# Avoiding Signature Avoidance (in OCaml)

Didier Rémy

Joint work with
Clément Blaudeau

Inria

IFIP WG2.8. Utretch, April 2024

## A key part of the OCaml system

- Key feature for type abstraction and programming in the large
- Easy to use for simple, standard patterns
- Their dynamic semantics is really obvious

## A key part of the OCaml system

- Key feature for type abstraction and programming in the large
- Easy to use for simple, standard patterns
- Their dynamic semantics is really obvious

## However,

- Their static semantics is somewhat involved, and non standard,
- Difficulties witnessed by a long line of works,
- with some significant advances, spread over 3 decades ! ***F-ing ML modules***

## A key part of the OCaml system

- Key feature for type abstraction and programming in the large
- Easy to use for simple, standard patterns
- Their dynamic semantics is really obvious

## However,

- Their static semantics is somewhat involved, and non standard,
- Difficulties witnessed by a long line of works,
- with some significant advances, spread over 3 decades ! ***F-ing ML modules***

## Still, OCaml

- lacks a complete specification,
- suffers from signature avoidance problem, *solved in unpredictable ways*
- *Poor interaction with the core language*

# Our goal (today)

## Improve signature avoidance in OCaml

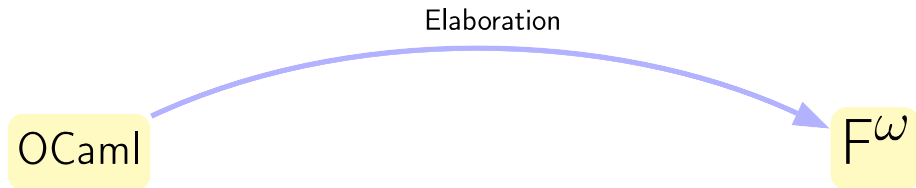
- A more principled, complete approach
- wtr a restricted, incomplete, but simple specification

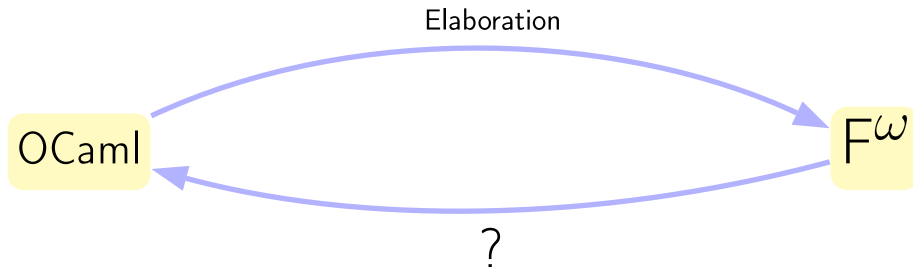
# Our goal (today)

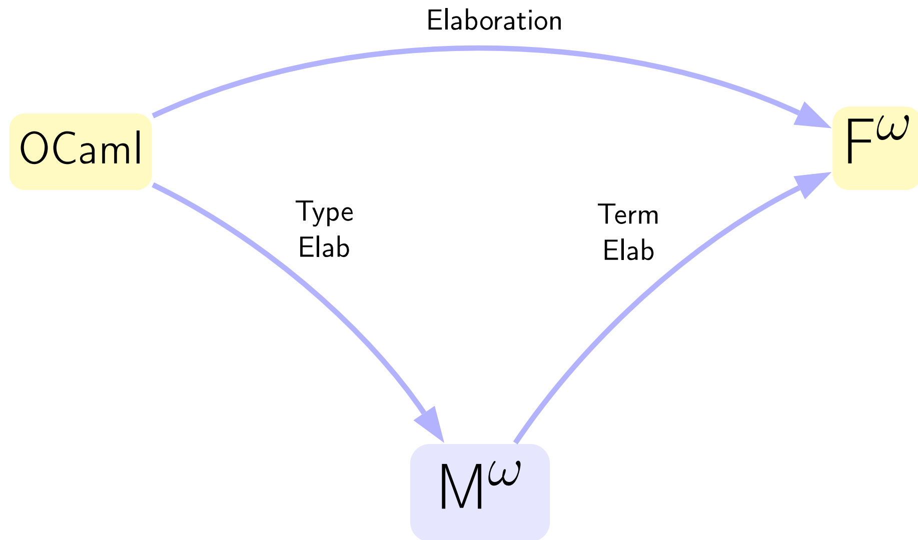
## Improve signature avoidance in OCaml

- A more principled, complete approach
- wtr a restricted, incomplete, but simple specification

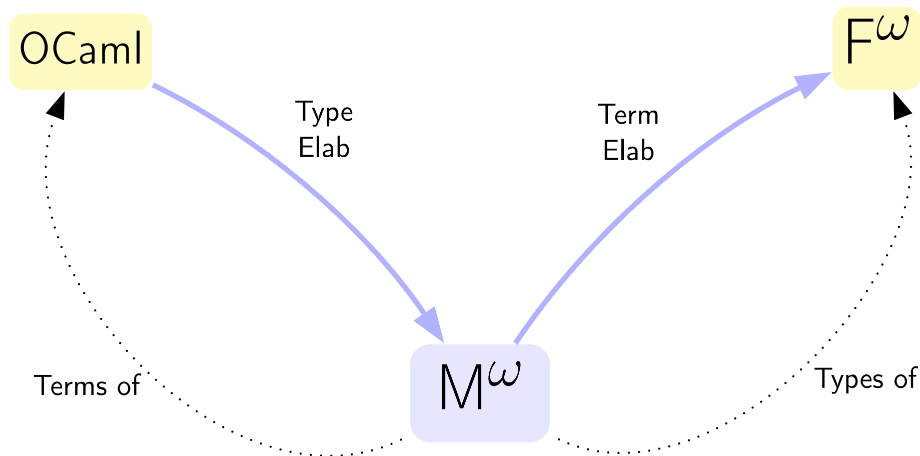
## Based on a detour in $F^\omega$ (previous work, OOPSLA'24)


Elaboration
OCaml
$F^\omega$

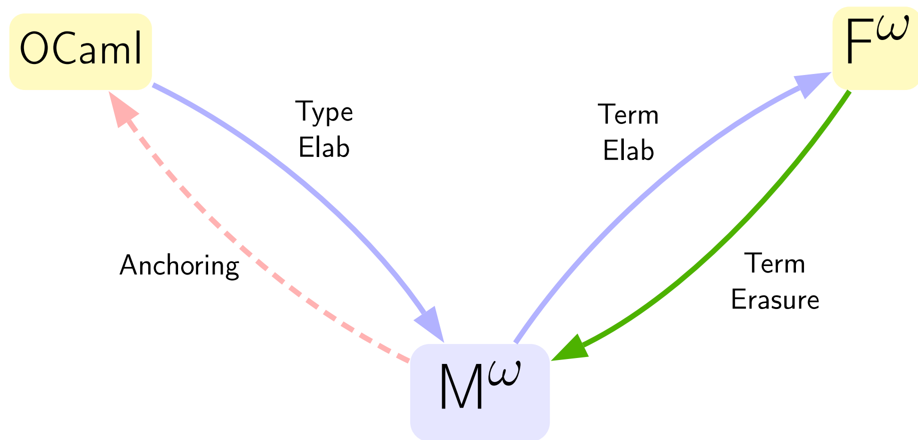

Elaboration
OCaml
$F^\omega$
?


Elaboration
OCaml
$F^{\omega}$
Type
Elab
Term
Elab
$M^{\omega}$

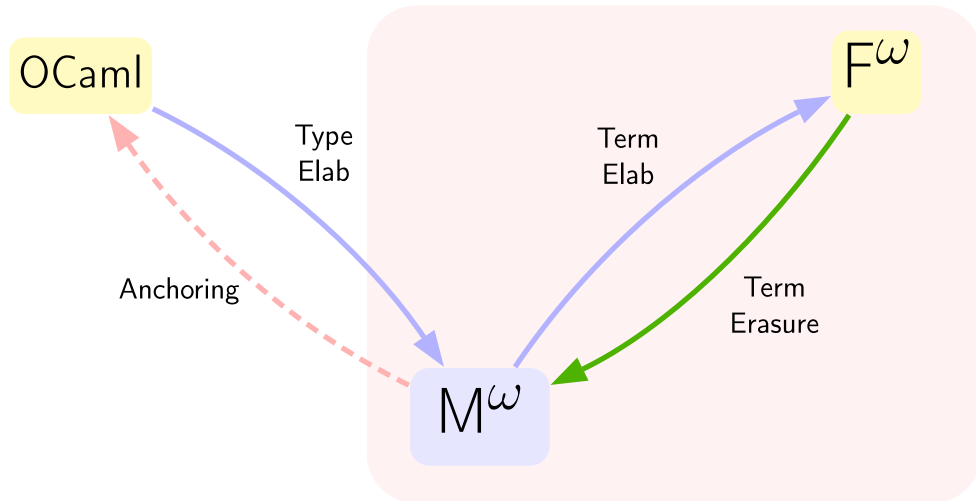
# The big picture



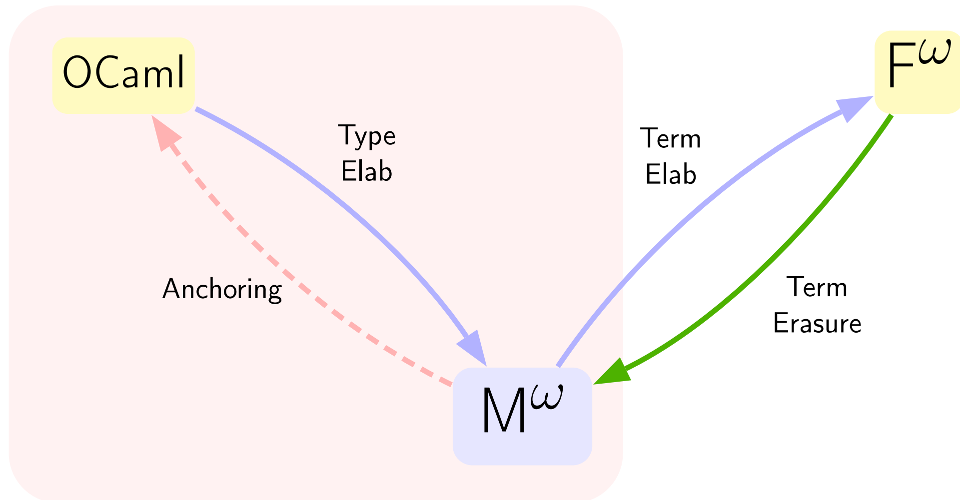
# The big picture



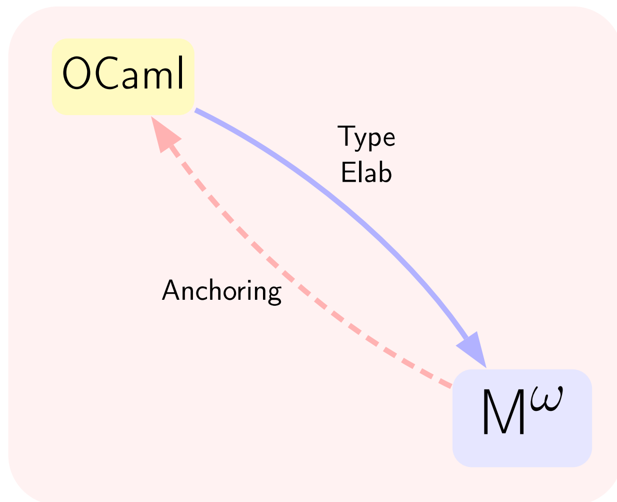
- Extending $F^\omega$ with primitive modules
- Ensuring soundness of $M^\omega$



- Improving signature avoidance
- Adding transparent ascription

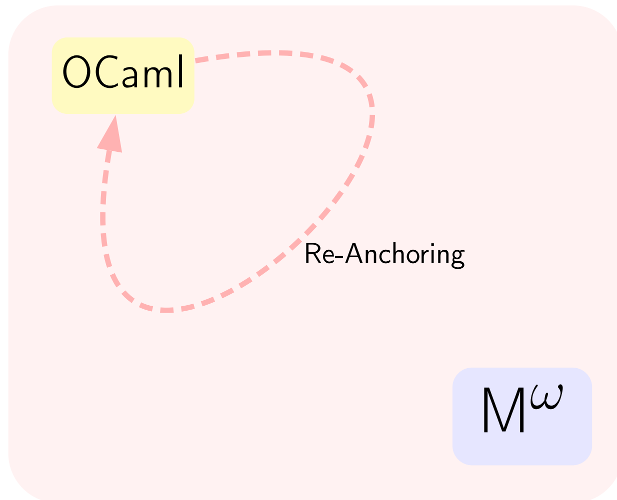


- Improving signature avoidance
- Adding transparent ascription



- Improving signature avoidance
- Adding transparent ascription

} Pulled back to OCaml



# Signature synthesis *(nothing to avoid)*

```
module M = struct
  module X = struct type t end
  type u = X.t type v = X.t type w = u
end
```

In signatures:

✓
```
M : sig
  module X : sig type t end
  type u = X.t type v = X.t type w = u
end
```

```
module M = struct
  module X = struct type t end
  type u = X.t type v = X.t type w = u
end
```

In signatures:

✓
```
M : sig
  module X : sig type t end
  type u = X.t type v = X.t type w = u
end
```

Here,

- there is only one abstract type $X.t$
- all free types $u$, $v$, and $w$ are concrete, equal to $X.t$

Type inference may loose sharing of abstract types

```
module M = struct
  open struct module X = struct type t end end
  type u = X.t type v = X.t type w = u
end
```

When some abstract type get out of scope

```
M : sig
  module X : sig type t end
  type u = X.t type v = X.t type w = u
end
```

# Signature avoidance *is misbehaving*

Type inference may loose sharing of abstract types

```
module M = struct
  let module X = struct type t end in
  type u = X.t type v = X.t type w = u
end
```

When some abstract type get out of scope

```
M : sig
  module X : sig type t end
  type u = X.t type v = X.t type w = u
end
```

# Signature avoidance *is misbehaving*

Type inference may loose sharing of abstract types

```
module M = struct
  let module X = struct type t end in
  type u = X.t type v = X.t type w = u
end
```

When some abstract type get out of scope

✓

```
M : sig
  type u     type v     type w = u
end
```

# Signature avoidance *is misbehaving*

Type inference may loose sharing of abstract types

```
module M = struct
  let module X = struct type t end in
  type u = X.t type v = X.t type w = u
end
```

When some abstract type get out of scope

✓

```
M : sig
  type u          type v          type w = u
end
```

However,

- equality between $u$ and $v$ is lost !

# Signature avoidance *is misbehaving*

Type inference may loose sharing of abstract types

```
module M = struct
  let module X = struct type t end in
  type u = X.t type v = X.t type w = u
end
```

When some abstract type get out of scope

✓
```
M : sig
  type u      type v      type w = v
end
```

However,

- equality between *u* and *v* is lost !
- another, incomparable choice exists ...

# Signature avoidance *is misbehaving*

Type inference may loose sharing of abstract types

```
module M = struct
  let module X = struct type t end in
  type u = X.t type v = X.t type w = u
end
```

When some abstract type get out of scope

✓

```
M : sig
  type u     type v = u     type w = u
end
```

However,

- equality between $u$ and $v$ is lost !
- another, incomparable choice exists . . .
- while a correct, principal signature exists !

Type inference may loose sharing of abstract types

```
module M = struct
  let module X = struct type t end in
  type u = X.t type v = X.t type w = u
end
```

When some abstract type get out of scope

✓
```
M : sig
  module X : sig type t end
  type u = X.t type v = u type w = u
end
```

However,

- equality between *u* and *v* is lost !
- another, incomparable choice exists . . .
- while a correct, principal signature exists !

Similar situations for projections and applications.

# Signature avoidance is *unavoidable*

```
module M = struct
  let module X = struct type t end in
  let f x : X.t = x
end
```

✗

```
M : sig
  module X : sig type t end
  val f : X.t → X.t
end
```

# Signature avoidance is *sometimes solvable*

```
module M = struct
  let module X = struct type t end in
  type u = X.t   type v = u
end
```

```
M : sig
  module X : sig type t end
  type u = X.t   type v = u
end
```

# Signature avoidance is *not so obvious*

```
module M = struct
  let module X = struct type t end in
  type u = X.t list
end
```

?

```
M : sig
  module X : sig type t end
  type u = X.t list
end
```

# Signature avoidance is *not so obvious*

```
module M = struct
  let module X = struct type t end in
  type u = X.t list
end
```

✓

```
M : sig
  module X : sig type t end
  type u = X.t list
end
```

OCaml solves it by over abstraction:

# Signature avoidance is *not so obvious*

```
module M = struct
  let module X = struct type t end in
  type u = X.t list    let x : u = []
end
```

✓
```
M : sig
  module X : sig type t end
  type u = X.t list    val x : u
end
```

OCaml solves it by over abstraction:

✗
```
let n = List.length M.x
```

# Signature avoidance is *not so obvious*

```
module M = struct
  let module X = struct type t end in
  type u = X.t list    let x : u = []
end
```

✓
```
M : sig
  module X : sig type t end
  type u = X.t list   val x : u
end
```

OCaml solves it by over abstraction:

✓
```
let n = List.length M.x
```

# Signature avoidance is *not so obvious*

```
module M = struct
  let module X = struct type t end in
  type u = X.t list    let x : u = []
end
```

```
M : sig
  module X : sig type t end
  type u = X.t list    val x : u
end
```

✓

OCaml solves it by over abstraction:

```
let n = List.length M.x
```

✗

**Over-abstraction should not be allowed**

Even when it is the best solution (up to subtyping).

Since this is usually a delayed typechecking error.

# Signature avoidance is *really not obvious*

```
module E = struct type e end          module type E = sig type e end
```

```
module M = struct
  let module F(X : E)(Y : E) = struct type t end in
  module Diag(Z : E) = F(Z)(Z)
  module U = F(E)(E)   module V = Diag(E)
  let eq (x : U.t) : V.t = x
end
```

✓
```
module M : sig
    module F : functor (X : E) (Y : E) → sig type t end
    module Diag : functor (Z : E) → sig type t = F(E)(E).t end
    module U : sig type t = F(E)(E).t end
    module V : sig type t = F(E)(E).t end
    val eq : U.t → V.t
  end
```

✓
```
let eq (x : M.U.t) : M.V.t = x
```

# Signature avoidance is *really not obvious*

```
module E = struct type e end                module type E = sig type e end
```

```
module M = struct
  let module F(X : E)(Y : E) = struct type t end in
  module Diag(Z : E) = F(Z)(Z)
  module U = F(E)(E)   module V = Diag(E)
  let eq (x : U.t) : V.t = x
end
```

✓
```
module M : sig
    module F : functor (X : E) (Y : E) → sig type t end
    module Diag : functor (Z : E) → sig type t = F(E)(E).t end
    module U : sig type t = F(E)(E).t end
    module V : sig type t = F(E)(E).t end
    val eq : U.t → V.t
  end
```

✗
```
let eq (x : M.U.t) : M.V.t = x
```

## Signature avoidance is *really not obvious*

```
module E = struct type e end            module type E = sig type e end

module M = struct
  let module F(X : E)(Y : E) = struct type t end in
  module Diag(Z : E) = F(Z)(Z)
  let module U = F(E)(E) in      module V = Diag(E)
  let eq (x : U.t) : V.t = x
end

module M : sig
    module F : functor (X : E) (Y : E) → sig type t end
    module Diag : functor (Z : E) → sig type t = F(E)(E).t end
✗   module U : sig type t = F(E)(E).t end
    module V : sig type t = F(E)(E).t end
    val eq : U.t → V.t
  end
```

OCaml fails... to replace *U.t* by *V.t*

# Signature avoidance is *really not obvious*

```
module E = struct type e end          module type E = sig type e end
```

```
module M = struct
  let module F(X : E)(Y : E) = struct type t end in
  module Diag(X : E) = F(X)(X)
  (* inlining U = F(E)(E) *)
  type t = F(E)(E).t
end
```

✓

```
module M : sig
    module F : functor (X : E) (X : E) → sig type t end
    module Diag : functor (X : E) → sig type t = F(X)(X).t end


    type t = F(E)(E).t
  end
```

OCaml succeeds with over abstraction !

# Signature avoidance is *really not obvious*

```
module E = struct type e end          module type E = sig type e end
```

```
module M = struct
  let module F(X : E)(Y : E) = struct type t end in
  module Diag(X : E) = F(X)(X)
  (* inlining U = F(E)(E) *)
  type t = F(E)(E).t
end
```

✓

```
module M : sig
    module F : functor (X : E) (X : E) → sig type t end
    module Diag : functor (X : E) → sig type t = F(X)(X).t end


    type t = F(E)(E).t = Diag(E).t
  end
```

OCaml succeeds with over abstraction !

# Signature avoidance is *really not obvious*

```
module E = struct type e end            module type E = sig type e end

module M = struct
  let module F(X : E)(Y : E) = struct type t end in
  module Diag(X : E) = F(X)(X)
  (* inlining U = F(E)(E) *)
  type t = F(E)(E).t
end

module M : sig
    module F : functor (X : E) (X : E) → sig type t end
    module Diag : functor (X : E) → sig type t = F(X)(X).t end
✓
✗
    type t = F(E)(E).t = Diag(E).t
  end
```

OCaml succeeds with over abstraction !

Should we really invent the application $Diag(E)$ to avoid $F(E)(E).t$ ?

# Signature avoidance is *really not obvious*

```
module E = struct type e end          module type E = sig type e end
```

```
module M = struct
  let module F(X : E)(Y : E) = struct type t end in
  module Diag(X : E) = F(X)(X)
  (* inlining U = F(E)(E) *)
  type t = F(E)(E).t
end
```

```
module M : sig
    module F : functor (X : E) (X : E) → sig type t end
    module Diag : functor (X : E) → sig type t = F(X)(X).t end
✓
✗
    type t = F(E)(E).t = Diag(E).t
  end
```

OCaml succeeds with over abstraction !
Should we really invent the application $Diag(E)$ to avoid $F(E)(E).t$ ?

# Over abstraction is not so obvious

```
module F (X : E) (Y : E) = struct
  let module G(Z : E) = struct type t end in
  type u = G(X).t

end
```

✓

```
module F (X : E) (Y : E) : sig
  module G : functor (Z : E) → sig type t end
  type u = G(X).t

end
```

# Over abstraction is not so obvious

```
module F (X : E) (Y : E) = struct
  let module G(Z : E) = struct type t end in
  type u = G(X).t

end
```

```
module F (X : E) (Y : E) : sig
  module G : functor (Z : E) → sig type t end
  type u = G(X).t

end
```

# Over abstraction is not so obvious

```
module F (X : E) (Y : E) = struct
  let module G(Z : E) = struct type t end in
  type u = G(X).t
  type v = G(Y).t
end
```

?

```
module F (X : E) (Y : E) : sig
  module G : functor (Z : E) → sig type t end
  type u = G(X).t
  type v = G(Y).t
end
```

# Over abstraction is not so obvious

```
module F (X : E) (Y : E) = struct
  let module G(Z : E) = struct type t end in
  type u = G(X).t
  type v = G(Y).t
end
```

```
module F (X : E) (Y : E) : sig
  module G : functor (Z : E) → sig type t end
  type u = G(X).t
  type v = G(Y).t
end
```

# Over abstraction is not so obvious

```
module F (X : E) (Y : E) = struct
  let module G(Z : E) = struct type t end in
  type u = G(X).t
  type v = G(Y).t
end
```

✓

```
module F (X : E) (Y : E) : sig
  module G : functor (Z : E) → sig type t end
  type u = G(X).t
  type v = G(Y).t
end
```

But,

✗

```
module M = F(E)(E)
let eq (x : M.u) : M.v = x
```

# Over abstraction is not so obvious

```
module F (X : E) (Y : E) = struct
  let module G(Z : E) = struct type t end in
  type u = G(X).t
  type v = G(Y).t
end
```

✓

```
module F (X : E) (Y : E) : sig
  module G : functor (Z : E) → sig type t end
  type u = G(X).t
  type v = G(Y).t
end
```

While,

✓

```
module M = F(E)(E)
let eq (x : M.u) : M.v = x
```

Without $G$, we cannot express that $u = v$ only when $X = Y$ !

# We may now focus on signatures

```
module F (X : E) (Y : E) : sig
  module G : functor (Z : E) → sig type t end
  type u = G(X).t
  type v = G(Y).t
end
```

# We may now focus on signatures using self-references

$$
\begin{array}{l}
\textbf{\textit{module}}\ F\ (X : E)\ (Y : E) : \textbf{\textit{sig}}(A) \\
\quad \textbf{\textit{module}}\ G : \textbf{\textit{functor}}\ (Z : E) \rightarrow \textbf{\textit{sig}}(B)\ \textbf{\textit{type}}\ t = B.t\ \textbf{\textit{end}} \\
\quad \textbf{\textit{type}}\ u = A.G(X).t \\
\quad \textbf{\textit{type}}\ v = A.G(Y).t \\
\textbf{\textit{end}}
\end{array}
$$

# The signature avoidance problem

## Type components of a signature sig ( A ) $D$ end

| | | |
|---|---|---|
| – abstract type definition: | type $t$ = A.t | *(equal to itself)* |
| – type alias | type $t = Q.u$ | where $Q.u \neq A.t$ |
| – concrete type definition | type $t = \tau$ | where $\tau \neq Q.u$ |

Type components induce an equivalence on types. (related to strengthening)

$$\frac{\Gamma \vdash P : \texttt{sig}\,(A)\; D\;\texttt{end} \qquad \text{type } t = \tau \in D}{\Gamma \vdash P.t \approx \tau[A \leftarrow P]}$$

## The challenge

- when some components are to be hidden in a signature $S$,
- can we write a (well-formed) subsignature while preserving type equalities ?

Signature avoidance can be defined, (almost) wlog, as:

$$\begin{array}{llll} \text{given } S & \triangleq \texttt{sig}\,(A)\ D_F\ ;\ D\ \texttt{end} & \text{where } \Gamma \vdash S & \\ \text{find } S' & \triangleq \texttt{sig}\,(A)\ D'\ ;\ D'_F\ \texttt{end} & \text{where } \Gamma \vdash S \approx S' \text{ and } \lfloor D' \rfloor = \lfloor D \rfloor \end{array}$$

where

$$\Gamma \vdash S \approx S' \iff \Gamma \vdash S \leq S' \wedge \Gamma \vdash S' \leq S$$

$$\lfloor D' \rfloor = \lfloor D \rfloor \iff \text{fields of } D \text{ and } D' \text{ appear in the same order, recursively}$$

$\Gamma \vdash S \approx S'$ implies $\Gamma \vdash S'$, which implies $\Gamma \vdash \texttt{sig}\,(A)\ D'\ \texttt{end}$,

Hence $D'_F$ can be removed

# The problem *equivalently high-level view*

Signature avoidance can be defined, (almost) wlog, as:

given $S \triangleq \texttt{sig}\,(A)\ \texttt{module}\,F : S_F\ ;\texttt{module}\,D : S_D\ \texttt{end}$ where $\Gamma \vdash S$
find $S' \triangleq \texttt{sig}\,(A)\ \texttt{module}\,D : S'_D\ ;\texttt{module}\,F : S'_F\ \texttt{end}$ where $\Gamma \vdash S \approx S'$ and $\lfloor S'_D \rfloor = \lfloor S_D \rfloor$

where

$$\Gamma \vdash S \approx S' \iff \Gamma \vdash S \leq S' \wedge \Gamma \vdash S' \leq S$$

$$\lfloor S'_D \rfloor = \lfloor S_D \rfloor \iff \text{fields of } D \text{ and } D' \text{ appear in the same order, recursively}$$

$\Gamma \vdash S \approx S'$ implies $\Gamma \vdash S'$, which implies $\Gamma \vdash S'_D$,

Hence $S'_F$ can be projected on $D$

# A trivial (no) solution

No rewrite, always fail

# Avoiding signature avoidance

## Enrich signatures

with floating components

- keep components to be hidden (in yellow) as floating components
- statically visible, but dynamically inaccessible — and inaccessible to the user !
- allow them to be deleted when possible or necessary (for subtyping).

# Avoiding signature avoidance

## Enrich signatures

with floating components

- keep components to be hidden (in yellow) as floating components
- statically visible, but dynamically inaccessible — and inaccessible to the user !
- allow them to be deleted when possible or necessary (for subtyping).

## Key remark

- floating components looks like garbage, but their names will never clash !
- since applications use subtyping (and not equality)
  - subtyping is always against an explicit source signature on the right-hand side,
  - which does not contain floating fields.
- Then subtyping will remove floating fields.
- This can be seen as delaying signature avoidance resolution
  until source signature annotations are provided by the user

(Somewhat generalizes de-Bruin-indexed inaccessible types of SML)

# A range of solutions in between

Always fail, when avoidance occurs

Never fail, keep floating components (as expressive as $M^\omega$)

# A range of solutions in between

## Always fail, when avoidance occurs

- a value field can always be removed
- a floating field can be moved foward when this preserves well-formedness
- a floating field can be removed when no other field depends on it

## Never fail, keep floating components (as expressive as $\mathsf{M}^\omega$)

# A range of solutions in between

## Always fail, when avoidance occurs

- a value field can always be removed
- a floating field can be moved foward when this preserves well-formedness
- a floating field can be removed when no other field depends on it

## Try to remove all floating components

## Never fail, keep floating components (as expressive as $M^\omega$)

# A range of solutions in between

## Always fail, when avoidance occurs

- a value field can always be removed
- a floating field can be moved foward when this preserves well-formedness
- a floating field can be removed when no other field depends on it

## Try to remove all floating components

- using type alias equivalence as much as possible

## Never fail, keep floating components (as expressive as $M^{\omega}$)

# A range of solutions in between

## Always fail, when avoidance occurs

- a value field can always be removed
- a floating field can be moved foward when this preserves well-formedness
- a floating field can be removed when no other field depends on it

## Try to remove all floating components

- using type alias equivalence as much as possible
- Design choice:
  - restrict equivalence to keep avoidance tractable, intuitive, and predictable
- fail, or stay with remaining floating components

## Never fail, keep floating components (as expressive as $M^\omega$)

# Without applicative functors

## We may (pre) compute type equivalence

$$P \ ::= \ X \mid P.Z \ \mid \ \cancel{P(P)}$$

# Without applicative functors

## We may (pre) compute type equivalence

$$P ::= X \mid P.Z \mid \cancel{P(P)}$$

$$\frac{\Gamma \vdash P : \texttt{sig}\,(A)\ D\ \texttt{end} \qquad \text{type } t = \tau \in D}{\Gamma \vdash P.t \approx \tau[A \leftarrow P]}$$

# Without applicative functors

## We may (pre) compute type equivalence

$$P ::= X \mid P.Z \mid \cancel{P(P)}$$

$$\frac{\Gamma \vdash P : \texttt{sig}\,(A)\; D\;\texttt{end} \qquad \text{type } t = \tau \in D}{\Gamma \vdash P.t \leadsto \tau[A \leftarrow P]}$$

Equivalence classes can be represented by a tree of redirections.

The root of a path is its type.

# Without applicative functors

## We may (pre) compute type equivalence

$$P ::= X \mid P.Z \mid \not{P(P)}$$

$$\frac{\Gamma \vdash P : \texttt{sig}\,(A)\; D\; \textsf{end} \qquad \textsf{type}\; t = \tau \in D}{\Gamma \vdash P.t \rightsquigarrow \tau[A \leftarrow P]}$$

Equivalence classes can be represented by a tree of redirections.

The root of a path is its type.

## Resolving signature avoidance is complete *(wrt the specification)*

# Example — *building the redirection tree*

$$
\begin{array}{l}
P : \textbf{sig}\,(A) \\
\quad \textbf{module}\ X : \textbf{sig}\ \ (B)\ \textbf{type}\ t\ = B.t\ \ \textbf{end} \\
\quad \textbf{type}\ z\ = A.X.t\ \ list \\
\quad \textbf{type}\ u\ = A.X.t \\
\quad \textbf{type}\ v\ = A.X.t \\
\quad \textbf{type}\ w = A.v \\
\textbf{end}
\end{array}
$$

$P$ : ***sig*** $(A)$
- ***module*** $X$ : ***sig*** $(B)$ ***type*** $t = B.t$ ***end***

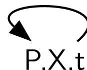
  ***type*** $z = A.X.t$ *list*

  ***type*** $u = A.X.t$

  ***type*** $v = A.X.t$

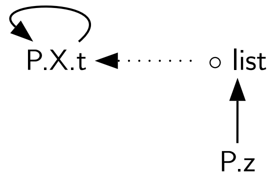
  ***type*** $w = A.v$

***end***



# Example *building the redirection tree*

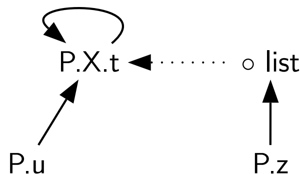
$P$ : ***sig*** $(A)$
  ***module*** $X$ : ***sig*** $(B)$ ***type*** $t = B.t$ ***end***
▶ ***type*** $z = A.X.t$ *list*
  ***type*** $u = A.X.t$
  ***type*** $v = A.X.t$
  ***type*** $w = A.v$
***end***



# Example

## *building the redirection tree*

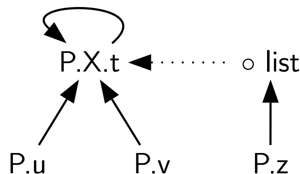
```
P : sig (A)
  module X : sig (B) type t = B.t end
  type z = A.X.t list
▶ type u = A.X.t
  type v = A.X.t
  type w = A.v
end
```



# Example

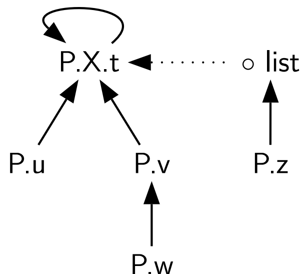
## *building the redirection tree*

$P$ : ***sig*** $(A)$
  ***module*** $X$ : ***sig*** $(B)$ ***type*** $t = B.t$ ***end***
  ***type*** $z = A.X.t$ *list*
  ***type*** $u = A.X.t$
▶ ***type*** $v = A.X.t$
  ***type*** $w = A.v$
***end***



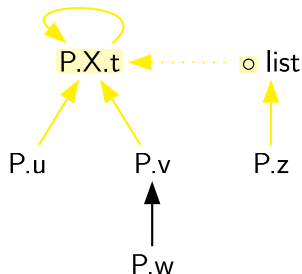
# Example *building the redirection tree*

$P$ : ***sig*** $(A)$
  ***module*** $X$ : ***sig*** $(B)$ ***type*** $t = B.t$ ***end***
  ***type*** $z = A.X.t$ *list*
  ***type*** $u = A.X.t$
  ***type*** $v = A.X.t$
▶ ***type*** $w = A.v$
***end***



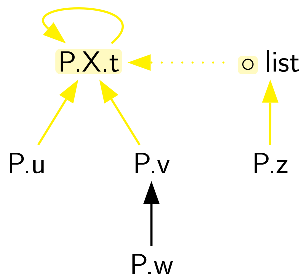
# Example *avoiding A.X*

$$
\begin{array}{l}
P : \textbf{sig}\,(A) \\
\quad \textbf{module}\ X : \textbf{sig}\ (B)\ \textbf{type}\ t = B.t\ \textbf{end} \\
\quad \textbf{type}\ z = A.X.t\ \mathit{list} \\
\quad \textbf{type}\ u = A.X.t \\
\quad \textbf{type}\ v = A.X.t \\
\quad \textbf{type}\ w = A.v \\
\textbf{end}
\end{array}
$$



# Example *avoiding A.X*

```
P : sig (A)
  module X : sig (B) type t = B.t end
▶ type z = A.X.t list
  type u = A.X.t
  type v = A.X.t
  type w = A.v
end
```

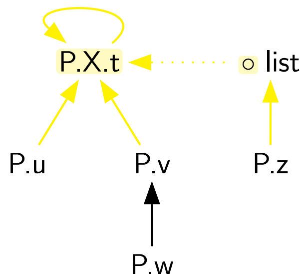



✗ Current anchor for type P.X.t is floating/out of scope

# Example



$P$ : ***sig*** $(A)$
  ***module*** $X$ : ***sig*** $(B)$ ***type*** $t = B.t$ ***end***
  ***type*** $u = A.X.t$
  ***type*** $z = A.X.t$ *list*
  ***type*** $v = A.X.t$
  ***type*** $w = A.v$
***end***

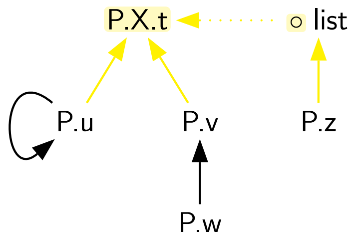



- Assume we bound $u$ before $z$...

# Example

*avoiding A.X*

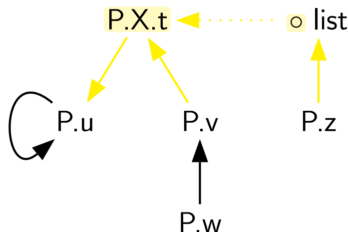
$P$ : ***sig*** $(A)$
    ***module*** $X$ : ***sig*** $(B)$ ***type*** $t = A.u$ ***end***
    ***type*** $u = A.X.t = A.u$
    ***type*** $z = A.X.t$ *list*
    ***type*** $v = A.X.t$
    ***type*** $w = A.v$
***end***



# Example

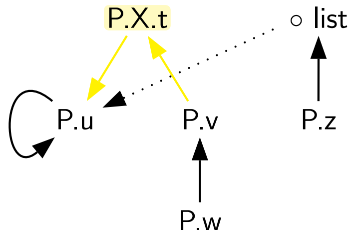
*avoiding A.X*

$P$ : ***sig*** $(A)$
▶ ***module*** $X$ : ***sig*** $(B)$ ***type*** $t = A.u$ ***end***
***type*** $u = A.X.t = A.u$
***type*** $z = A.X.t$ *list*
***type*** $v = A.X.t$
***type*** $w = A.v$
***end***



# Example *avoiding A.X*

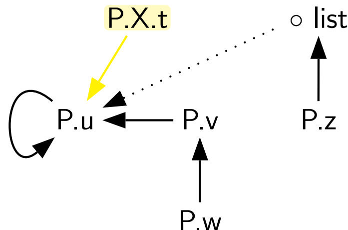
$$
\begin{array}{l}
P : \textbf{sig}\,(A) \\
\quad \textbf{module}\ X : \textbf{sig}\ (B)\ \textbf{type}\ t = A.u\ \textbf{end} \\
\quad \textbf{type}\ u = A.X.t = A.u \\
\quad \textbf{type}\ z = A.X.t\ \mathit{list} = A.u\ \mathit{list} \\
\quad \textbf{type}\ v = A.X.t \\
\quad \textbf{type}\ w = A.v \\
\textbf{end}
\end{array}
$$



# Example

*avoiding A.X*

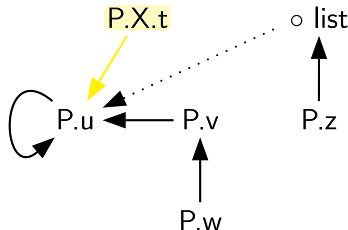
$$
\begin{array}{l}
P : \textbf{sig}\,(A) \\
\quad \textbf{module}\ X : \textbf{sig}\ (B)\ \textbf{type}\ t = A.u\ \textbf{end} \\
\quad \textbf{type}\ u = A.X.t = A.u \\
\quad \textbf{type}\ z = A.X.t\ \mathit{list} = A.u\ \mathit{list} \\
\blacktriangleright \quad \textbf{type}\ v = A.X.t = A.u \\
\quad \textbf{type}\ w = A.v \\
\textbf{end}
\end{array}
$$



# Example



$P$ : ***sig*** $(A)$
  ***module*** $X$ : ***sig*** $(B)$ ***type*** $t = A.u$ ***end***
  ***type*** $u = A.X.t = A.u$
  ***type*** $z = A.X.t$ *list* $= A.u$ *list*
  ***type*** $v = A.X.t = A.u$
▶ ***type*** $w = A.v$
***end***




✓ Success. All sharing has been preserved.

# Applicative functors

$P ::= X \mid P.Z \mid P(P)$

If $P$ : **functor** $(X : S) \to \texttt{sig}\,(A)$ type $t$ **end**

- two applications of $P$ to the very same module $V$
- produce the same abstract type $P(V).t$.

We collect redirections as before

A functor does not itself produce redirections

An application

$$P(V) : \texttt{sig}\,(A)\ \text{type}\ t = A.t; \text{type}\ u = V.t\ \textbf{end}$$

will produce

- an abstract type definition $P(V).t \rightsquigarrow P(V).t$ (self-reference)
- a type alias $P(V).u \rightsquigarrow V.t$

# Module aliases

OCaml extends signatures with module aliases $(= Q)$:

Example:

```
module X = struct type t end
module Y = X
```

```
module X : sig type t end
module Y : (= X)
```

Type aliases induce an equivalence on paths

$$\frac{\Gamma \vdash P : (= Q)}{\Gamma \vdash P \approx Q}$$

# Module aliases

OCaml extends signatures with module aliases $(= Q)$:

Example:

| | |
|---|---|
| **module** X = **struct type** t **end**<br>**module** Y = X | ***module*** *X* : ***sig type*** *t* ***end***<br>***module*** *Y* : *(= X)* |

Type aliases induce an equivalence on paths

$$\frac{\Gamma \vdash P : (= Q)}{\Gamma \vdash P \approx Q} \qquad \frac{\Gamma \vdash P \approx P' \qquad \Gamma \vdash Q \approx Q'}{\Gamma \vdash P(Q) \approx P'(Q')} \qquad \frac{\Gamma \vdash P \approx Q}{\Gamma \vdash P.X \approx Q.X}$$

OCaml extends signatures with module aliases $(= Q)$:

Example:

```
module X = struct type t end
module Y = X
```

```
module X : sig type t end
module Y : (= X)
```

Type aliases induce an equivalence on paths

$$\frac{\Gamma \vdash P : (= Q)}{\Gamma \vdash P \approx Q} \qquad \frac{\Gamma \vdash P \approx P' \qquad \Gamma \vdash Q \approx Q'}{\Gamma \vdash P(Q) \approx P'(Q')} \qquad \frac{\Gamma \vdash P \approx Q}{\Gamma \vdash P.X \approx Q.X}$$

This enriches the equivalence on types:

$$\frac{\Gamma \vdash P \approx Q}{\Gamma \vdash P.t \approx Q.t}$$

# Module aliases

OCaml extends signatures with module aliases $(= Q)$:

Example:

```
module X = struct type t end
module Y = X
```

```
module X : sig type t end
module Y : (= X)
```

Type aliases induce an equivalence on paths

$$\frac{\Gamma \vdash P : (= Q)}{\Gamma \vdash P \approx Q} \qquad \frac{\Gamma \vdash P \approx P' \qquad \Gamma \vdash Q \approx Q'}{\Gamma \vdash P(Q) \approx P'(Q')} \qquad \frac{\Gamma \vdash P \approx Q}{\Gamma \vdash P.X \approx Q.X}$$

This enriches the equivalence on types:

$$\frac{\Gamma \vdash P \approx Q}{\Gamma \vdash P.t \approx Q.t} \qquad \frac{\vdash Diag(E) : (= F(E)(E))}{Diag(E) \approx F(E)(E)}$$

This is too large for signature avoidance: it may invent paths with new applications !

OCaml extends signatures with module aliases $(= Q)$:

Example:

```
module X = struct type t end
module Y = X
```

```
module X : sig type t end
module Y : (= X)
```

Type aliases induce an equivalence on paths

$$\frac{\Gamma \vdash P : (= Q) \quad \text{noapp}(P)}{\Gamma \vdash P \approx Q} \qquad \frac{\Gamma \vdash P \approx P' \qquad \Gamma \vdash Q \approx Q'}{\Gamma \vdash P(Q) \approx P'(Q')} \qquad \frac{\Gamma \vdash P \approx Q}{\Gamma \vdash P.X \approx Q.X}$$

This enriches the equivalence on types:

$$\frac{\Gamma \vdash P \approx Q}{\Gamma \vdash P.t \approx Q.t} \qquad \frac{\vdash Diag(E) : (= F(E)(E))}{Diag(E) \approx F(E)(E)} \qquad \frac{\vdash Diag \not\approx F}{\vdash Diag(E) \not\approx F(E)(E)}$$

(The middle rule is crossed out.)

This is too large for signature avoidance: it may invent paths with new applications !

# Module aliases — restricted path equivalence

OCaml extends signatures with module aliases $(= Q)$:

Example:

```
module X = struct type t end
module Y = X
```

```
module X : sig type t end
module Y : (= X)
```

Type aliases induce an equivalence on paths

$$\frac{\Gamma \vdash P : (= Q) \quad \text{noapp}(P)}{\Gamma \vdash P \rightsquigarrow Q} \qquad \frac{\Gamma \vdash P \rightsquigarrow P' \qquad \Gamma \vdash Q \rightsquigarrow Q'}{\Gamma \vdash P(Q) \rightsquigarrow P'(Q')} \qquad \frac{\Gamma \vdash P \rightsquigarrow Q}{\Gamma \vdash P.X \rightsquigarrow Q.X}$$

This enriches the equivalence on types:

$$\frac{\Gamma \vdash P \rightsquigarrow Q}{\Gamma \vdash P.t \rightsquigarrow Q.t} \qquad \frac{\vdash Diag(E) : (= F(E)(E))}{Diag(E) \rightsquigarrow F(E)(E)} \qquad \frac{\vdash Diag \not\rightsquigarrow F}{\vdash Diag(E) \not\rightsquigarrow F(E)(E)}$$

This is too large for signature avoidance: it may invent paths with new applications !

We use an independent, similar tree-redirection map for path equivalence.

# Transparent ascription

Transparent ascription $(= P <: S)$ generalizes module aliases to keep both

- the identity of $P$
- an interface $S$ smaller than the one of $P$

Signature avoidance works as before,

- ignoring the signature restriction first, and
- retyping the signature afterward

*Transparent ascription allows*

- *to keep module aliases across applications*
- *hence preserve more type equalities*

# Take away

## Signature avoidance can be avoided

- using existential types, as in $M^{\omega}$, *F-ing*, or Moscow-ML
- but also using path-based OCaml-like signatures with floating components

## Signature avoidance can be improved in OCaml

- with a more principled, complete approach,
- using a restricted, incomplete, but simple specification (for applications)

## Further improvements, for yet more avoidance ?

? allow more application equivalences see

? reduce false over-abstraction failures see

# Take away

## Signature avoidance can be avoided

- using existential types, as in $M^{\omega}$, *F-ing*, or Moscow-ML
- but also using path-based OCaml-like signatures with floating components

## Signature avoidance can be improved in OCaml

- with a more principled, complete approach,
- using a restricted, incomplete, but simple specification (for applications)

## Further improvements, for yet more avoidance ?

✗ allow more application equivalences see

✓ reduce false over-abstraction failures see

The end

## More interaction with the core language

*a little of 1ML*

- for programming in the small with *modular explicits*, e.g., treating first-class functors as first-class functions.

## Modular implicits

- Letting implicit module arguments be inferred from modules types.
- Completeness of typechecking becomes essential for coherence
- Calls for transparent ascription
  - to keep module identities across subtyping — and applications
  - increasing module equalities $\Longrightarrow$ reduces coherence issues

## *More program proofs*

- *Mismatch with "standard" type theory (e.g., System $F^\omega$)*
- *Makes it more difficult to transport invariants/proofs by typing.*