OCaml Modules and signatures: formalization, insights and improvements

Clément Blaudeau
EPFL master thesis under the supervision of Didier Rémy and Gabriel Radanne

October 15, 2021

Table of contents

Table of contents

Table of contents

Table of contents

Table of contents

The power of the OCaml Module system

Modules and signatures: OCaml modularity in a nutshell

```
(* A simple module
   to encapsulate integers *)
module Integers = struct
  type t = int
  let eq (x:t) (y:t) = (x=y)
end
```

Modules and signatures: OCaml modularity in a nutshell

```
(* A simple module
   to encapsulate integers *)
module Integers = struct
  type t = int
  let eq (x:t) (y:t) = (x=y)
end

(* A generic interface for
   a module with an equality
   function and a base type *)
module type Comp = sig
  type t
  val eq : t -> t -> bool
end
```

Modules and signatures: OCaml modularity in a nutshell

```
(* A simple module
   to encapsulate integers *)
module Integers = struct
  type t = int
  let eq (x:t) (y:t) = (x=y)
end

(* A generic interface for
   a module with an equality
   function and a base type *)
module type Comp = sig
  type t
  val eq : t -> t -> bool
end
```

Modules and signatures: OCaml modularity in a nutshell

```
(* A simple module
   to encapsulate integers *)
module Integers = struct
  type t = int
  let eq (x:t) (y:t) = (x=y)
end

(* A generic interface for
   a module with an equality
   function and a base type *)
module type Comp = sig
  type t
  val eq : t -> t -> bool
end
```

```
(* A set functor with signature
   ascription to abstract the set type *)
module Set = functor (A: Comp) ->
struct
  type t = A.t
  type set = t list
  let empty_set : set = []
  let rec add x s : set = ...
end
```

Modules and signatures: OCaml modularity in a nutshell

```
(* A simple module
   to encapsulate integers *)
module Integers = struct
  type t = int
  let eq (x:t) (y:t) = (x=y)
end

(* A generic interface for
   a module with an equality
   function and a base type *)
module type Comp = sig
  type t
  val eq : t -> t -> bool
end
```

```
(* A set functor with signature
   ascription to abstract the set type *)
module Set = functor (A: Comp) -> (
struct
  type t = A.t
  type set = t list
  let empty_set : set = []
  let rec add x s : set = ...
end : sig
  type t
  type set (* abstract ! *)
  val empty_set : set
  val add : t -> set -> set
end)
```

Modules and signatures: OCaml modularity in a nutshell

```
(* A simple module
   to encapsulate integers *)
module Integers = struct
  type t = int
  let eq (x:t) (y:t) = (x=y)
end

(* A generic interface for
   a module with an equality
   function and a base type *)
module type Comp = sig
  type t
  val eq : t -> t -> bool
end
```

```
(* A set functor with signature
   ascription to abstract the set type *)
module Set = functor (A: Comp) -> (
struct
  type t = A.t
  type set = t list
  let empty_set : set = []
  let rec add x s : set = ...
end : sig
  type t
  type set (* abstract ! *)
  val empty_set : set
  val add : t -> set -> set
end)
```

Modules and signatures: OCaml modularity in a nutshell

```
(* A simple module
   to encapsulate integers *)
module Integers = struct
  type t = int
  let eq (x:t) (y:t) = (x=y)
end

(* A generic interface for
   a module with an equality
   function and a base type *)
module type Comp = sig
  type t
  val eq : t -> t -> bool
end
```

```
(* A set functor with signature
   ascription to abstract the set type *)
module Set = functor (A: Comp) -> (
struct
  type t = A.t
  type set = t list
  let empty_set : set = []
  let rec add x s : set = ...
end : sig
  type t
  type set (* abstract ! *)
  val empty_set : set
  val add : t -> set -> set
end)

(* Integer sets *)
module IntegerSet = Set(Integer)
```

The key mechanism: *construction* and *description* languages

The key mechanism: *construction* and *description* languages

Module Expressions

$M ::= P$		*Variables*
	$\mid M.X$	*Projection*
	$\mid (P : S)$	*Sealing*
	$\mid P_1(P_2)$	*Functor application*
	$\mid (X : S) \to M$	*Functor*
	$\mid \texttt{struct}_Y\ B\ \texttt{end}$	*Structure*

The key mechanism: *construction* and *description* languages

Module Expressions

$M ::= P$	*Variables*
$\mid M.X$	*Projection*
$\mid (P : S)$	*Sealing*
$\mid P_1(P_2)$	*Functor application*
$\mid (X : S) \rightarrow M$	*Functor*
$\mid \texttt{struct}_Y\ B\ \texttt{end}$	*Structure*

The key mechanism: *construction* and *description* languages

Module Expressions

$M ::= P$	*Variables*
$\mid M.X$	*Projection*
$\mid (P : S)$	*Sealing*
$\mid P_1(P_2)$	*Functor application*
$\mid (X : S) \rightarrow M$	*Functor*
$\mid \texttt{struct}_Y\ B\ \texttt{end}$	*Structure*

Bindings

$B ::= B; B$	*Sequence*
$\mid \text{let}\, x = E$	*Values*
$\mid \text{type}\, t = T$	*Types*
$\mid \text{module}\, X = M$	*Modules*
$\mid \text{module type}\, A = S$	*Module types*

The key mechanism: *construction* and *description* languages

Module Expressions

$M ::= P$		*Variables*
	$\mid M.X$	*Projection*
	$\mid (P : S)$	*Sealing*
	$\mid P_1(P_2)$	*Functor application*
	$\mid (X : S) \to M$	*Functor*
	$\mid \texttt{struct}_Y\ B\ \texttt{end}$	*Structure*

Bindings

$B ::= B; B$		*Sequence*
	$\mid \text{let}\ x = E$	*Values*
	$\mid \text{type}\ t = T$	*Types*
	$\mid \text{module}\ X = M$	*Modules*
	$\mid \text{module type}\ A = S$	*Module types*

Signatures

$S ::= P.A$		*Variables*
	$\mid (X : S_1) \to S_2$	*Functor*
	$\mid \texttt{sig}_Y\ \overline{D}\ \texttt{end}$	*Signature*

The key mechanism: *construction* and *description* languages

Module Expressions

$M ::= P$		*Variables*
$\mid M.X$		*Projection*
$\mid (P : S)$		*Sealing*
$\mid P_1(P_2)$		*Functor application*
$\mid (X : S) \rightarrow M$		*Functor*
$\mid \texttt{struct}_Y\ B\ \texttt{end}$		*Structure*

Signatures

$S ::= P.A$	*Variables*
$\mid (X : S_1) \rightarrow S_2$	*Functor*
$\mid \texttt{sig}_Y\ \overline{D}\ \texttt{end}$	*Signature*

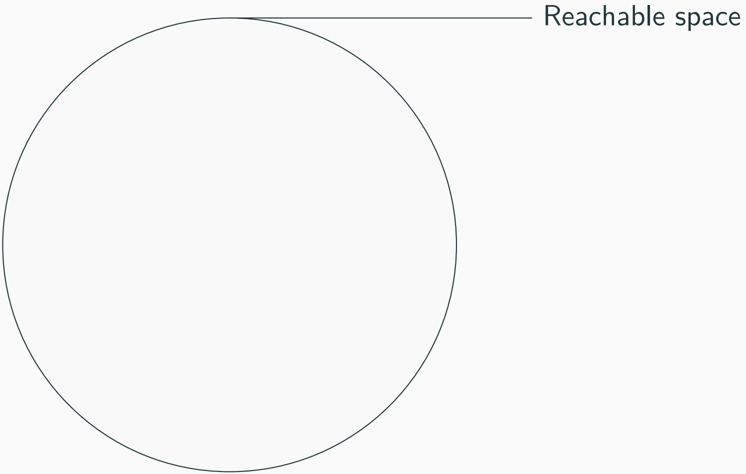
Bindings

$B ::= B; B$	*Sequence*
$\mid \text{let}\ x = E$	*Values*
$\mid \text{type}\ t = T$	*Types*
$\mid \text{module}\ X = M$	*Modules*
$\mid \text{module type}\ A = S$	*Module types*

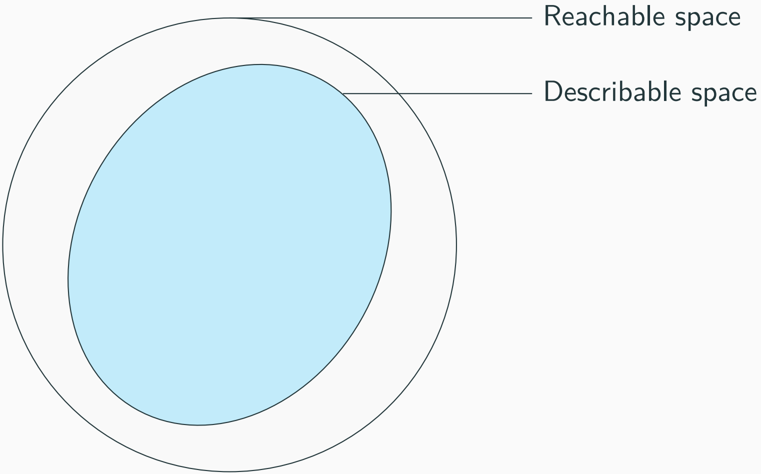
Declarations

$D ::= \text{val}\ x : T$	*Values*
$\mid \text{type}\ t = T$	*Types*
$\mid \text{type}\ t$	*Abstract types*
$\mid \text{module}\ X : S$	*Modules*
$\mid \text{module type}\ A = S$	*Module types*

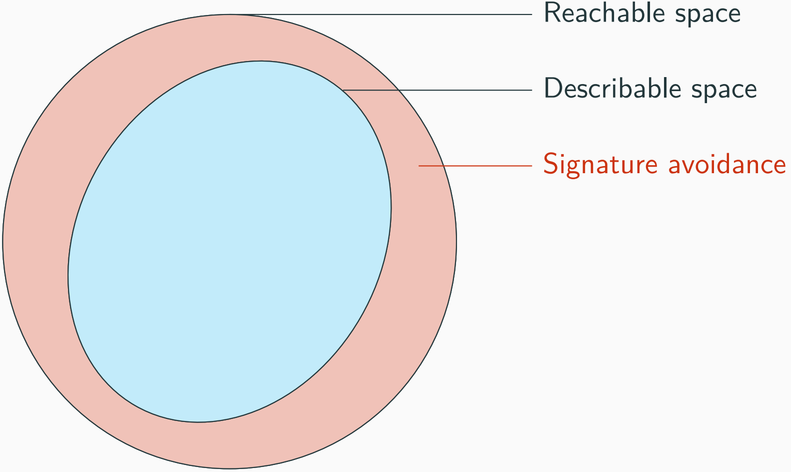
Reachable and *describable* spaces mismatch



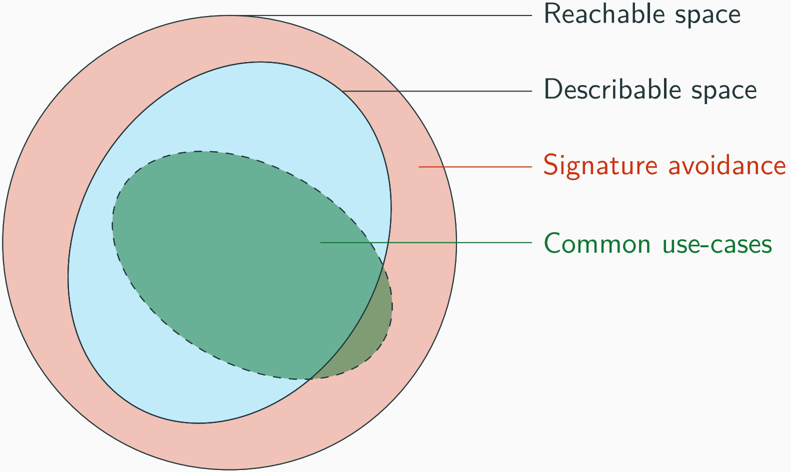
Reachable and *describable* spaces mismatch



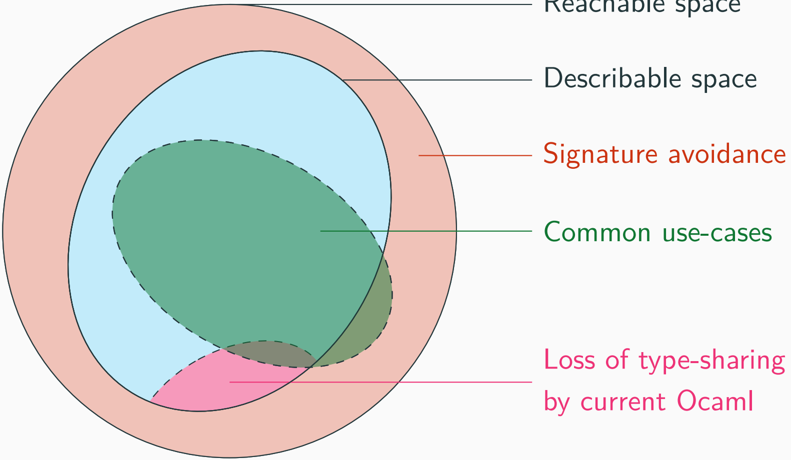
Reachable and *describable* spaces mismatch



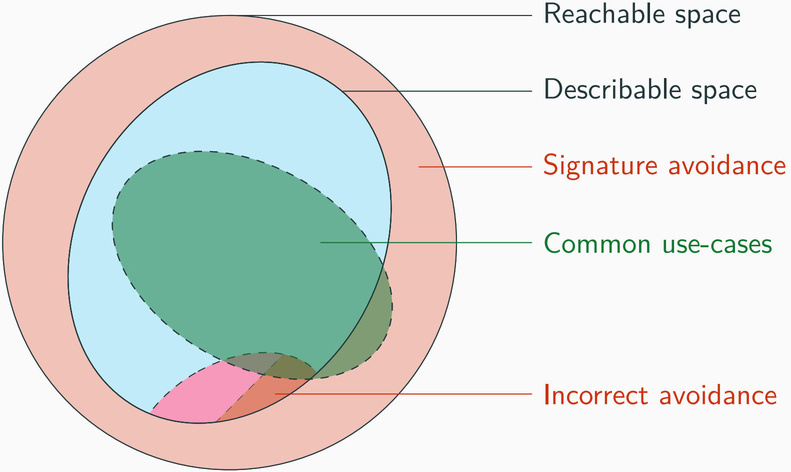
Reachable and *describable* spaces mismatch



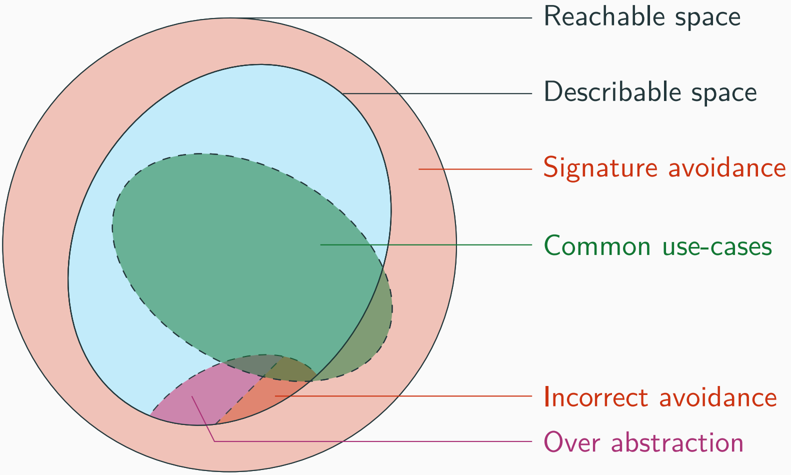
Reachable and *describable* spaces mismatch



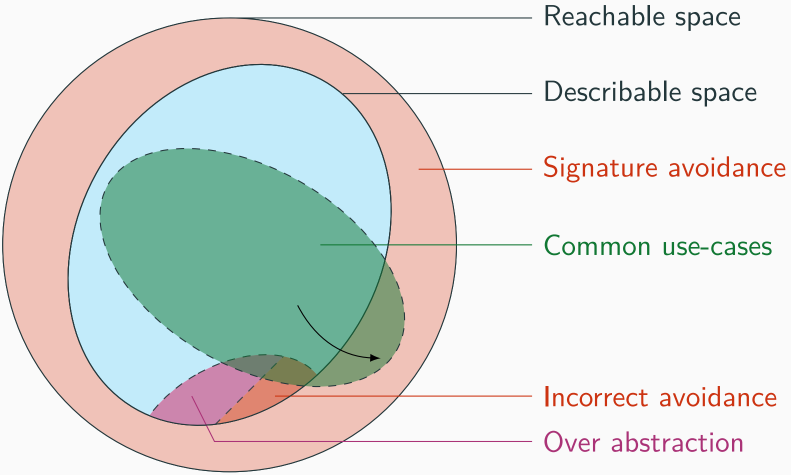
Reachable and *describable* spaces mismatch



Reachable and *describable* spaces mismatch

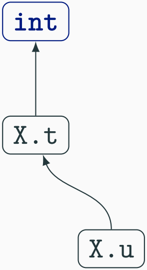


Reachable and *describable* spaces mismatch



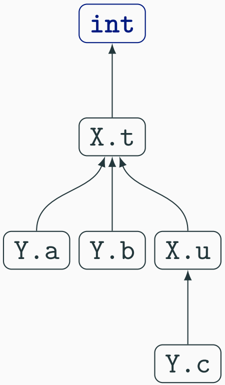
Signature avoidance and equivalence

```
module M =  struct






    module X =  struct
        type t = int
        type u = t
    end









end
```



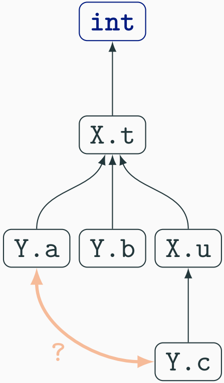
Signature avoidance and equivalence

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

    module Y : struct
        type a = X.t
        type b = X.t
        type c = X.u
    end

end
```



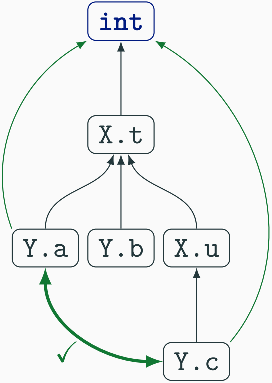
Signature avoidance and equivalence

```
module M =  struct






    module X =  struct
        type t = int
        type u = t
    end

    module Y : struct
        type a = X.t
        type b = X.t
        type c = X.u
    end

end

let f (x : M.Y.a) = (x : M.Y.c) (* ? *)
```



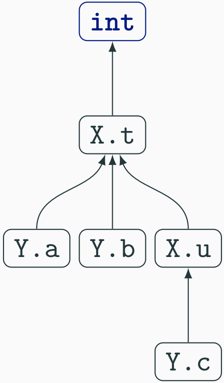
Signature avoidance and equivalence

```
module M =  struct






    module X =  struct
        type t = int
        type u = t
    end

    module Y : struct
        type a = X.t
        type b = X.t
        type c = X.u
    end

end

let f (x : M.Y.a) = (x : M.Y.c) (* ok *)
```



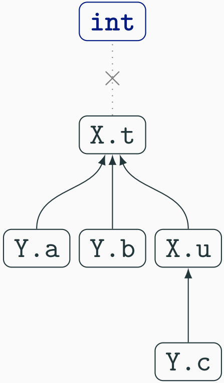
Signature avoidance and equivalence

```
module M =  struct

    module type S = sig
        type t
        type u = t
    end

    module X =  struct
        type t = int
        type u = t
    end

    module Y : struct
        type a = X.t
        type b = X.t
        type c = X.u
    end

end

let f (x : M.Y.a) = (x : M.Y.c)
```



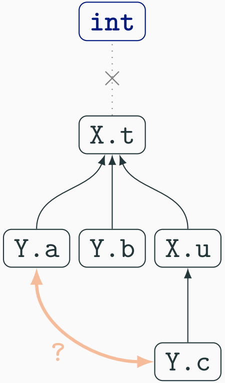
Signature avoidance and equivalence

```
module M =  struct

    module type S = sig
        type t
        type u = t
    end

    module X = (struct
        type t = int
        type u = t
    end : S)

    module Y : struct
        type a = X.t
        type b = X.t
        type c = X.u
    end

end

let f (x : M.Y.a) = (x : M.Y.c)
```



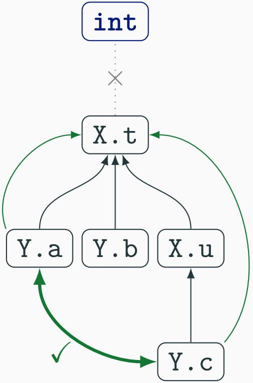
Signature avoidance and equivalence

```
module M =  struct

    module type S = sig
        type t
        type u = t
    end

    module X = (struct
        type t = int
        type u = t
    end : S)

    module Y : struct
        type a = X.t
        type b = X.t
        type c = X.u
    end

end

let f (x : M.Y.a) = (x : M.Y.c) (* ? *)
```



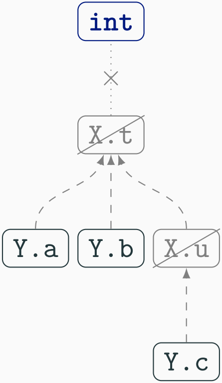
Signature avoidance and equivalence

```
module M =  struct

    module type S = sig
        type t
        type u = t
    end

    module X = (struct
        type t = int
        type u = t
    end : S)

    module Y : struct
        type a = X.t
        type b = X.t
        type c = X.u
    end

end

let f (x : M.Y.a) = (x : M.Y.c) (* ok *)
```



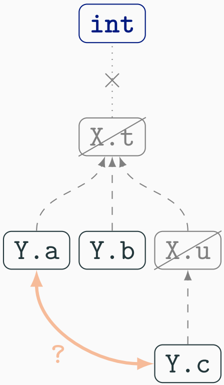
Signature avoidance and equivalence

```
module M = (struct

    module type S = sig
        type t
        type u = t
    end

    module X = (struct
        type t = int
        type u = t
    end : S)

    module Y : struct
        type a = X.t
        type b = X.t
        type c = X.u
    end

end).Y

let f (x : M.a) = (x : M.c)
```



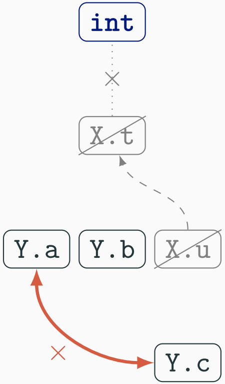
Signature avoidance and equivalence

```
module M = (struct

    module type S = sig
        type t
        type u = t
    end

    module X = (struct
        type t = int
        type u = t
    end : S)

    module Y : struct
        type a = X.t
        type b = X.t
        type c = X.u
    end

end).Y

let f (x : M.a) = (x : M.c) (* ? *)
```



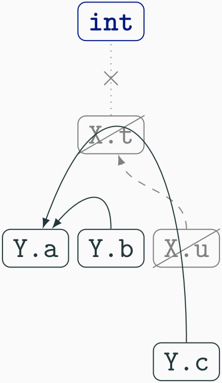
Signature avoidance and equivalence

```
module M = (struct

    module type S = sig
        type t
        type u = t
    end

    module X = (struct
        type t = int
        type u = t
    end : S)

    module Y : struct
        type a
        type b
        type c
    end

end).Y

let f (x : M.a) = (x : M.c) (* error *)
```



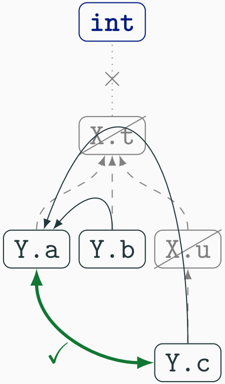
Signature avoidance and equivalence

```
module M = (struct

    module type S = sig
        type t
        type u = t
    end

    module X = (struct
        type t = int
        type u = t
    end : S)

    module Y : struct
        type a
        type b = a
        type c = a
    end

end).Y

let f (x : M.a) = (x : M.c)
```



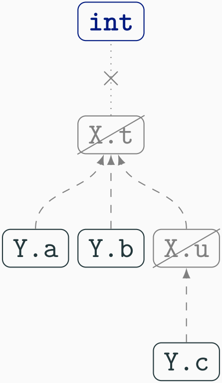
Signature avoidance and equivalence

```
module M = (struct

    module type S = sig
        type t
        type u = t
    end

    module X = (struct
        type t = int
        type u = t
    end : S)

    module Y : struct
        type a
        type b = a
        type c = a
    end

end).Y

let f (x : M.a) = (x : M.c) (* ok *)
```



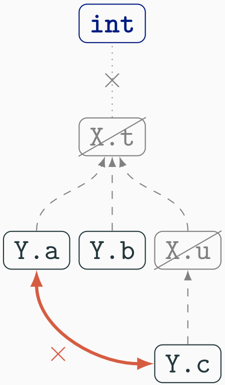
Signature avoidance - unsolvable cases

```
module M = (struct

    module type S = sig
        type t
        type u = t
    end

    module X = (struct
        type t = int
        type u = t
    end : S)

    module Y : struct
        type a = X.t * int
        type b = X.t -> string
        type c = X.u * bool
    end

end).Y

let f (x : M.a) = ((fst x, true) : M.c)
```



Signature avoidance - unsolvable cases

```
module M = (struct

    module type S = sig
        type t
        type u = t
    end

    module X = (struct
        type t = int
        type u = t
    end : S)

    module Y : struct
        type a = X.t * int
        type b = X.t -> string
        type c = X.u * bool
    end

end).Y

let f (x : M.a) = ((fst x, true) : M.c) (* error *)
```

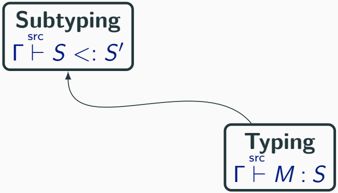


A presentation of the OCaml module system

Typing

$\Gamma \overset{\text{src}}{\vdash} M : S$

A presentation of the OCaml module system



A presentation of the OCaml module system

S-Typ-Sig

$$\frac{\Gamma \overset{\text{src}}{\vdash} P : S' \qquad \Gamma \overset{\text{src}}{\vdash} S' <: S}{\Gamma \overset{\text{src}}{\vdash} (P : S) : S}$$

A presentation of the OCaml module system

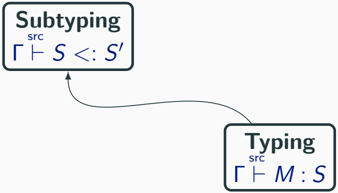
S-Typ-Sig

$$\frac{\Gamma \overset{\text{src}}{\vdash} P : S' \qquad \Gamma \overset{\text{src}}{\vdash} S' <: S}{\Gamma \overset{\text{src}}{\vdash} (P : S) : S}$$

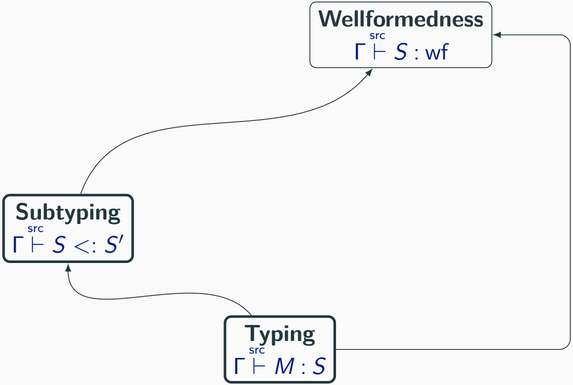
S-Typ-Fct

$$\frac{X \notin \Gamma \qquad \Gamma; \text{module}\, X : S_a \overset{\text{src}}{\vdash} M : S_r}{\Gamma \overset{\text{src}}{\vdash} ((X : S_a) \to M) : (X : S_a) \to S_r}$$

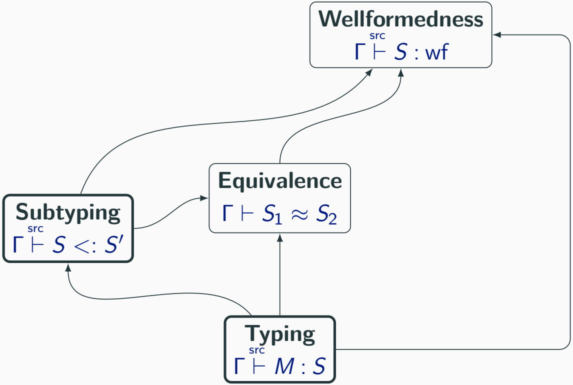
A presentation of the OCaml module system



A presentation of the OCaml module system



A presentation of the OCaml module system



A presentation of the OCaml module system

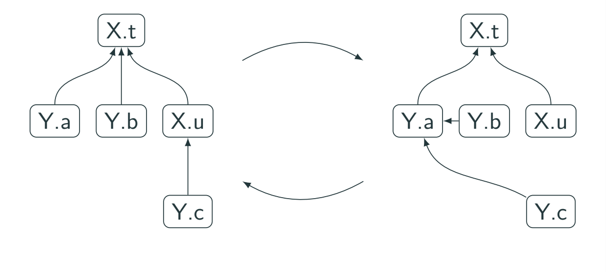
S-Eqv-Type

$$\frac{\Gamma \vdash T_1 \approx T_2}{\Gamma \vdash_Y (\text{type}\, t = T_1) \approx (\text{type}\, t = T_2)}$$

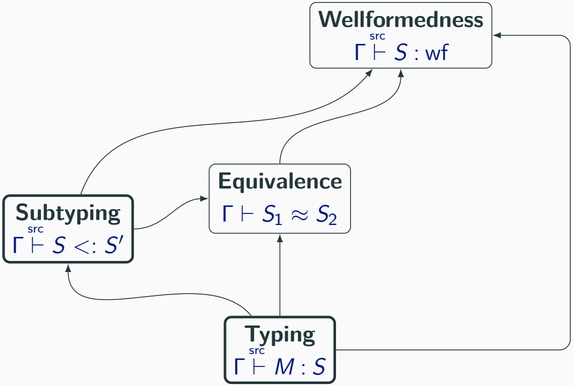
A presentation of the OCaml module system

S-Eqv-Type

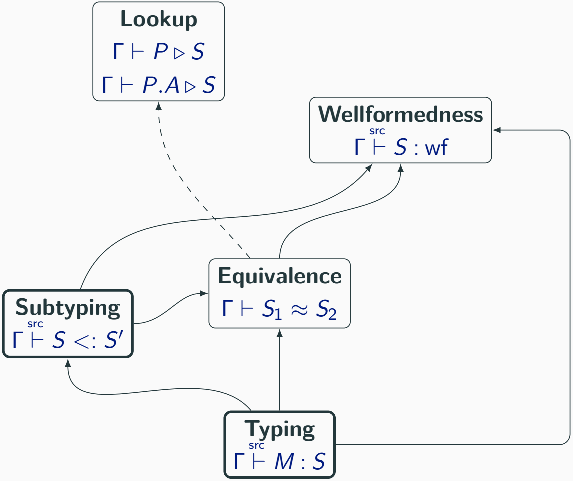
$$\frac{\Gamma \vdash T_1 \approx T_2}{\Gamma \vdash_Y (\text{type}\ t = T_1) \approx (\text{type}\ t = T_2)}$$



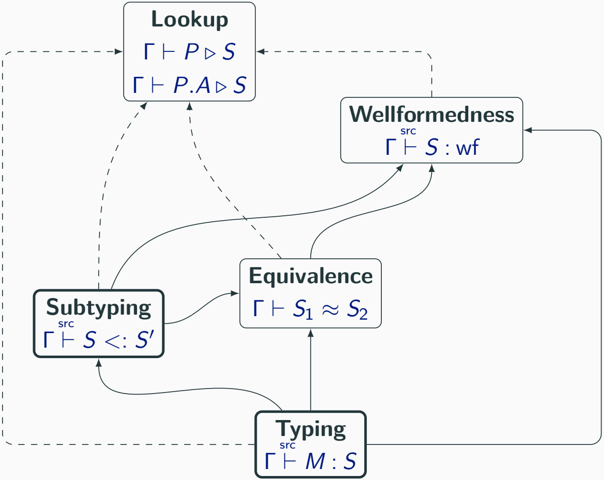
A presentation of the OCaml module system



A presentation of the OCaml module system



A presentation of the OCaml module system



A presentation of the OCaml module system

S-Lkp-Alias

$$\frac{\Gamma \vdash P \triangleright P'.A \qquad \Gamma \vdash P'.A \triangleright S}{\Gamma \vdash P \triangleright S}$$

A presentation of the OCaml module system

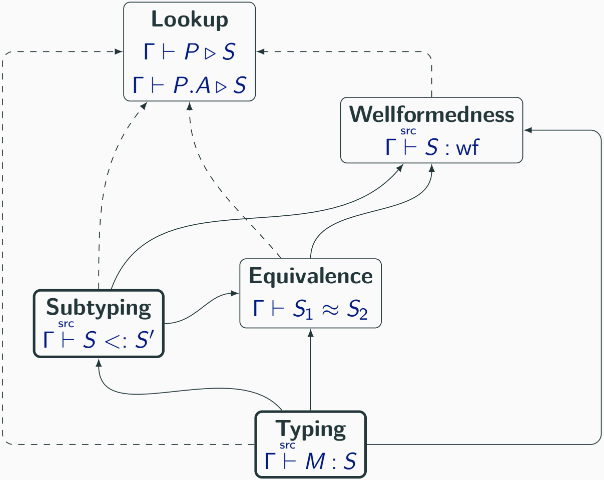
S-Lkp-Alias

$$\frac{\Gamma \vdash P \triangleright P'.A \qquad \Gamma \vdash P'.A \triangleright S}{\Gamma \vdash P \triangleright S}$$

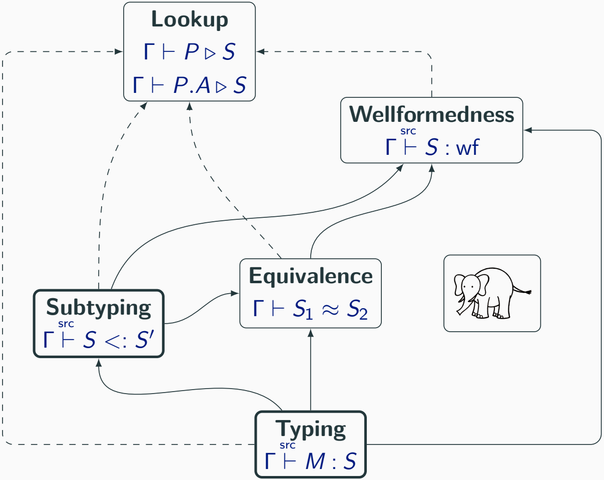
S-Lkp-Proj-Mod

$$\frac{\Gamma \vdash P \triangleright \texttt{sig}_Y\ \overline{D}\ \texttt{end} \qquad (\text{module}\ X : S) \in \overline{D}}{\Gamma \vdash P.X \triangleright S[Y \mapsto P]}$$

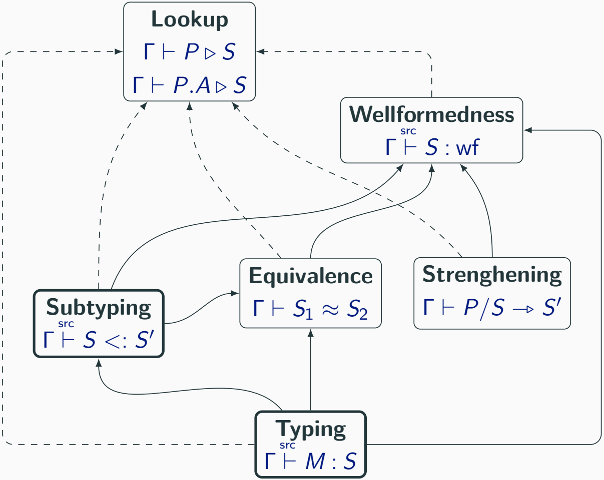
A presentation of the OCaml module system



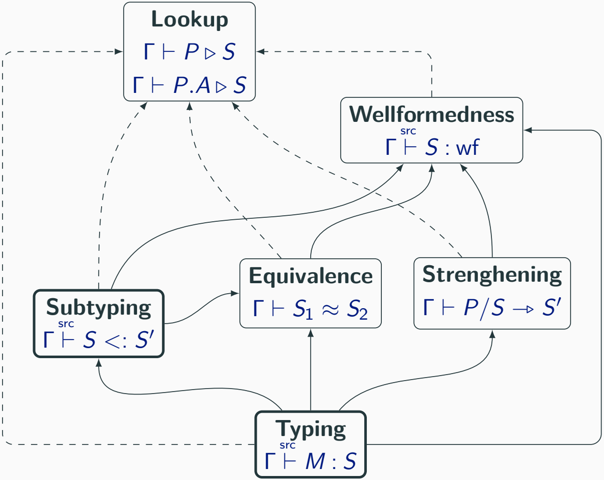
A presentation of the OCaml module system



A presentation of the OCaml module system

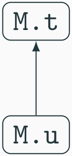


A presentation of the OCaml module system



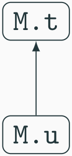
A Note on Strengthening

```
module M = (... : sig
    type t
    type u = t
end)
```



A Note on Strengthening

```
module M = (... : sig
    type t
    type u = t
end)

module N = struct
    include M
end
```

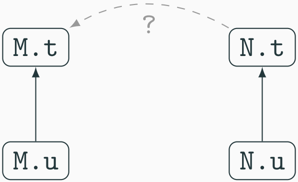


A Note on Strengthening

```
module M = (... : sig
    type t
    type u = t
end)

module N = struct
    include M
end (* : sig
    type t
    type u = t
*)
```

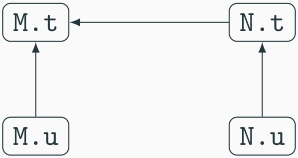
A Note on Strengthening

```
module M = (... : sig
    type t
    type u = t
end)

module N = struct
    include M
end (* : sig
    type t
    type u = t
*)

(* M.t =? N.t *)
```



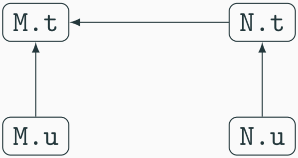
A Note on Strengthening

```
module M = (... : sig
    type t
    type u = t
end)

module N = struct
    include M
end (* : sig
    type t = M.t
    type u = t
*)

(* M.t =? N.t *)
```



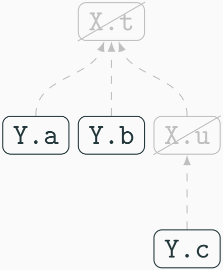
A Note on Strengthening

```
module M = (... : sig
    type t
    type u = t
end)

module N = struct
    include M
end (* : sig
    type t = M.t
    type u = t
*)

(* M.t =? N.t *)
```



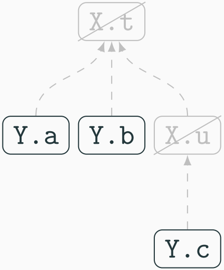
Simplifying and improving with the canonical system

Simplifying equivalence



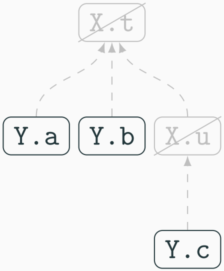
Simplifying equivalence

- Representing equalities through single links in an alias tree is **fragile**



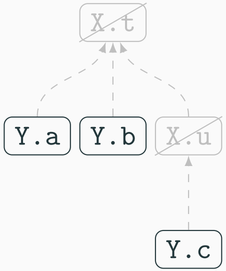
Simplifying equivalence

- Representing equalities through single links in an alias tree is **fragile**
- Only **connected components** matter

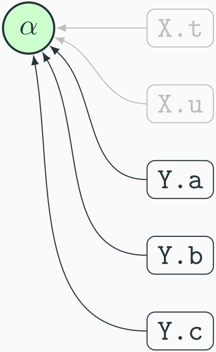


Simplifying equivalence

- Representing equalities through single links in an alias tree is **fragile**
- Only **connected components** matter
- Identify **where** abstract types are created



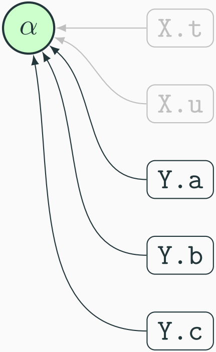
Existential types $\exists\alpha$



Existential types

Existential types $\exists\alpha$

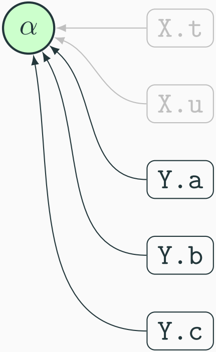
- Identify where abstract types (new types) are created



Existential types

Existential types $\exists\alpha$

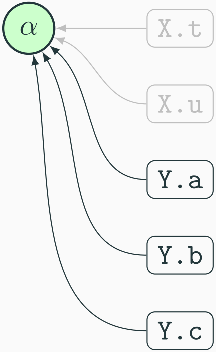
- Identify where abstract types (new types) are created
- Give a *path-independent* way to refer to a type (type $t = \alpha$)



Existential types

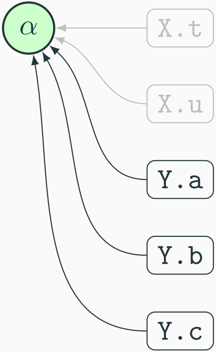
Existential types $\exists\alpha$

- Identify where abstract types (new types) are created
- Give a *path-independent* way to refer to a type (type $t = \alpha$)
- Logical meaning of existential (no introspection)



Existential types $\exists\alpha$

- Identify where abstract types (new types) are created
- Give a *path-independent* way to refer to a type (type $t = \alpha$)
- Logical meaning of existential (no introspection)
- No naming issues (α-convertibility)



The idea of existential types

ML-Modules rich history

The idea of existential types

ML-Modules rich history

- SML modules Harper et al. [1989]

The idea of existential types

ML-Modules rich history

- SML modules Harper et al. [1989]
- First formalizations of OCaml Modules: Leroy [2000], Leroy [1995]

The idea of existential types

ML-Modules rich history

- SML modules Harper et al. [1989]
- First formalizations of OCaml Modules: Leroy [2000], Leroy [1995]

Existential types for modules

The idea of existential types

ML-Modules rich history

- SML modules Harper et al. [1989]
- First formalizations of OCaml Modules: Leroy [2000], Leroy [1995]

Existential types for modules

- First with dependent types MacQueen [1986]

The idea of existential types

ML-Modules rich history

- SML modules Harper et al. [1989]
- First formalizations of OCaml Modules: Leroy [2000], Leroy [1995]

Existential types for modules

- First with dependent types MacQueen [1986]
- Return of existential types Russo [2004]

The idea of existential types

ML-Modules rich history

- SML modules Harper et al. [1989]
- First formalizations of OCaml Modules: Leroy [2000], Leroy [1995]

Existential types for modules

- First with dependent types MacQueen [1986]
- Return of existential types Russo [2004]
- Existentials for mutually recursive modules DREYER [2007] and open existentials Montagu and Rémy [2009]

The idea of existential types

ML-Modules rich history

- SML modules Harper et al. [1989]
- First formalizations of OCaml Modules: Leroy [2000], Leroy [1995]

Existential types for modules

- First with dependent types MacQueen [1986]
- Return of existential types Russo [2004]
- Existentials for mutually recursive modules DREYER [2007] and open existentials Montagu and Rémy [2009]

Formalizing by elaboration

The idea of existential types

ML-Modules rich history

- SML modules Harper et al. [1989]
- First formalizations of OCaml Modules: Leroy [2000], Leroy [1995]

Existential types for modules

- First with dependent types MacQueen [1986]
- Return of existential types Russo [2004]
- Existentials for mutually recursive modules DREYER [2007] and open existentials Montagu and Rémy [2009]

Formalizing by elaboration

- **A significant part of SML translated into F^ω : F-ing Rossberg et al. [2014]**

The idea of existential types

ML-Modules rich history

- SML modules Harper et al. [1989]
- First formalizations of OCaml Modules: Leroy [2000], Leroy [1995]

Existential types for modules

- First with dependent types MacQueen [1986]
- Return of existential types Russo [2004]
- Existentials for mutually recursive modules DREYER [2007] and open existentials Montagu and Rémy [2009]

Formalizing by elaboration

- **A significant part of SML translated into F^{ω} : F-ing Rossberg et al. [2014]**
- Core and module united 1ML Rossberg [2018]

The *canonical* system - Grammar extensions

Canonical Types

$$\tau ::= \ \ldots \mid \alpha$$ *Existential identifier*

The *canonical* system - Grammar extensions

Canonical Types

$$\tau ::= \ \ldots \mid \alpha$$ *Existential identifier*

Canonical abstract signatures

$$\mathcal{C} ::= \exists \overline{\alpha}.\mathcal{R}$$ *Abstract signature*

The *canonical* system - Grammar extensions

Canonical Types

$$\tau ::= \ldots \mid \alpha \qquad \textit{Existential identifier}$$

Canonical abstract signatures

$$\mathcal{C} ::= \exists \overline{\alpha}.\mathcal{R} \qquad \textit{Abstract signature}$$

Canonical manifest signatures

$$\mathcal{R} ::= \forall \overline{\alpha}.\,(X : \mathcal{R}) \to \mathcal{C} \qquad \textit{Functor}$$

$$\mid \texttt{sig}_Y\ \overline{\mathcal{D}}\ \texttt{end} \qquad \textit{Signature}$$

The *canonical* system - Grammar extensions

Canonical Types

$$\tau ::= \ \dots \mid \alpha$$ *Existential identifier*

Canonical abstract signatures

$$\mathcal{C} ::= \exists \overline{\alpha}.\mathcal{R}$$ *Abstract signature*

Canonical manifest signatures

$$\mathcal{R} ::= \forall \overline{\alpha}. (X : \mathcal{R}) \to \mathcal{C}$$ *Functor*

$$\mid \texttt{sig}_Y \ \overline{\mathcal{D}} \ \texttt{end}$$ *Signature*

Canonical declarations

$$\mathcal{D} ::= \ \mathsf{val}\, x : (\approx \tau)$$ *Values*

$$\mid \ \mathsf{type}\, t \approx \tau$$ *Types*

$$\mid \ \mathsf{module}\, X : \mathcal{R}$$ *Modules*

$$\mid \ \mathsf{module\ type}\, A = \mathcal{C}$$ *Module types*

The *canonical* system - Grammar extensions

Canonical Types

$$\tau ::= \ \dots \mid \alpha \qquad \textit{Existential identifier}$$

Canonical abstract signatures

$$\mathcal{C} ::= \exists \overline{\alpha}.\mathcal{R} \qquad \textit{Abstract signature}$$

Canonical manifest signatures

$$\begin{aligned} \mathcal{R} ::= & \ \forall \overline{\alpha}.\, (X : \mathcal{R}) \to \mathcal{C} && \textit{Functor} \\ \mid & \ \texttt{sig}_Y\ \overline{\mathcal{D}}\ \texttt{end} && \textit{Signature} \end{aligned}$$

Canonical declarations

$$\begin{aligned} \mathcal{D} ::= & \ \mathsf{val}\, x : (\approx \tau) && \textit{Values} \\ \mid & \ \mathsf{type}\, t \approx \tau && \textit{Types} \\ \mid & \ \mathsf{module}\, X : \mathcal{R} && \textit{Modules} \\ \mid & \ \mathsf{module\ type}\, A = \mathcal{C} && \textit{Module types} \end{aligned}$$

But same module expressions !

The *canonical* system - Grammar extensions

Canonical Types

$$\tau ::= \ \ldots \mid \alpha$$ *Existential identifier*

Canonical abstract signatures

$$\mathcal{C} ::= \exists \overline{\alpha}.\mathcal{R}$$ *Abstract signature*

Canonical manifest signatures

$$\begin{aligned} \mathcal{R} ::= & \ \forall \overline{\alpha}. (X : \mathcal{R}) \to \mathcal{C} & \textit{Functor} \\ & \mid \texttt{sig}_Y \ \overline{\mathcal{D}} \ \texttt{end} & \textit{Signature} \end{aligned}$$

Canonical declarations

$$\begin{aligned} \mathcal{D} ::= & \ \mathsf{val}\, x : (\approx \tau) & \textit{Values} \\ & \mid \mathsf{type}\, t \approx \tau & \textit{Types} \\ & \mid \mathsf{module}\, X : \mathcal{R} & \textit{Modules} \\ & \mid \mathsf{module\ type}\, A = \mathcal{C} & \textit{Module types} \end{aligned}$$

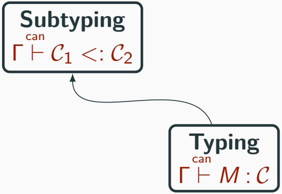
But same module expressions !

The *canonical* system - Judgments

Typing

$\Gamma \overset{\text{can}}{\vdash} M : \mathcal{C}$

The *canonical* system - Judgments



The *canonical* system - Judgments

C-Typ-Proj

$$\frac{\Gamma \overset{\text{can}}{\vdash} M : \exists\overline{\alpha}.\texttt{sig}_{Y}\ \overline{\mathcal{D}}_1\ ,\ \text{module}\ X : \mathcal{R}\ ,\overline{\mathcal{D}}_2\ \texttt{end} \qquad \Gamma \overset{\text{can}}{\vdash} \exists\overline{\alpha}.\mathcal{R} : \text{wf}}{\Gamma \overset{\text{can}}{\vdash} M.X : \exists\overline{\alpha}.\mathcal{R}}$$

The *canonical* system - Judgments

C-Typ-Proj

$$\frac{\Gamma \overset{\text{can}}{\vdash} M : \exists\overline{\alpha}.\mathtt{sig}_{Y}\ \overline{\mathcal{D}}_1\ ,\ \text{module}\ X : \mathcal{R}\ ,\overline{\mathcal{D}}_2\ \mathtt{end} \qquad \Gamma \overset{\text{can}}{\vdash} \exists\overline{\alpha}.\mathcal{R} : \text{wf}}{\Gamma \overset{\text{can}}{\vdash} M.X : \exists\overline{\alpha}.\mathcal{R}}$$

The *canonical* system - Judgments

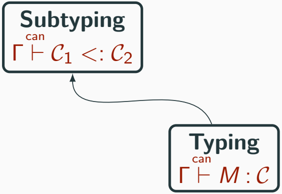
C-Typ-Proj

$$\frac{\Gamma \overset{\text{can}}{\vdash} M : \exists\overline{\alpha}.\mathtt{sig}_Y\ \overline{\mathcal{D}}_1\ ,\ \text{module}\, X : \mathcal{R}\ ,\overline{\mathcal{D}}_2\ \mathtt{end} \qquad \Gamma \overset{\text{can}}{\vdash} \exists\overline{\alpha}.\mathcal{R} : \text{wf}}{\Gamma \overset{\text{can}}{\vdash} M.X : \exists\overline{\alpha}.\mathcal{R}}$$

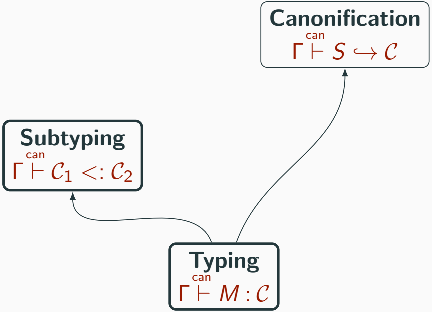
C-Typ-Sig

$$\frac{\Gamma \overset{\text{can}}{\vdash} S \hookrightarrow \mathcal{C} \qquad \Gamma \overset{\text{can}}{\vdash} P : \mathcal{R} \qquad \Gamma \overset{\text{can}}{\vdash} \mathcal{R} <: \mathcal{C}}{\Gamma \overset{\text{can}}{\vdash} (P : S) : \mathcal{C}}$$

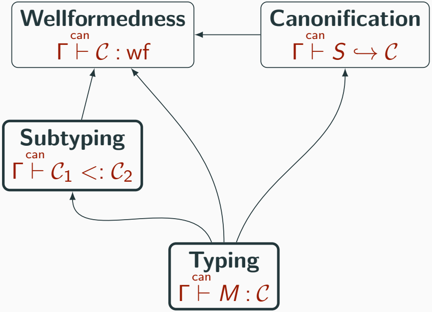
The *canonical* system - Judgments



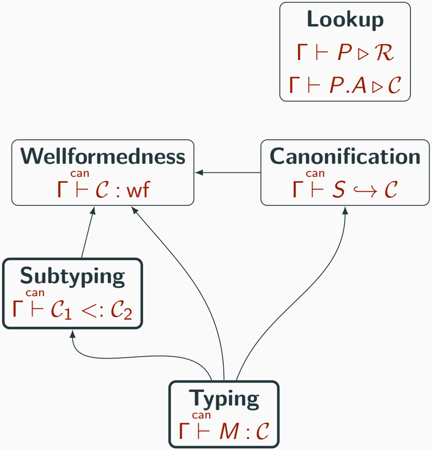
The *canonical* system - Judgments



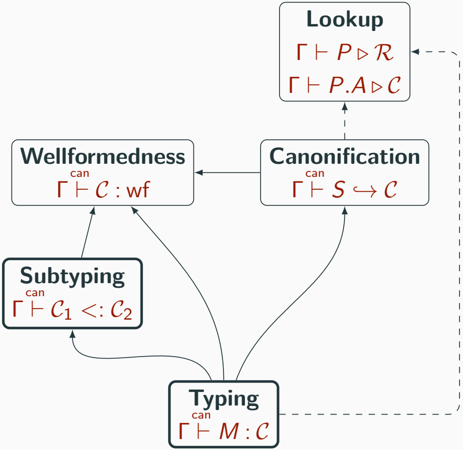
The *canonical* system - Judgments



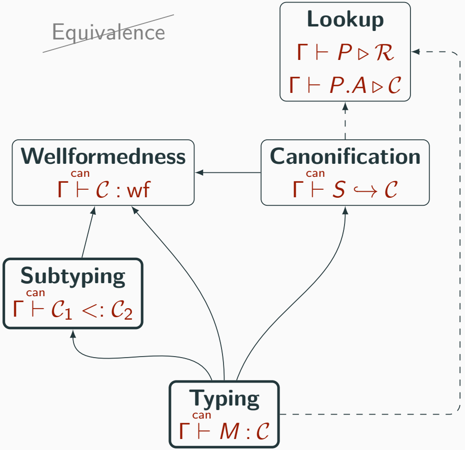
The *canonical* system - Judgments



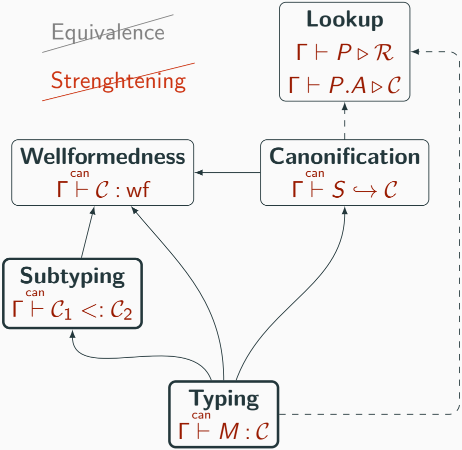
The *canonical* system - Judgments



The *canonical* system - Judgments



The *canonical* system - Judgments



Canonification and anchorability: intuitions of the canonical system towards the source one

Canonification results

- The canonical system is more powerful than the source one

Canonification results

- The canonical system is more powerful than the source one
- The canonical system keeps all type-sharing, by design, whereas the source system can lose some links

Canonification results

- The canonical system is more powerful than the source one
- The canonical system keeps all type-sharing, by design, whereas the source system can lose some links

Theorem (Canonification of typing)

Given any environment Γ*, module* M*, and signature* S*, we have:*

$$\Gamma \overset{src}{\vdash} M : S \implies \exists \mathcal{C}', \begin{cases} \Gamma^c \overset{can}{\vdash} M : \mathcal{C}' \\ \Gamma^c \overset{can}{\vdash} \mathcal{C}' \prec: \mathcal{C}(S) \end{cases} \quad 1$$

Anchorability : existence vs identity - example

```
module M = (struct
  ...
  module Y : struct
    type a = X.t
    type b = X.t
    type c = X.u
    end
end).Y
```

```
module M = (struct
  ...
  module Y : struct
    type a = X.t -> string
    type b = X.t -> int
    type c = X.u * bool
    end
end).Y
```

Anchorability : existence vs identity - example

```
module M = (struct
  ...
  module Y : struct
    type a = X.t
    type b = X.t
    type c = X.u
    end
end).Y


(* M.Y : ∃α. sig
  type a = α
  type b = α
  type c = α
  end *)
```

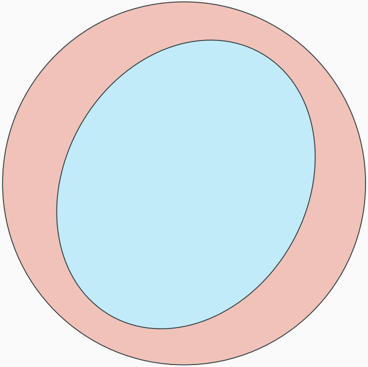
```
module M = (struct
  ...
  module Y : struct
    type a = X.t -> string
    type b = X.t -> int
    type c = X.u * bool
    end
end).Y
```

Anchorability : existence vs identity - example

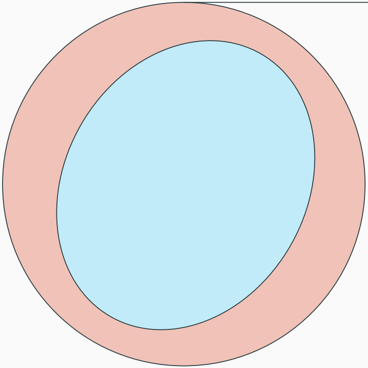
```
module M = (struct
  ...
  module Y : struct
    type a = X.t
    type b = X.t
    type c = X.u
    end
end).Y


(* M.Y : ∃α. sig
  type a = α
  type b = α
  type c = α
  end *)
```

```
module M = (struct
  ...
  module Y : struct
    type a = X.t -> string
    type b = X.t -> int
    type c = X.u * bool
    end
end).Y


(* M.Y : ∃α. sig
  type a = α -> string
  type b = α -> int
  type c = α * bool
  end *)
```

Anchorability : existence vs identity

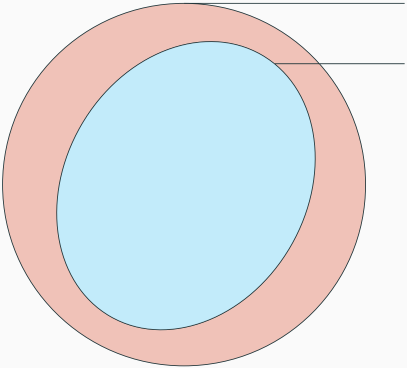


Anchorability : existence vs identity

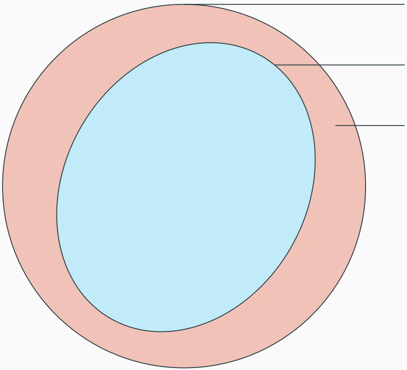


type components **exist** (are used)

Anchorability : existence vs identity



Anchorability : existence vs identity

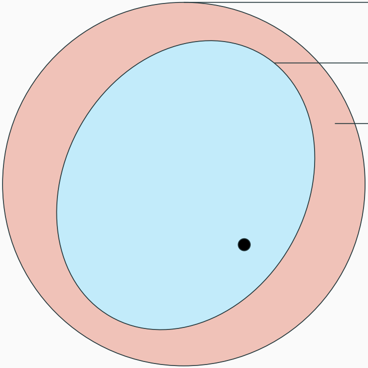


type components **exist** (are used)

type components have an **identity** (an alias)

some type components are **used** but no alias is in scope

Anchorability : existence vs identity

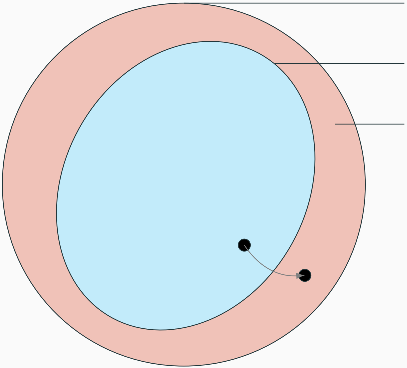


type components **exist** (are used)

type components have an **identity** (an alias)

some type components are
used but no alias is in scope

Anchorability : existence vs identity



type components **exist** (are used)

type components have an **identity** (an alias)

some type components are
used but no alias is in scope

Anchorability results

- Restricting the use of existential types

Anchorability results

- Restricting the use of existential types
- Storing only existentials with a path

Anchorability results

- Restricting the use of existential types
- Storing only existentials with a path

Theorem (Anchorable typing)

Given any environment Γ, module expression M and (canonical) signature $\mathcal{C}$, we have:

$$\Gamma \vdash_a M : \mathcal{C} \implies \exists \Gamma_s, S, \begin{cases} \Gamma_s \hookrightarrow \Gamma \\ \Gamma \overset{can}{\vdash} S \hookrightarrow \mathcal{C} \\ \Gamma_s \overset{src}{\vdash} M : S \end{cases} \tag{2}$$

Anchorability results

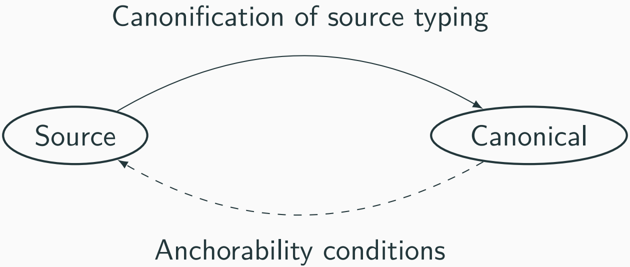
- Restricting the use of existential types
- Storing only existentials with a path

Theorem (Anchorable typing)

Given any environment Γ, module expression M and (canonical) signature $\mathcal{C}$, we have:

$$\Gamma \vdash_a M : \mathcal{C} \implies \exists \Gamma_s, S, \begin{cases} \Gamma_s \hookrightarrow \Gamma \\ \Gamma \overset{can}{\vdash} S \hookrightarrow \mathcal{C} \\ \Gamma_s \overset{src}{\vdash} M : S \end{cases}$$ 2

Canonical and source system links



Elaboration into F^ω: guarantees for the canonical system

Standard F^ω with records and existential types

$$\kappa ::= \star \mid \kappa \to \kappa \qquad \textit{kinds}$$

$$\tau ::= \alpha \mid \tau \to \tau \mid \{\overline{l : \tau}\} \mid \forall \alpha : \kappa.\tau \mid \exists \alpha : \kappa.\tau \mid \lambda \alpha : \kappa.\tau \mid \tau\,\tau \qquad \textit{types}$$

$$e ::= x \mid \lambda x : \tau.e \mid e\,e \mid \{\overline{l : e}\} \mid e.l \mid \Lambda \alpha : \kappa.e \mid e\,\tau$$

$$\mid \mathsf{pack}\,\langle \tau, e \rangle_\tau \mid \mathsf{unpack}\,\langle \alpha, x \rangle = e\,\mathsf{in}\,e \qquad \textit{terms}$$

$$v ::= \lambda x : \tau.e \mid \{\overline{l : v}\} \mid \lambda \alpha : \kappa.e \mid \mathsf{pack}\,\langle \tau, v \rangle_\tau \qquad \textit{values}$$

$$\Theta ::= \cdot \mid \Theta, \alpha : \kappa \mid \Theta, x : \tau \qquad \textit{environments}$$

Standard F^ω with records and existential types

$$\kappa ::= \star \mid \kappa \to \kappa \qquad \textit{kinds}$$

$$\tau ::= \alpha \mid \tau \to \tau \mid \{\overline{l : \tau}\} \mid \forall \alpha : \kappa.\tau \mid \exists \alpha : \kappa.\tau \mid \lambda \alpha : \kappa.\tau \mid \tau\, \tau \qquad \textit{types}$$

$$e ::= x \mid \lambda x : \tau.e \mid e\, e \mid \{\overline{l : e}\} \mid e.l \mid \Lambda \alpha : \kappa.e \mid e\, \tau$$

$$\mid \text{pack}\,\langle \tau, e \rangle_\tau \mid \text{unpack}\,\langle \alpha, x \rangle = e \,\text{in}\, e \qquad \textit{terms}$$

$$v ::= \lambda x : \tau.e \mid \{\overline{l : v}\} \mid \lambda \alpha : \kappa.e \mid \text{pack}\,\langle \tau, v \rangle_\tau \qquad \textit{values}$$

$$\Theta ::= \cdot \mid \Theta, \alpha : \kappa \mid \Theta, x : \tau \qquad \textit{environments}$$

No subtyping !

Encoding signatures into F^{ω} terms/types

Signatures

Encoding signatures into F^ω terms/types

Signatures

$\Pi := \exists\overline{\alpha}.\Sigma$ *abstract signatures*

Encoding signatures into F^ω terms/types

Signatures

$$\Pi := \exists\overline{\alpha}.\Sigma \qquad \textit{abstract signatures}$$

$$\Sigma := [\![\tau]\!] \mid [\![= \tau : \star]\!] \mid [\![= \Pi]\!] \mid \left\{\overline{l_X : \Sigma}\right\} \mid \forall\alpha.\Sigma \to \Pi \qquad \textit{concrete signatures}$$

Encoding signatures into F^ω terms/types

Signatures

$$\Pi := \exists\overline{\alpha}.\Sigma \qquad \textit{abstract signatures}$$

$$\Sigma := [\![\tau]\!] \mid [\![= \tau : \star]\!] \mid [\![= \Pi]\!] \mid \left\{\overline{l_X : \Sigma}\right\} \mid \forall\alpha.\Sigma \to \Pi \qquad \textit{concrete signatures}$$

Types

$$[\![\tau]\!] \triangleq \{\mathsf{val} : \tau\} \qquad \textit{value}$$

$$[\![= \tau : \star]\!] \triangleq \{\mathsf{typ} : \forall\alpha : (\star \to \star)\,.\alpha\ \tau \to \alpha\ \tau\} \qquad \textit{type}$$

$$[\![= \Pi]\!] \triangleq \{\mathsf{sig} : \Pi \to \Pi\} \qquad \textit{module type}$$

Encoding signatures into F^ω terms/types

Signatures

$$\Pi := \exists\overline{\alpha}.\Sigma \qquad \textit{abstract signatures}$$

$$\Sigma := [\![\tau]\!] \mid [\![= \tau : \star]\!] \mid [\![= \Pi]\!] \mid \left\{\overline{l_X : \Sigma}\right\} \mid \forall\alpha.\Sigma \to \Pi \qquad \textit{concrete signatures}$$

Types

$$[\![\tau]\!] \triangleq \{\mathsf{val} : \tau\} \qquad \textit{value}$$

$$[\![= \tau : \star]\!] \triangleq \{\mathsf{typ} : \forall\alpha : (\star \to \star).\alpha\ \tau \to \alpha\ \tau\} \qquad \textit{type}$$

$$[\![= \Pi]\!] \triangleq \{\mathsf{sig} : \Pi \to \Pi\} \qquad \textit{module type}$$

Terms

$$[\![e]\!] \triangleq \{\mathsf{val} = e\} \qquad \textit{value}$$

$$[\![\tau : \star]\!] \triangleq \{\mathsf{typ} = \lambda\alpha : (\star \to \star).\lambda x : \alpha\ \tau.x\} \qquad \textit{type}$$

$$[\![\Pi]\!] \triangleq \{\mathsf{sig} = \lambda x : \Pi.x\} \qquad \textit{module type}$$

Encoding example

```
module M = struct









end)
```

Encoding example

```
module M = struct
   module X =  struct
      type t = int
      let x = 42
   end




end)
```

Encoding example

```
module M = struct
   module X = (struct
      type t = int
      let x = 42
   end : sig
      type t
      val x : t
   end)

end)
```

Encoding example

```
module M = struct
   module X = (struct
      type t = int
      let x = 42
   end : sig
      type t
      val x : t
   end)
   type u = X.t * bool
end)
```

Encoding example

```
module M = struct
   module X = (struct
      type t = int
      let x = 42
   end : sig
      type t
      val x : t
   end)
   type u = X.t * bool
end)
```

Signature

Encoding example

```
module M = struct
  module X = (struct
    type t = int
    let x = 42
  end : sig
    type t
    val x : t
  end)
  type u = X.t * bool
end)
```

Signature

$$\exists \alpha . \left\{ \begin{array}{l} l_X : \left\{ \begin{array}{l} l_t : [\![= \alpha : \star]\!] \\ l_x : [\![\alpha]\!] \end{array} \right\} \\ l_u : [\![= \alpha \times \texttt{bool} : \star]\!] \end{array} \right\}$$

Encoding example

```
module M = struct
   module X = (struct
      type t = int
      let x = 42
   end : sig
      type t
      val x : t
   end)
   type u = X.t * bool
end)
```

Signature

$$\exists\alpha.\left\{ \begin{array}{l} l_X : \left\{ \begin{array}{l} l_t : [\![= \alpha : \star]\!] \\ l_x : [\![\alpha]\!] \end{array} \right\} \\ l_u : [\![= \alpha \times \texttt{bool} : \star]\!] \end{array} \right\}$$

Module

$$l_X : \left\{ \begin{array}{l} l_t = [\![\texttt{int} : \star]\!] \\ l_x = [\![42]\!] \end{array} \right\}$$

Encoding example

```
module M = struct
   module X = (struct
      type t = int
      let x = 42
   end : sig
      type t
      val x : t
   end)
   type u = X.t * bool
end)
```

Signature

$$\exists\alpha.\left\{\begin{array}{l} l_X : \left\{\begin{array}{l} l_t : [\![= \alpha : \star]\!] \\ l_x : [\![\alpha]\!] \end{array}\right\} \\ l_u : [\![= \alpha \times \texttt{bool} : \star]\!] \end{array}\right\}$$

Module

$$l_X : \text{pack}\left\langle \texttt{int}, \left\{\begin{array}{l} l_t = [\![\texttt{int} : \star]\!] \\ l_x = [\![42]\!] \end{array}\right\}\right\rangle$$

Encoding example

```
module M = struct
   module X = (struct
      type t = int
      let x = 42
   end : sig
      type t
      val x : t
   end)
   type u = X.t * bool
end)
```

Signature

$$\exists\alpha.\left\{\begin{array}{l} l_X : \left\{\begin{array}{l} l_t : [\![= \alpha : \star]\!] \\ l_x : [\![\alpha]\!] \end{array}\right\} \\ l_u : [\![= \alpha \times \texttt{bool} : \star]\!] \end{array}\right\}$$

Module

$$\left\{\begin{array}{l} l_X : \mathsf{pack}\,\langle \texttt{int}, \{l_t = [\![\texttt{int} : \star]\!],\ l_x = [\![42]\!]\}\rangle \\ \\ \end{array}\right\}$$

Encoding example

```
module M = struct
   module X = (struct
      type t = int
      let x = 42
   end : sig
      type t
      val x : t
   end)
   type u = X.t * bool
end)
```

Signature

$$\exists \alpha. \left\{ \begin{array}{l} l_X : \left\{ \begin{array}{l} l_t : [\![= \alpha : \star]\!] \\ l_x : [\![\alpha]\!] \end{array} \right\} \\ l_u : [\![= \alpha \times \texttt{bool} : \star]\!] \end{array} \right\}$$

Module

$$\left\{ \begin{array}{l} l_X : \mathsf{pack}\, \langle \texttt{int}, \{ l_t = [\![\texttt{int} : \star]\!],\ l_x = [\![42]\!] \} \rangle \\ l_u : ? \times \texttt{bool} \end{array} \right\}$$

Encoding example

```
module M = struct
   module X = (struct
      type t = int
      let x = 42
   end : sig
      type t
      val x : t
   end)
   type u = X.t * bool
end)
```

Signature

$$\exists\alpha.\left\{ \begin{array}{l} l_X : \left\{\begin{array}{l} l_t : [\![= \alpha : \star]\!] \\ l_x : [\![\alpha]\!]\end{array}\right\} \\ l_u : [\![= \alpha \times \texttt{bool} : \star]\!]\end{array}\right\}$$

Module

$$\mathsf{unpack}\,\langle \alpha, y_1\rangle = \{ l_X : \mathsf{pack}\,\langle \texttt{int}, \{ l_t = [\![\texttt{int} : \star]\!],\ l_x = [\![42]\!]\}\rangle\}\ \ \mathsf{in}$$

Encoding example

```
module M = struct
   module X = (struct
      type t = int
      let x = 42
   end : sig
      type t
      val x : t
   end)
   type u = X.t * bool
end)
```

Signature

$$\exists\alpha.\left\{\begin{array}{l} l_X : \left\{\begin{array}{l} l_t : [\![= \alpha : \star]\!] \\ l_x : [\![\alpha]\!] \end{array}\right\} \\ l_u : [\![= \alpha \times \texttt{bool} : \star]\!] \end{array}\right\}$$

Module

$$\text{unpack}\,\langle\alpha, y_1\rangle = \{l_X : \text{pack}\,\langle\texttt{int}, \{l_t = [\![\texttt{int} : \star]\!],\ l_x = [\![42]\!]\}\rangle\}\ \text{in}$$

$$\text{unpack}\,\langle[], y_2\rangle = \{l_u :\ X.t \times \texttt{bool}\}\ \text{in}$$

Encoding example

```
module M = struct
   module X = (struct
      type t = int
      let x = 42
   end : sig
      type t
      val x : t
   end)
   type u = X.t * bool
end)
```

Signature

$$\exists\alpha.\left\{\begin{array}{l} l_X : \left\{\begin{array}{l} l_t : \llbracket = \alpha : \star \rrbracket \\ l_x : \llbracket \alpha \rrbracket \end{array}\right\} \\ l_u : \llbracket = \alpha \times \texttt{bool} : \star \rrbracket \end{array}\right\}$$

Module

$$\text{unpack}\,\langle \alpha, y_1 \rangle = \{ l_X : \text{pack}\,\langle \texttt{int}, \{ l_t = \llbracket \texttt{int} : \star \rrbracket,\; l_x = \llbracket 42 \rrbracket \} \rangle \} \;\text{ in}$$

$$\text{unpack}\,\langle [], y_2 \rangle = (\text{let}\; X.t, X.x = y_1.l_t, y_1.l_x \;\text{in}\; \{ l_u :\; X.t \times \texttt{bool} \}) \;\text{ in}$$

Encoding example

```
module M = struct
  module X = (struct
    type t = int
    let x = 42
  end : sig
    type t
    val x : t
  end)
  type u = X.t * bool
end)
```

Signature

$$\exists\alpha.\left\{\begin{array}{l} l_X : \left\{\begin{array}{l} l_t : [\![= \alpha : \star]\!] \\ l_x : [\![\alpha]\!] \end{array}\right\} \\ l_u : [\![= \alpha \times \mathtt{bool} : \star]\!] \end{array}\right\}$$

Module

$$\begin{array}{l} \text{unpack}\,\langle \alpha, y_1 \rangle = \{ l_X : \text{pack}\,\langle \mathtt{int}, \{ l_t = [\![\mathtt{int} : \star]\!],\ l_x = [\![42]\!] \} \rangle \}\ \text{in} \\ \text{unpack}\,\langle [], y_2 \rangle = (\text{let}\, X.t, X.x = y_1.l_t, y_1.l_x\ \text{in}\ \{ l_u :\ X.t \times \mathtt{bool} \})\ \text{in} \\ \text{pack}\,\langle \alpha, \{ l_X : y_1.l_X, l_u : y_2.l_u \} \rangle \end{array}$$

Encoding example

```
module M = struct
  module X = (struct
    type t = int
    let x = 42
  end : sig
    type t
    val x : t
  end)
  type u = X.t * bool
end)
```

Signature

$$\exists\alpha.\left\{\begin{array}{l} l_X : \left\{\begin{array}{l} l_t : [\![= \alpha : \star]\!] \\ l_x : [\![\alpha]\!] \end{array}\right\} \\ l_u : [\![= \alpha \times \texttt{bool} : \star]\!] \end{array}\right\}$$

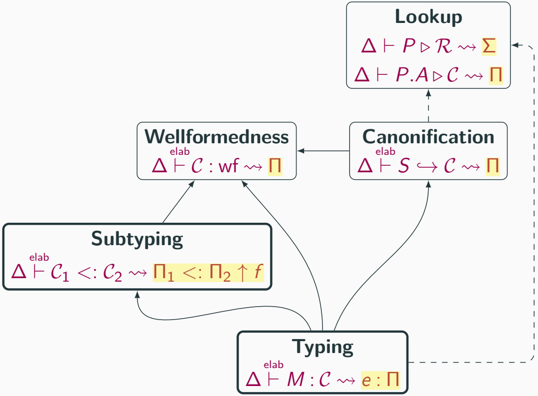
Module

$$\text{unpack}\,\langle \alpha, y_1 \rangle = \{l_X : \text{pack}\,\langle \texttt{int}, \{l_t = [\![\texttt{int} : \star]\!],\ l_x = [\![42]\!]\}\rangle\}\ \text{in}$$

$$\text{unpack}\,\langle [], y_2 \rangle = (\text{let}\, X.t, X.x = y_1.l_t, y_1.l_x\ \text{in}\ \{l_u :\ X.t \times \texttt{bool}\})\ \text{in}$$

$$\text{pack}\,\langle \alpha, \{l_X : y_1.l_X, l_u : y_2.l_u\}\rangle$$

Full elaboration of the judgments



Full elaboration of the judgments

E-Typ-Let

$$\frac{\Delta \overset{\text{elab}}{\vdash} E : T \rightsquigarrow e : \tau \qquad Y.x \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y (\text{let}\, x = E) : (\text{val}\, x : (\approx \tau)) \rightsquigarrow \{l_x = e\} : \{l_x : [\tau]\}}$$

E-Typ-Type

$$\frac{\Delta \overset{\text{elab}}{\vdash} T \hookrightarrow \tau \qquad Y.t \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y \text{type}\, t = T : \text{type}\, t \approx \tau \rightsquigarrow \{l_t = [\tau : \star]\} : \{l_t : [= \tau : \star]\}}$$

Full elaboration of the judgments

E-Typ-Let

$$\frac{\Delta \overset{\text{elab}}{\vdash} E : T \rightsquigarrow e : \tau \qquad Y.x \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y (\text{let}\, x = E) : (\text{val}\, x : (\approx \tau)) \rightsquigarrow \{l_x = e\} : \{l_x : [\tau]\}}$$

E-Typ-Type

$$\frac{\Delta \overset{\text{elab}}{\vdash} T \hookrightarrow \tau \qquad Y.t \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y \text{type}\, t = T : \text{type}\, t \approx \tau \rightsquigarrow \{l_t = [\tau : \star]\} : \{l_t : [= \tau : \star]\}}$$

E-Typ-Mod

$$\frac{\Delta \overset{\text{elab}}{\vdash} M : \exists \overline{\alpha}.\mathcal{R} \rightsquigarrow e : \exists \overline{\alpha}.\Sigma \qquad Y.X \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y (\text{module}\, X = M) : (\exists \overline{\alpha}.\, \text{module}\, X : \mathcal{R}) \rightsquigarrow \text{unpack}\, \langle \overline{\alpha}, x \rangle = e \text{ in pack}\, \langle \overline{\alpha}, \{l_X = x\} \rangle : \exists \overline{\alpha}.\, \{l_X : \Sigma\}}$$

Full elaboration of the judgments

E-Typ-Let

$$\frac{\Delta \overset{\text{elab}}{\vdash} E : T \rightsquigarrow e : \tau \qquad Y.x \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y (\text{let}\, x = E) : (\text{val}\, x : (\approx \tau)) \rightsquigarrow \{l_x = e\} : \{l_x : [\tau]\}}$$

E-Typ-Type

$$\frac{\Delta \overset{\text{elab}}{\vdash} T \hookrightarrow \tau \qquad Y.t \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y \text{type}\, t = T : \text{type}\, t \approx \tau \rightsquigarrow \{l_t = [\tau : \star]\} : \{l_t : [= \tau : \star]\}}$$

E-Typ-Mod

$$\frac{\Delta \overset{\text{elab}}{\vdash} M : \exists \overline{\alpha}.\mathcal{R} \rightsquigarrow e : \exists \overline{\alpha}.\Sigma \qquad Y.X \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y (\text{module}\, X = M) : (\exists \overline{\alpha}.\, \text{module}\, X : \mathcal{R}) \rightsquigarrow \text{unpack}\, \langle \overline{\alpha}, x \rangle = e \text{ in pack}\, \langle \overline{\alpha}, \{l_X = x\} \rangle : \exists \overline{\alpha}.\, \{l_X : \Sigma\}}$$

Correctness results

Theorem (Correctness of elaboration)

Given an environment Δ, a module expression M, a signature $\mathcal{C}$, a term e and a encoded signature Π, we have:

$$\Delta \overset{elab}{\vdash} M : \mathcal{C} \rightsquigarrow e : \Pi \implies \omega(\Delta) \overset{F_\omega}{\vdash} e : \Pi.$$ 3

Link with the canonical system

Theorem (Elaboration of typing)

Given an environment Γ, *a module expression* M, *and a signature* $\mathcal{C}$, *we have:*

$$\Gamma \overset{can}{\vdash} M : \mathcal{C} \implies \exists e,\ \Delta_\Gamma \overset{elab}{\vdash} M : \mathcal{C} \rightsquigarrow e : \Pi_{\mathcal{C}} \qquad 4$$

Link with the canonical system

Theorem (Elaboration of typing)
Given an environment Γ*, a module expression* M*, and a signature* $\mathcal{C}$*, we have:*

$$\Gamma \overset{can}{\vdash} M : \mathcal{C} \implies \exists e,\ \Delta_{\Gamma} \overset{elab}{\vdash} M : \mathcal{C} \rightsquigarrow e : \Pi_{\mathcal{C}} \qquad (4)$$

Theorem (Elaboration stripping)
Given an environment Δ*, a module expression* M*, a signature* $\mathcal{C}$*, a term* e *and a encoded signature* Π*, we have:*

$$\Delta \overset{elab}{\vdash} M : \mathcal{C} \rightsquigarrow e : \Pi \implies \mu(\Delta) \overset{can}{\vdash} M : \mathcal{C}. \qquad (5)$$

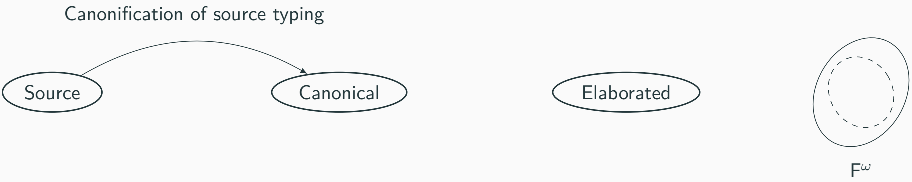
The big picture

The big picture

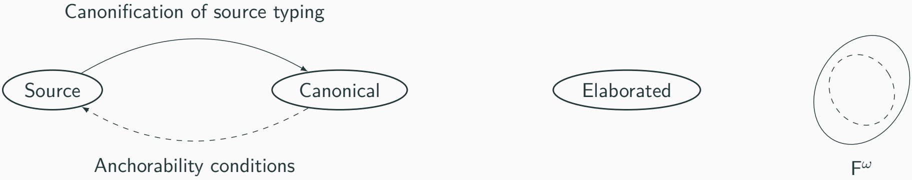
The big picture

The big picture

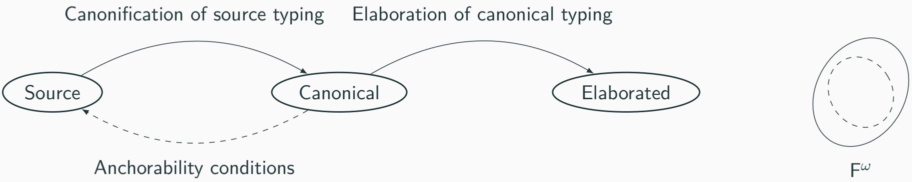
The big picture



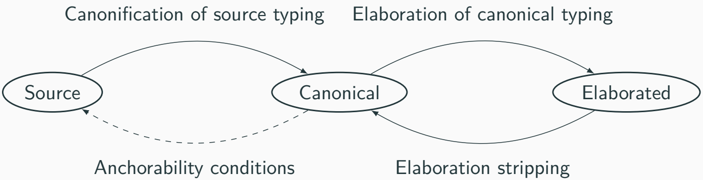
The big picture

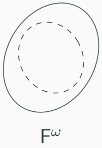


The big picture

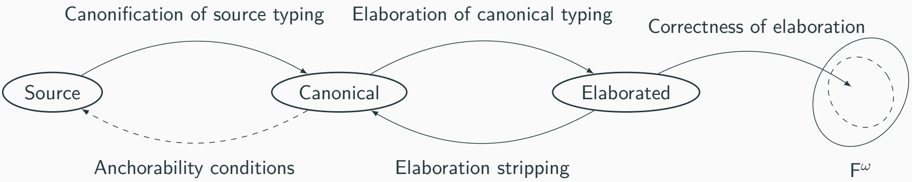


The big picture

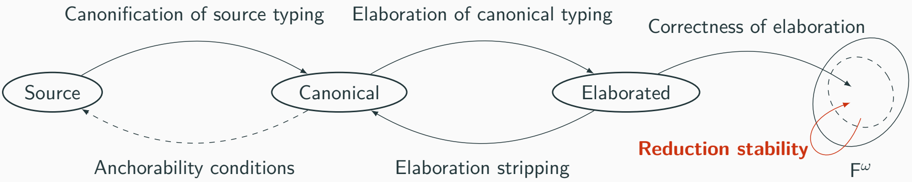




The big picture



The big picture



Conclusion and future work

Future work

Support a more significant subset of OCaml

Future work

Support a more significant subset of OCaml

- Extend the system to applicative functors

Future work

Support a more significant subset of OCaml

- Extend the system to applicative functors (higher kinds existentials)

Future work

Support a more significant subset of OCaml

- Extend the system to applicative functors (higher kinds existentials)
- Abstract signatures, 1^{st} Class modules, OCaml *private* types, etc.

Future work

Support a more significant subset of OCaml

- Extend the system to applicative functors (higher kinds existentials)
- Abstract signatures, 1^{st} Class modules, OCaml *private* types, etc.

Add new features

Future work

Support a more significant subset of OCaml

- Extend the system to applicative functors (higher kinds existentials)
- Abstract signatures, 1^{st} Class modules, OCaml *private* types, etc.

Add new features

- From type sharing to module sharing: module aliases

Future work

Support a more significant subset of OCaml

- Extend the system to applicative functors (higher kinds existentials)
- Abstract signatures, 1^{st} Class modules, OCaml *private* types, etc.

Add new features

- From type sharing to module sharing: module aliases
- Field removal without loss of type-sharing: transparent ascription

Future work

Support a more significant subset of OCaml

- Extend the system to applicative functors (higher kinds existentials)
- Abstract signatures, 1^{st} Class modules, OCaml *private* types, etc.

Add new features

- From type sharing to module sharing: module aliases
- Field removal without loss of type-sharing: transparent ascription

Mechanize the descriptions and proofs

Future work

Support a more significant subset of OCaml

- Extend the system to applicative functors (higher kinds existentials)
- Abstract signatures, 1^{st} Class modules, OCaml *private* types, etc.

Add new features

- From type sharing to module sharing: module aliases
- Field removal without loss of type-sharing: transparent ascription

Mechanize the descriptions and proofs

- Using a proof assistant (Coq)

Future work

Support a more significant subset of OCaml

- Extend the system to applicative functors (higher kinds existentials)
- Abstract signatures, 1^{st} Class modules, OCaml *private* types, etc.

Add new features

- From type sharing to module sharing: module aliases
- Field removal without loss of type-sharing: transparent ascription

Mechanize the descriptions and proofs

- Using a proof assistant (Coq)

Implement canonical-inspired algorithms in the typechecker

Future work

Support a more significant subset of OCaml

- Extend the system to applicative functors (higher kinds existentials)
- Abstract signatures, 1^{st} Class modules, OCaml *private* types, etc.

Add new features

- From type sharing to module sharing: module aliases
- Field removal without loss of type-sharing: transparent ascription

Mechanize the descriptions and proofs

- Using a proof assistant (Coq)

Implement canonical-inspired algorithms in the typechecker

- To solve *solvable* cases of signature avoidance

Future work

Support a more significant subset of OCaml

- Extend the system to applicative functors (higher kinds existentials)
- Abstract signatures, 1^{st} Class modules, OCaml *private* types, etc.

Add new features

- From type sharing to module sharing: module aliases
- Field removal without loss of type-sharing: transparent ascription

Mechanize the descriptions and proofs

- Using a proof assistant (Coq)

Implement canonical-inspired algorithms in the typechecker

- To solve *solvable* cases of signature avoidance
- To extend the signature syntax

Future work and challenges

- How a logical specification can serve the practical implementation ?

Future work and challenges

- How a logical specification can serve the practical implementation ?
- What techniques can be used to limit the size of the context ?

Future work and challenges

- How a logical specification can serve the practical implementation ?
- What techniques can be used to limit the size of the context ?
- How is the user experience with explicit existential quantification ?

Future work and challenges

- How a logical specification can serve the practical implementation ?
- What techniques can be used to limit the size of the context ?
- How is the user experience with explicit existential quantification ? (especially with applicative functors)

Future work and challenges

- How a logical specification can serve the practical implementation ?
- What techniques can be used to limit the size of the context ?
- How is the user experience with explicit existential quantification ? (especially with applicative functors)
- How about a mixed interface (existentials only appear in signature-avoidance cases) ?

Takeway

Takeway

1. An interesting presentation *spectrum* for OCaml modules, from the current path-based representation to the formal F^ω encoding, with the canonical system as a middle-point

Takeway

1. An interesting presentation *spectrum* for OCaml modules, from the current path-based representation to the formal F^ω encoding, with the canonical system as a middle-point
2. Intuitions and solutions for the signature avoidance problem:
   - Correct the current algorithm in current OCaml typechecker by improving equivalence

Takeway

1. An interesting presentation *spectrum* for OCaml modules, from the current path-based representation to the formal F^{ω} encoding, with the canonical system as a middle-point
2. Intuitions and solutions for the signature avoidance problem:
    - Correct the current algorithm in current OCaml typechecker by improving equivalence
    - Perspectives of extending the syntax to support all signature avoidance cases

Takeway

1. An interesting presentation *spectrum* for OCaml modules, from the current path-based representation to the formal F^ω encoding, with the canonical system as a middle-point
2. Intuitions and solutions for the signature avoidance problem:
   - Correct the current algorithm in current OCaml typechecker by improving equivalence
   - Perspectives of extending the syntax to support all signature avoidance cases
3. A clean framework for future extensions of the module system

Takeway

1. An interesting presentation *spectrum* for OCaml modules, from the current path-based representation to the formal F^ω encoding, with the canonical system as a middle-point
2. Intuitions and solutions for the signature avoidance problem:
   - Correct the current algorithm in current OCaml typechecker by improving equivalence
   - Perspectives of extending the syntax to support all signature avoidance cases
3. A clean framework for future extensions of the module system
4. Formal guarantees through elaboration into F^ω

References

D. DREYER. Recursive type generativity. *J. Funct. Program.*, 17(4–5):433–471, July 2007. ISSN 0956-7968. doi: 10.1017/S0956796807006429. URL https://doi.org/10.1017/S0956796807006429.

R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, page 341–354, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913434. doi: 10.1145/96709.96744. URL https://doi.org/10.1145/96709.96744.

References ii

X. Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95*, pages 142–153, San Francisco, California, United States, 1995. ACM Press. ISBN 978-0-89791-692-9. doi: 10.1145/199448.199476. URL http://portal.acm.org/citation.cfm?doid=199448.199476.

X. Leroy. A modular module system. *J. Funct. Program.*, 10(3):269–303, 2000. URL http://journals.cambridge.org/action/displayAbstract?aid=54525.

D. B. MacQueen. *Using Dependent Types to Express Modular Structure*, page 277–286. Association for Computing Machinery, New York, NY, USA, 1986. ISBN 9781450373470. URL https://doi.org/10.1145/512644.512670.

B. Montagu and D. Rémy. Modeling Abstract Types in Modules with Open Existential Types. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL'09)*, pages 354–365, Savannah, GA, USA, Jan. 2009. ISBN 978-1-60558-379-2. doi: http://doi.acm.org/10.1145/1480881.1480926.

References iii

A. Rossberg. 1ML - Core and modules united. *J. Funct. Program.*, 28:e22, 2018. doi: 10.1017/S0956796818000205. URL https://doi.org/10.1017/S0956796818000205.

A. Rossberg, C. Russo, and D. Dreyer. F-ing modules. *Journal of Functional Programming*, 24(5):529–607, Sept. 2014. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796814000264. URL https://www.cambridge.org/core/product/identifier/S0956796814000264/type/journal_article.

C. V. Russo. Types for modules. *Electronic Notes in Theoretical Computer Science*, 60:3–421, 2004. ISSN 1571-0661. doi: https://doi.org/10.1016/S1571-0661(05)82621-0. URL https://www.sciencedirect.com/science/article/pii/S1571066105826210.