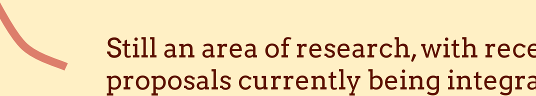
Abstract

We present and discuss the design and implementation of *modular explicits*, an extension of OCaml first-class modules with *module-dependent functions*, functions taking *first-class modules* as arguments. We show some difficulties with the present use of first-class modules and how modular explicits solve them in a simpler, more direct way. Modular explicits are fully compatible with, and can be presented as an extension of, first-class modules. Interestingly, both the formalization and the implementation reuse the mechanism designed to ensure principal types in the presence of semi-explicit first-class polymorphism and OCaml polymorphic methods. Modular explicits are also meant to be the underlying language in which *modular implicits*, i.e., module arguments left implicit from their signatures, should be elaborated.

Introduction

The name *modular explicits* is coined from *modular implicits* an extension proposed by White, Bour, and Yallop (2014) to provide OCaml with a mechanism similar to Haskell type classes but based on, and compatible with, the module system of OCaml. This builds on two ideas: implementing type classes as implicit arguments in Scala proposed by Oliveira, Moors, and Odersky (2010), but emulating type classes at the module-level rather then at the core-level as proposed by Dreyer, Harper, Chakravarty, and Keller (2007). Using implicit module arguments

Work In progress



Still an area of research, with recent proposals currently being integrated.

Transparent ascription is necessary for modular implicits because **implicit resolution relies on visible type equalities** to match modules with the required types.

If we have that… we can have

Adhoc Polymorphism

Fulfilling OCaml Modules with Transparency

BLAUDEAU CLEMENT, Inria, France and Université Paris Cité, France
DIDIER RÉMY, Inria, France
GABRIEL RADANNE, Inria, France

ML modules come as an additional layer on top of a core language to offer large-scale notions of composition and abstraction. They largely contributed to the success of OCaml and SML. While modules are easy to write for common cases, their advanced use may become tricky. Additionally, despite a long line of works, their meta-theory remains difficult to comprehend, with involved soundness proofs. In fact, the module layer of OCaml does not currently have a formal specification and its implementation has some surprising behaviors.

Building on previous translations from ML modules to F^ω, we propose a type system, called M^ω, that covers a large subset of OCaml modules, including both applicative and generative functors, and extended with transparent ascription. This system produces signatures in an OCaml-like syntax extended with F^ω quantifiers. We provide a reverse translation from M^ω signatures to path-based source signatures along with a characterization of *signature avoidance* cases, making M^ω signatures well suited to serve as a new internal representation for a typechecker. The soundness of the type system is shown by elaboration in F^ω. We improve over previous encodings of sealing *within applicative functors*, by the introduction of *transparent existential types*, a weaker form of existential types that can be lifted out of universal and arrow types. This shines a new light on the form of abstraction provided by applicative functors and brings their treatment much closer to those of generative functors.

CCS Concepts: • **Software and its engineering** → *Functional languages*; *Semantics*; **Modules / packages**; •

On the design and implementation of Modular Explicits

Samuel Vivien* Didier Rémy† Thomas Refis‡ Gabriel Scherer§

Presented at the OCaml workshop 2024, September 7

Abstract

We present and discuss the design and implementation of *modular explicits*, an extension of OCaml first-class modules with *module-dependent functions*, functions taking *first-class modules* as arguments. We show some difficulties with the present use of first-class modules and how modular explicits solve them in a simpler, more direct way. Modular explicits are fully compatible with, and can be presented as an extension of, first-class modules. Interestingly, both the formalization and the implementation reuse the mechanism designed to ensure principal types in the presence of semi-explicit first-class polymorphism and OCaml polymorphic methods. Modular explicits are also meant to be the underlying language in which *modular implicits*, i.e., module arguments left implicit from their signatures, should be elaborated.

Introduction

The name *modular explicits* is coined from *modular implicits* an extension proposed by White, Bour, and Yallop (2014) to provide OCaml with a mechanism similar to Haskell type classes but based on, and compatible with, the module system of OCaml. This builds on two ideas: implementing type classes as implicit arguments in Scala proposed by Oliveira, Moors, and Odersky (2010), but emulating type classes at the module-level rather then at the core-level as proposed by Dreyer, Harper, Chakravarty, and Keller (2007). Using implicit module arguments

To conclude

We have seen that modules form a **small language** governed by its **own type system**. This enforces a **clear separation between implementation and interface**, allowing for **namespacing** and **reusability**, while providing powerful **abstraction** mechanisms.

But **Functors can be heavy**
(event if module-dependent function is a clear improvement, reducing verbosity)

To conclude

We have seen that modules form a **small language** governed by its **own type system**. This enforces a **clear separation between implementation and interface**, allowing for **namespacing** and **reusability**, while providing powerful **abstraction** mechanisms.

But **Functors can be heavy**
(event if module-dependent function is a clear improvement, reducing verbosity)

To conclude

I hope we have covered **a large part** of the **OCaml module language** in an *accessible way* *(and that the presentation was not too simplistic)*.

To conclude

We have seen that modules form a **small language** governed by its **own type system**. This enforces a **clear separation between implementation and interface**, allowing for **namespacing** and **reusability**, while providing powerful **abstraction** mechanisms.

But **Functors can be heavy**
(event if module-dependent function is a clear improvement, reducing verbosity)

I hope we have covered **a large part** of the **OCaml module language** in an *accessible way* *(and that the presentation was not too simplistic)*.

- OCaml is **cool**
- Module are **expressive and powerful**
- The field is constantly evolving.
- With **Modular Implicit**, there's no need to envy Type Classes (*FMO*) anymore.

You should definitely use OCaml for your next project, whether personal or professional!

Thanks!