Retrofitting OCaml modules

An F^ω-inspired approach for a modern module system

Clément Blaudeau, Cambium, Inria

February 9, 2023

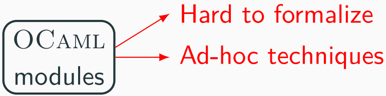
The big picture

OCaml
modules

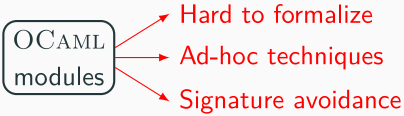
The big picture

OCaml modules → Hard to formalize

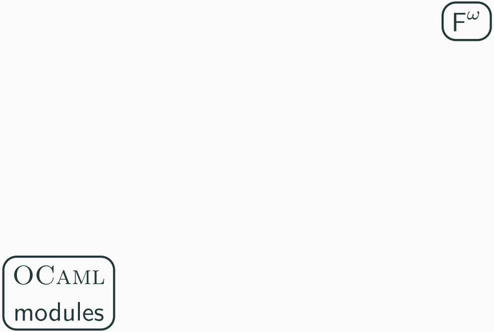
The big picture



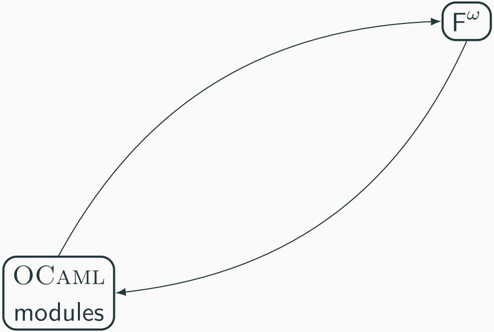
The big picture



The big picture



The big picture



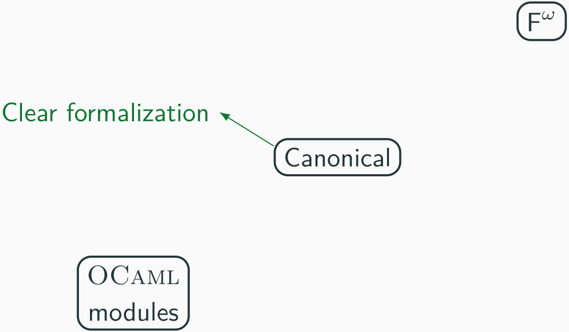
The big picture

F^ω

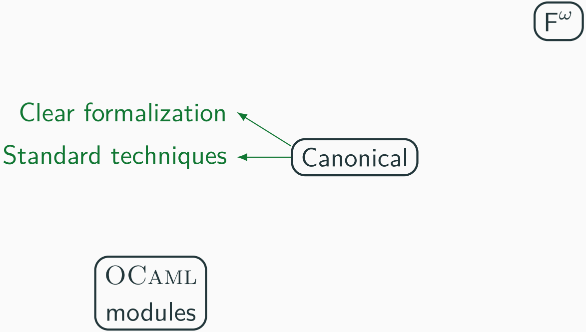
Canonical

OCaml modules

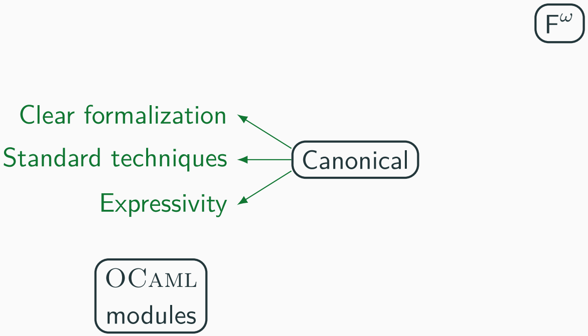
The big picture



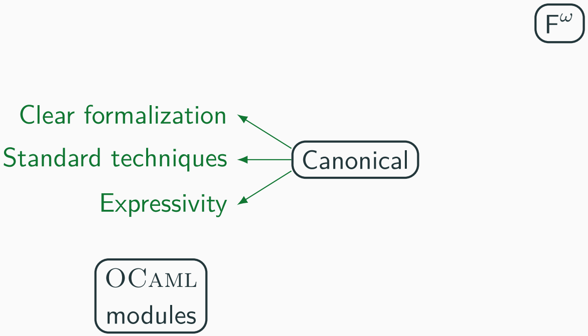
The big picture



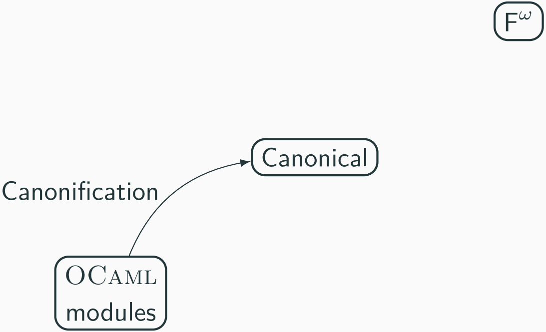
The big picture



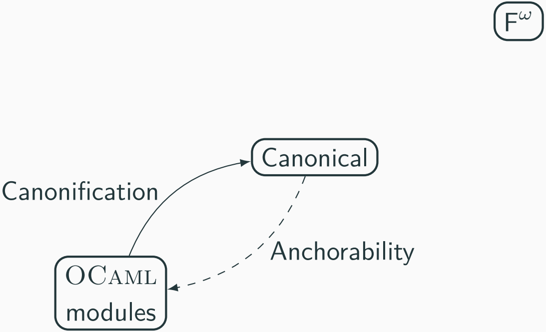
The big picture



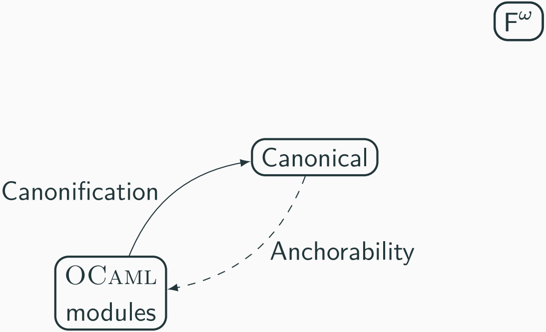
The big picture



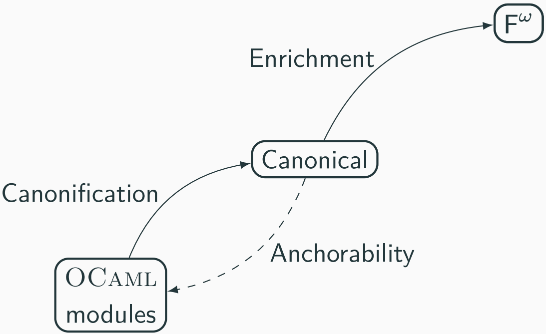
The big picture



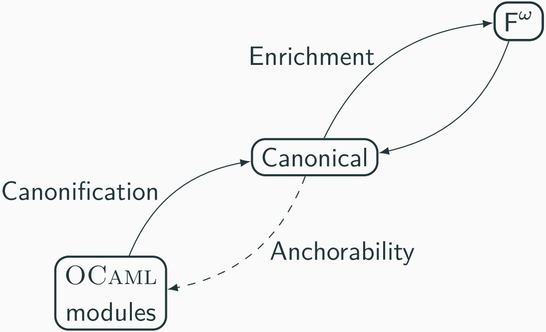
The big picture



The big picture



The big picture



The OCaml Module system

As a module developer

As a module user

As a module developer

```
module Complex  = struct






end
```

As a module user

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex  = struct
  type t = float * float




end
```

As a module user

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex  = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end
```

As a module user

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex  = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig





end
```

As a module user

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex  = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t




end
```

As a module user

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex  = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
end
```

As a module user

As a module developer

```
module Complex : Ring = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
end
```

As a module user

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex : Ring = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
end
```

As a module user

```
module Polynomials =






end
```

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex : Ring = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
end
```

As a module user

```
module Polynomials =
  functor (R: Ring) -> struct






end
```

```





```

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex : Ring = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
end
```

As a module user

```
module Polynomials =
  functor (R: Ring) -> struct
    type t = R.t list
    let zero = []
    let one = [R.one]
    let add = ...
    let mult = ...
end
```

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex : Ring = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
end
```

As a module user

```
module Polynomials =
  functor (R: Ring) -> struct
    type t = R.t list
    let zero = []
    let one = [R.one]
    let add = ...
    let mult = ...
end
```

```
module CX =
```

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex : Ring = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
end
```

As a module user

```
module Polynomials =
  functor (R: Ring) -> struct
    type t = R.t list
    let zero = []
    let one = [R.one]
    let add = ...
    let mult = ...
end
```

```
module CX =
  Polynomials(Complex)
```

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex : Ring = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
end
```

As a module user

```
module Polynomials =
  functor (R: Ring) -> struct
    type t = R.t list
    let zero = []
    let one = [R.one]
    let add = ...
    let mult = ...
end
```

```
module CX =
  Polynomials(Complex)

module CXY =

```

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex : Ring = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
end
```

As a module user

```
module Polynomials =
  functor (R: Ring) -> struct
    type t = R.t list
    let zero = []
    let one = [R.one]
    let add = ...
    let mult = ...
end
```

```
module CX =
  Polynomials(Complex)

module CXY =
  Polynomials(Polynomials(Complex))
```

What flavor for you functor ?

What flavor for you functor ?

Generative

What flavor for you functor ?

Generative

Applicative

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

Applicative

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

- Internal state / effects

Applicative

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- Dynamic choice of implementation
  (via 1^{st} class modules)

Applicative

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- Dynamic choice of implementation (via 1^{st} class modules)

```
module SymbolTable () =
```

Applicative

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- Dynamic choice of implementation (via 1^{st} class modules)

```
module SymbolTable () = struct
  type t = int
  let x = ref 0
  ...
end
```

Applicative

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- Dynamic choice of implementation (via 1^{st} class modules)

```
module SymbolTable () = (struct
  type t = int
  let x = ref 0
  ...
end : sig type t ... end)
```

Applicative

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
module SymbolTable () = (struct
  type t = int
  let x = ref 0
  ...
end : sig type t ... end)

module ST1 = SymboleTable()
module ST2 = SymboleTable()
```

Applicative

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
module SymbolTable () = (struct
  type t = int
  let x = ref 0
  ...
end : sig type t ... end)

module ST1 = SymboleTable()
module ST2 = SymboleTable()
(* ST1.t ≠ ST2.t *) ×
```

Applicative

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
module SymbolTable () = (struct
  type t = int
  let x = ref 0
  ...
end : sig type t ... end)

module ST1 = SymboleTable()
module ST2 = SymboleTable()
(* ST1.t ≠ ST2.t *) ×
```

Applicative

Functors as parameterized *libraries*

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
module SymbolTable () = (struct
  type t = int
  let x = ref 0
  ...
end : sig type t ... end)

module ST1 = SymboleTable()
module ST2 = SymboleTable()
(* ST1.t ≠ ST2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
module SymbolTable () = (struct
  type t = int
  let x = ref 0
  ...
end : sig type t ... end)

module ST1 = SymboleTable()
module ST2 = SymboleTable()
(* ST1.t ≠ ST2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity
- Static choice of implementation

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
module SymbolTable () = (struct
  type t = int
  let x = ref 0
  ...
end : sig type t ... end)

module ST1 = SymboleTable()
module ST2 = SymboleTable()
(* ST1.t ≠ ST2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity
- Static choice of implementation

```
module Set (E:OrderedType) = struct
```

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
module SymbolTable () = (struct
  type t = int
  let x = ref 0
  ...
end : sig type t ... end)

module ST1 = SymboleTable()
module ST2 = SymboleTable()
(* ST1.t ≠ ST2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity
- Static choice of implementation

```
module Set (E:OrderedType) = struct
  type t = E.t list
  let empty : t = []
  ...
end
```

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
module SymbolTable () = (struct
  type t = int
  let x = ref 0
  ...
end : sig type t ... end)

module ST1 = SymboleTable()
module ST2 = SymboleTable()
(* ST1.t ≠ ST2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity
- Static choice of implementation

```
module Set (E:OrderedType) = (struct
  type t = E.t list
  let empty : t = []
  ...
end : sig type t ... end)
```

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- Dynamic choice of implementation (via 1^{st} class modules)

```
module SymbolTable () = (struct
  type t = int
  let x = ref 0
  ...
end : sig type t ... end)

module ST1 = SymboleTable()
module ST2 = SymboleTable()
(* ST1.t ≠ ST2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity
- Static choice of implementation

```
module Set (E:OrderedType) = (struct
  type t = E.t list
  let empty : t = []
  ...
end : sig type t ... end)

module S1 = Set(Integer)
module S2 = Set(Integer)
```

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
module SymbolTable () = (struct
  type t = int
  let x = ref 0
  ...
end : sig type t ... end)

module ST1 = SymboleTable()
module ST2 = SymboleTable()
(* ST1.t ≠ ST2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity
- Static choice of implementation

```
module Set (E:OrderedType) = (struct
  type t = E.t list
  let empty : t = []
  ...
end : sig type t ... end)

module S1 = Set(Integer)
module S2 = Set(Integer)
(* S1.t = S2.t *) ✓
```

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
module SymbolTable () = (struct
  type t = int
  let x = ref 0
  ...
end : sig type t ... end)

module ST1 = SymboleTable()
module ST2 = SymboleTable()
(* ST1.t ≠ ST2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity
- Static choice of implementation

→ *same applications* produce same results

```
module Set (E:OrderedType) = (struct
  type t = E.t list
  let empty : t = []
  ...
end : sig type t ... end)

module S1 = Set(Integer)
module S2 = Set(Integer)
(* S1.t = S2.t *) ✓
```

What flavor for you functor ?

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- Dynamic choice of implementation (via 1^{st} class modules)

```
module SymbolTable () = (struct
  type t = int
  let x = ref 0
  ...
end : sig type t ... end)

module ST1 = SymboleTable()
module ST2 = SymboleTable()
(* ST1.t ≠ ST2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity
- Static choice of implementation

→ *same applications* produce same results

```
module Set (E:OrderedType) = (struct
  type t = E.t list
  let empty : t = []
  ...
end : sig type t ... end)

module S1 = Set(Integer)
module S2 = Set(Integer)
(* S1.t = S2.t *) ✓
```

Applicativity granularity

```
module X1 = struct
  type t = int
  ...
end

module X2 = struct
  type t = int
  ...
end

Set(X1).t =? Set(X2).t
```

Applicativity granularity

Types only

```
module X1 = struct
  type t = int
  ...
end

module X2 = struct
  type t = int
  ...
end

Set(X1).t =? Set(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types

```
module X1 = struct
  type t = int
  ...
end

module X2 = struct
  type t = int
  ...
end

Set(X1).t =? Set(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

```
module X1 = struct
  type t = int
  ...
end

module X2 = struct
  type t = int
  ...
end

Set(X1).t =? Set(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

```
module X1 = struct
  type t = int
  let compare = (<)
end

module X2 = struct
  type t = int
  let compare = (>)
end

Set(X1).t =? Set(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

Types and values

```
module X1 = struct
  type t = int
  let compare = (<)
end

module X2 = struct
  type t = int
  let compare = (>)
end

Set(X1).t =? Set(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

Types and values

- *Abstraction safe*: dynamically equivalent

```
module X1 = struct
  type t = int
  let compare = (<)
end

module X2 = struct
  type t = int
  let compare = (>)
end

Set(X1).t =? Set(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

Types and values

- *Abstraction safe*: dynamically equivalent
  $\rightarrow$ tracking equality of values

```
module X1 = struct
  type t = int
  let compare = (<)
end

module X2 = struct
  type t = int
  let compare = (>)
end

Set(X1).t =? Set(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  → assumes the functor's body only depends on types fields

Types and values

- *Abstraction safe*: dynamically equivalent
  → tracking equality of values

```
module X1 = struct
  type t = int
  let compare = (<)
end

module X2 = struct
  type t = int
  let compare = X1.compare
end

Set(X1).t =? Set(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

Types and values - *fine grained*

- *Abstraction safe*: dynamically equivalent
  $\rightarrow$ tracking equality of values

```
module X1 = struct
  type t = int
  let compare = (<)
end

module X2 = struct
  type t = int
  let compare = X1.compare
end

Set(X1).t =? Set(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  → assumes the functor's body only depends on types fields

Types and values - *fine grained*

- *Abstraction safe*: dynamically equivalent
  → tracking equality of values

Modules - *coarse grained*

```
module X1 = struct
  type t = int
  let compare = (<)
end

module X2 = struct
  type t = int
  let compare = X1.compare
end

Set(X1).t =? Set(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

Types and values - *fine grained*

- *Abstraction safe*: dynamically equivalent
  $\rightarrow$ tracking equality of values

Modules - *coarse grained*

- Syntactic criterion

```
module X1 = struct
  type t = int
  let compare = (<)
end

module X2 = struct
  type t = int
  let compare = X1.compare
end

Set(X1).t =? Set(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  → assumes the functor's body only depends on types fields

Types and values - *fine grained*

- *Abstraction safe*: dynamically equivalent
  → tracking equality of values

Modules - *coarse grained*

- Syntactic criterion

```
module X1 = struct
  type t = int
  let compare = (<)
end







Set(X1).t =? Set(X1).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  → assumes the functor's body only depends on types fields

Types and values - *fine grained*

- *Abstraction safe*: dynamically equivalent
  → tracking equality of values

Modules - *coarse grained*

- Syntactic criterion
  → tracking of module *aliasing*

```
module X1 = struct
  type t = int
  let compare = (<)
end






Set(X1).t =? Set(X1).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

Types and values - *fine grained*

- *Abstraction safe*: dynamically equivalent
  $\rightarrow$ tracking equality of values

Modules - *coarse grained*

- Syntactic criterion
  $\rightarrow$ tracking of module *aliasing*

```
module X1 = struct
  type t = int
  let compare = (<)
end

module X2 = X1




Set(X1).t =? Set(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  → assumes the functor's body only depends on types fields

Types and values - *fine grained*

- *Abstraction safe*: dynamically equivalent
  → tracking equality of values

Modules - *coarse grained*

- Syntactic criterion
  → tracking of module *aliasing*

```
module X1 = struct
  type t = int
  let compare = (<)
end

module X2 = X1




Set(X1).t =? Set(X2).t
```

Aliases and transparent ascription

Modules

Signatures

Aliases and transparent ascription

Modules

```
module X1 = struct ... end
```

Signatures

Aliases and transparent ascription

Modules

```
module X1 = struct ... end
```

Signatures

```
module X1 : sig ... end
```

Aliases and transparent ascription

- Same module

Modules

```
module X1 = struct ... end
```

Signatures

```
module X1 : sig ... end
```

Aliases and transparent ascription

- Same module

Modules

```
module X1 = struct ... end

module X2 = X1
```

Signatures

```
module X1 : sig ... end
```

Aliases and transparent ascription

- Same module

Modules

```
module X1 = struct ... end

module X2 = X1
```

Signatures

```
module X1 : sig ... end

module X2 : (= X1)
```

Aliases and transparent ascription

- Same module
- Subtyping (not code-free)

Modules

```
module X1 = struct ... end

module X2 = X1
```

Signatures

```
module X1 : sig ... end

module X2 : (= X1)
```

Aliases and transparent ascription

- Same module
- Subtyping (not code-free)

Modules

```
module X1 = struct ... end

module X2 = X1

module X3 = (X1 : S)
```

Signatures

```
module X1 : sig ... end

module X2 : (= X1)
```

Aliases and transparent ascription

- Same module
- Subtyping (not code-free)

Modules

```
module X1 = struct ... end

module X2 = X1

module X3 = (X1 : S)

module X4 = (X1 < S)
```

Signatures

```
module X1 : sig ... end

module X2 : (= X1)
```

Aliases and transparent ascription

- Same module
- Subtyping (not code-free)

Modules

```
module X1 = struct ... end

module X2 = X1

module X3 = (X1 : S)

module X4 = (X1 < S)
```

Signatures

```
module X1 : sig ... end

module X2 : (= X1)

module X3 : S
```

Aliases and transparent ascription

- Same module
- Subtyping (not code-free)

Modules

```
module X1 = struct ... end

module X2 = X1

module X3 = (X1 : S)

module X4 = (X1 < S)


```

Signatures

```
module X1 : sig ... end

module X2 : (= X1)

module X3 : S

module X4 : (= X1 < S)


```

Aliases and transparent ascription

- Same module
- Subtyping (not code-free) $\rightarrow$ transparent/opaque

Modules

```
module X1 = struct ... end

module X2 = X1

module X3 = (X1 : S)

module X4 = (X1 < S)
```

Signatures

```
module X1 : sig ... end

module X2 : (= X1)

module X3 : S

module X4 : (= X1 < S)
```

Aliases and transparent ascription

- Same module
- Subtyping (not code-free) $\rightarrow$ transparent/opaque
- Functor applications

Modules

```
module X1 = struct ... end

module X2 = X1

module X3 = (X1 : S)

module X4 = (X1 < S)


```

Signatures

```
module X1 : sig ... end

module X2 : (= X1)

module X3 : S

module X4 : (= X1 < S)


```

Aliases and transparent ascription

- Same module
- Subtyping (not code-free) $\rightarrow$ transparent/opaque
- Functor applications

Modules

```
module X1 = struct ... end

module X2 = X1

module X3 = (X1 : S)

module X4 = (X1 < S)

module X5 = (functor (Y:S) -> Y)(X1)
```

Signatures

```
module X1 : sig ... end

module X2 : (= X1)

module X3 : S

module X4 : (= X1 < S)
```

Aliases and transparent ascription

- Same module
- Subtyping (not code-free) $\rightarrow$ transparent/opaque
- Functor applications

Modules

```
module X1 = struct ... end

module X2 = X1

module X3 = (X1 : S)

module X4 = (X1 < S)

module X5 = (functor (Y:S) -> Y)(X1)
```

Signatures

```
module X1 : sig ... end

module X2 : (= X1)

module X3 : S

module X4 : (= X1 < S)

module X5 : (= X1 < S)
```

The canonical system

Canonical signatures - canonification

Enriched syntax

→ F^{ω} quantifiers

Key mechanisms

module M

- module X
  - type t
  - val $x : t$
  - type $u = \text{int} \times t$
- module F (Y : sig type t end)
  - type t
  - type $u = Y.t$
- module G ()
  - type t
  - val $x : t$
- module type T
  - type t
  - val $x : t$

Canonical signatures - canonification

Enriched syntax

→ F^ω quantifiers

Key mechanisms

module M
- module X
  - type t
  - val x : t
  - type u = int × t
- module F (Y : sig type t end)
  - type t
  - type u = $Y.t$
- module G ()
  - type t
  - val x : t
- module type T
  - type t
  - val x : t

Canonical signatures - canonification

Enriched syntax

$\rightarrow$ F^{ω} quantifiers

- Existential for ascription

Key mechanisms

module M

- module X
  - $\exists\alpha$
    - type $t = \alpha$
    - val $x : t$
    - type $u = \text{int} \times t$
- module F (Y : sig type t end)
  - type t
  - type $u = Y.t$
- module G ()
  - type t
  - val $x : t$
- module type T
  - type t
  - val $x : t$

Canonical signatures - canonification

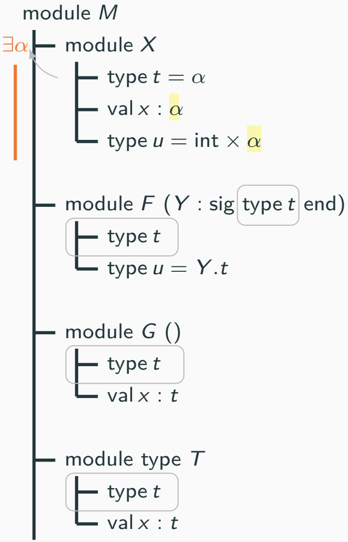
Enriched syntax

→ F^ω quantifiers

- Existential for ascription

Key mechanisms

- Existential lifting



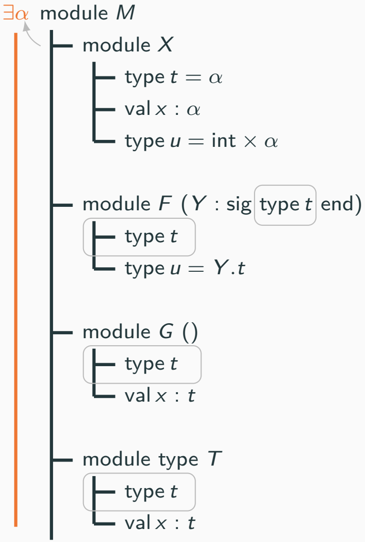
Enriched syntax

→ F^ω quantifiers

- Existential for ascription
- Universal for functors

Key mechanisms

- Existential lifting



Canonical signatures - canonification

Enriched syntax

$\rightarrow$ F^{ω} quantifiers

- Existential for ascription
- Universal for functors

Key mechanisms

- Existential lifting

$\exists\alpha$ module M

- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \text{int} \times \alpha$
- module $F : \forall\beta.(Y : \text{sig type } t = \beta \text{ end})$
  - type t
  - type $u = \beta$
- module G ()
  - type t
  - val $x : t$
- module type T
  - type t
  - val $x : t$

Canonical signatures - canonification

Enriched syntax

$\rightarrow$ F^{ω} quantifiers

- Existential for ascription
- Universal for functors

Key mechanisms

- Existential lifting

$\exists\alpha$ module M

- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \text{int} \times \alpha$
- module $F : \forall\beta.(Y : \text{sig type } t = \beta \text{ end})$
  - $\exists\gamma$ type $t = \gamma$
  - type $u = \beta$
- module G ()
  - type t
  - val $x : t$
- module type T
  - type t
  - val $x : t$

Canonical signatures - canonification

Enriched syntax

$\rightarrow$ F^{ω} quantifiers

- Existential for ascription
- Universal for functors

Key mechanisms

- Existential lifting
- Skolemization

```
∃α  module M
├─ module X
│  ├─ type t = α
│  ├─ val x : α
│  └─ type u = int × α
│
∃γ' ├─ module F : ∀β.(Y : sig type t = β end)
│  ├─ type t = γ'(β)
│  └─ type u = β
│
├─ module G ()
│  ├─ type t
│  └─ val x : t
│
└─ module type T
   ├─ type t
   └─ val x : t
```

Canonical signatures - canonification

Enriched syntax

$\rightarrow$ F^ω quantifiers

- Existential for ascription
- Universal for functors

Key mechanisms

- Existential lifting
- Skolemization

$\exists \alpha, \gamma'$ module M

- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \text{int} \times \alpha$
- module $F : \forall \beta.(Y : \text{sig type } t = \beta \text{ end})$
  - type $t = \gamma'(\beta)$
  - type $u = \beta$
- module G ()
  - type t
  - val $x : t$
- module type T
  - type t
  - val $x : t$

Canonical signatures - canonification

Enriched syntax

$\rightarrow$ F^{ω} quantifiers

- Existential for ascription
- Universal for functors

Key mechanisms

- Existential lifting
- Skolemization

$\exists \alpha, \gamma'$ module M

- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \text{int} \times \alpha$
- module $F : \forall \beta.(Y : \text{sig type } t = \beta \text{ end})$
  - type $t = \gamma'(\beta)$
  - type $u = \beta$
- module G ()
  - $\exists \delta$ type $t = \delta$
  - val $x : t$
- module type T
  - type t
  - val $x : t$

Canonical signatures - canonification

Enriched syntax

→ F^{ω} quantifiers

- Existential for ascription
- Universal for functors

Key mechanisms

- Existential lifting
- Skolemization

$\exists \alpha, \gamma'$ module M

- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \text{int} \times \alpha$
- module $F : \forall \beta.(Y : \text{sig type } t = \beta \text{ end})$
  - type $t = \gamma'(\beta)$
  - type $u = \beta$
- module G () : $\exists \delta$
  - type $t = \delta$
  - val $x : \delta$
- module type T
  - type t
  - val $x : t$

Canonical signatures - canonification

Enriched syntax

$\rightarrow$ F^ω quantifiers

- Existential for ascription
- Universal for functors
- Lambda for module types

Key mechanisms

- Existential lifting
- Skolemization

$\exists \alpha, \gamma'$ module M

- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \text{int} \times \alpha$
- module $F : \forall \beta.(Y : \text{sig type } t = \beta \text{ end})$
  - type $t = \gamma'(\beta)$
  - type $u = \beta$
- module $G\ () : \exists \delta$
  - type $t = \delta$
  - val $x : \delta$
- module type T
  - $\lambda \rho$
    - type $t = \rho$
    - val $x : t$

Canonical signatures - canonification

Enriched syntax

$\rightarrow$ F^{ω} quantifiers

- Existential for ascription
- Universal for functors
- Lambda for module types

Key mechanisms

- Existential lifting
- Skolemization

$\exists \alpha, \gamma'$ module M

- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \text{int} \times \alpha$
- module $F : \forall \beta.(Y : \text{sig type } t = \beta \text{ end})$
  - type $t = \gamma'(\beta)$
  - type $u = \beta$
- module $G\ () : \exists \delta$
  - type $t = \delta$
  - val $x : \delta$
- module type $T : \lambda \rho$
  - type $t = \rho$
  - val $x : \rho$

Module identities

module M

- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \text{int} \times \alpha$
- module $X' : (= X < S)$

Module identities

- Use type abstraction mechanisms to track module identities

module M
- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \text{int} \times \alpha$
- module $X' : (= X < S)$

Module identities

- Use type abstraction mechanisms to track module identities
- Add abstract id fields, with a special treatment by typing rules

module M

- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \text{int} \times \alpha$
- module $X' : (= X < S)$

Module identities

- Use type abstraction mechanisms to track module identities
- Add abstract `id` fields, with a special treatment by typing rules

$\exists \alpha_X$ module M

- module $X = \alpha_X$
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \text{int} \times \alpha$
- module $X' : (= X < S)$

Module identities

- Use type abstraction mechanisms to track module identities
- Add abstract `id` fields, with a special treatment by typing rules

$\exists\alpha_X$ module M

- module $X = \alpha_X$
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \text{int} \times \alpha$
- module $X' = \alpha_X$

Module identities

- Use type abstraction mechanisms to track module identities
- Add abstract `id` fields, with a special treatment by typing rules

$\exists \alpha_X$ module M

- module $X = \alpha_X$
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \text{int} \times \alpha$
- module $X' = \alpha_X$
  - type $t = \alpha$
  - val $x : \alpha$
  - ~~type $u = \text{int} \times \alpha$~~

Module identities

- Use type abstraction mechanisms to track module identities
- Add abstract `id` fields, with a special treatment by typing rules

$\exists \alpha_X$ module M

- module $X = \alpha_X$
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \text{int} \times \alpha$
- module $X' = \alpha_X$
  - type $t = \alpha$
  - val $x : \alpha$
  - ~~type $u = \text{int} \times \alpha$~~

Canonical grammar

Invariants

Canonical grammar

Invariants

- Quantifier positions ($\exists$,$\forall$,λ)

Canonical grammar

Invariants

- Quantifier positions ($\exists$,$\forall$,λ)
- Identities
  $\rightarrow$ when *reachable by a path*

Canonical grammar

Abstract signatures

$\mathcal{S} ::= \exists\overline{\alpha}.\mathcal{C}$

Invariants

- Quantifier positions ($\exists$,$\forall$,λ)
- Identities
  $\rightarrow$ when *reachable by a path*

Canonical grammar

Abstract signatures

$$\mathcal{S} ::= \exists\overline{\alpha}.\mathcal{C}$$

Identity signatures

$$\mathcal{C} ::= (\tau, \mathcal{R})$$

Invariants

- Quantifier positions ($\exists$,$\forall$,λ)
- Identities
  $\rightarrow$ when *reachable by a path*

Canonical grammar

Abstract signatures

$$\mathcal{S} ::= \exists\overline{\alpha}.\mathcal{C}$$

Identity signatures

$$\mathcal{C} ::= (\tau, \mathcal{R})$$

Invariants

- Quantifier positions ($\exists$,$\forall$,λ)
- Identities
  $\rightarrow$ when *reachable by a path*

Types

$$\tau ::= \alpha \mid \alpha(\overline{\tau})$$

Canonical grammar

Invariants

- Quantifier positions ($\exists$,$\forall$,λ)
- Identities
  $\rightarrow$ when *reachable by a path*

Abstract signatures

$$\mathcal{S} ::= \exists\overline{\alpha}.\mathcal{C}$$

Identity signatures

$$\mathcal{C} ::= (\tau, \mathcal{R})$$

Manifest signatures

$$\begin{aligned} \mathcal{R} ::= & \ \texttt{sig}\ \overline{\mathcal{D}}\ \texttt{end} && \textit{Structural signature} \\ | & \ \forall\overline{\alpha}.\,\mathcal{C} \rightarrow \mathcal{R} && \textit{Applicative functor} \\ | & \ () \rightarrow \exists\overline{\alpha}.\mathcal{R} \end{aligned}$$

Types

$$\begin{aligned} \tau ::= & \ \alpha \\ | & \ \alpha(\overline{\tau}) \end{aligned}$$

Canonical grammar

Invariants

- Quantifier positions ($\exists$,$\forall$,λ)
- Identities
  $\rightarrow$ when *reachable by a path*

Abstract signatures

$$\mathcal{S} ::= \exists\overline{\alpha}.\mathcal{C}$$

Identity signatures

$$\mathcal{C} ::= (\tau, \mathcal{R})$$

Manifest signatures

$\mathcal{R} ::= \texttt{sig}\ \overline{\mathcal{D}}\ \texttt{end}$	*Structural signature*
$\mid \forall\overline{\alpha}.\,\mathcal{C} \rightarrow \mathcal{R}$	*Applicative functor*
$\mid () \rightarrow \exists\overline{\alpha}.\mathcal{R}$	*Generative functor*

Declarations

$$\begin{aligned}\mathcal{D} ::=\ & \mathsf{val}\, x : \tau \\ \mid\ & \mathsf{type}\, t = \tau \\ \mid\ & \mathsf{module}\, X : \mathcal{C} \\ \mid\ & \mathsf{module\ type}\ \ T = \lambda\overline{\alpha}.\mathcal{C}\end{aligned}$$

Types

$$\begin{aligned}\tau ::=\ & \alpha \\ \mid\ & \alpha(\overline{\tau})\end{aligned}$$

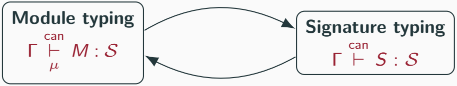
The canonical system

The canonical system

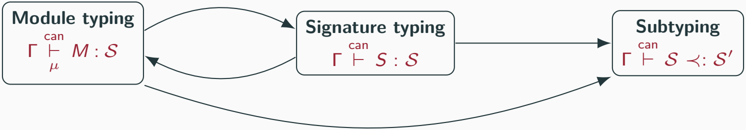
Signature typing

$\Gamma \overset{\text{can}}{\vdash} S : \mathcal{S}$

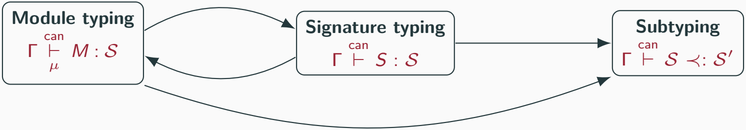
The canonical system



The canonical system



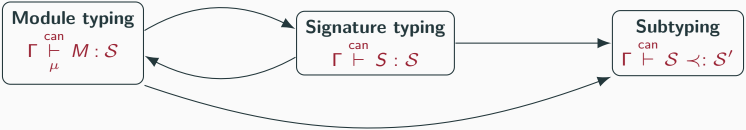
The canonical system



C-Typ-Mod-Struct

$$\frac{}{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} \texttt{struct}\ \overline{B}\ \texttt{end} :}$$

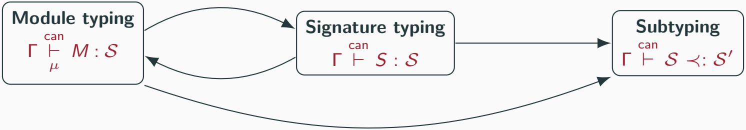
The canonical system



C-Typ-Mod-Struct

$$\frac{\Gamma \underset{\mu}{\overset{\text{can}}{\vdash}} \overline{B} : \exists \overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \underset{\mu}{\overset{\text{can}}{\vdash}} \texttt{struct}\ \overline{B}\ \texttt{end} :}$$

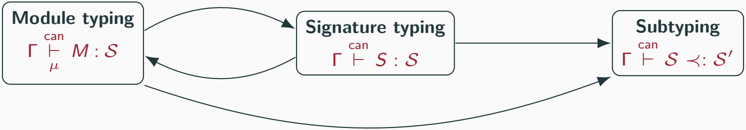
The canonical system



C-Typ-Mod-Struct

$$\frac{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} \overline{B} : \exists \overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} \texttt{struct}\ \overline{B}\ \texttt{end} : \exists \overline{\alpha}\quad .\left(\quad , \texttt{sig}\ \overline{\mathcal{D}}\ \texttt{end}\right)}$$

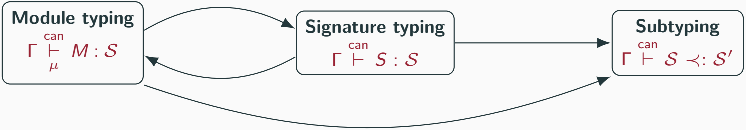
The canonical system



C-Typ-Mod-Struct

$$\frac{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} \overline{B} : \exists \overline{\alpha}. \overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} \texttt{struct}\ \overline{B}\ \texttt{end} : \exists \overline{\alpha}, \alpha. \left(\quad , \texttt{sig}\ \overline{\mathcal{D}}\ \texttt{end} \right)}$$

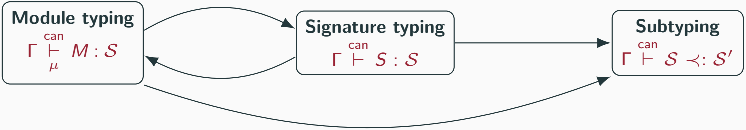
The canonical system



C-Typ-Mod-Struct

$$\frac{\Gamma \underset{\mu}{\overset{\text{can}}{\vdash}} \overline{B} : \exists \overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \underset{\mu}{\overset{\text{can}}{\vdash}} \texttt{struct}\ \overline{B}\ \texttt{end} : \exists \overline{\alpha}, \alpha.\ \left(\alpha, \texttt{sig}\ \overline{\mathcal{D}}\ \texttt{end}\right)}$$

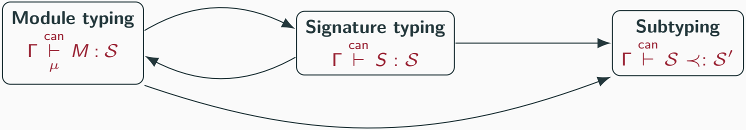
The canonical system



C-Typ-Mod-AppGen

$$\frac{}{\Gamma \overset{\text{can}}{\underset{\text{gen}}{\vdash}} P() :}$$

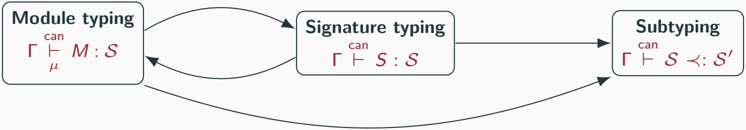
The canonical system



C-Typ-Mod-AppGen

$$\frac{\Gamma \underset{\mu}{\overset{\text{can}}{\vdash}} P : (\tau, () \to \exists \overline{\alpha}.\mathcal{R})}{\Gamma \underset{\text{gen}}{\overset{\text{can}}{\vdash}} P() :}$$

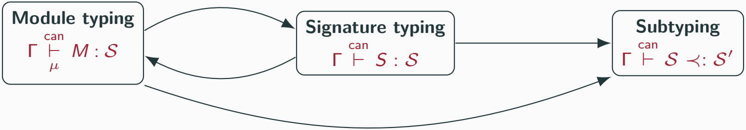
The canonical system



C-Typ-Mod-AppGen

$$\frac{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} P : (\tau, () \to \exists\overline{\alpha}.\mathcal{R})}{\Gamma \overset{\text{can}}{\underset{\text{gen}}{\vdash}} P() : \exists\overline{\alpha} \quad .(\quad , \mathcal{R})}$$

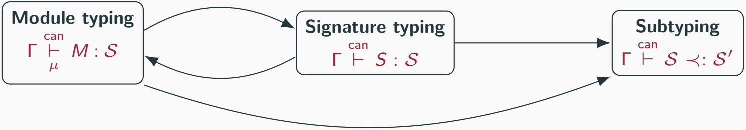
The canonical system



C-Typ-Mod-AppGen

$$\frac{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} P : (\tau, () \rightarrow \exists \overline{\alpha}.\mathcal{R})}{\Gamma \overset{\text{can}}{\underset{\text{gen}}{\vdash}} P() : \exists \overline{\alpha}, \alpha.(\quad, \mathcal{R})}$$

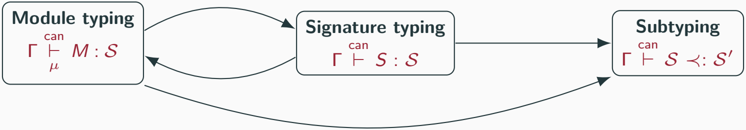
The canonical system



C-Typ-Mod-AppGen

$$\frac{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} P : (\tau, () \to \exists \overline{\alpha}.\mathcal{R})}{\Gamma \overset{\text{can}}{\underset{\text{gen}}{\vdash}} P() : \exists \overline{\alpha}, \alpha. (\alpha, \mathcal{R})}$$

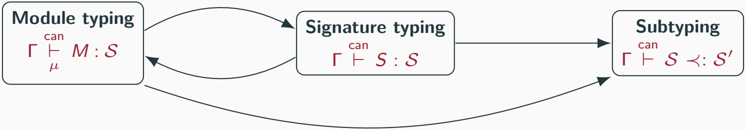
The canonical system



C-Typ-Mod-AppGen

$$\frac{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} P : (\tau, () \rightarrow \exists \overline{\alpha}.\mathcal{R})}{\Gamma \overset{\text{can}}{\underset{\text{gen}}{\vdash}} P() : \exists \overline{\alpha}, \alpha. (\alpha, \mathcal{R})}$$

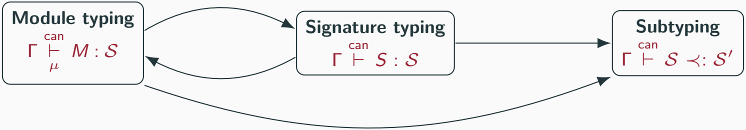
The canonical system



C-Typ-Mod-AppFct

$$\frac{Y \notin \Gamma}{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} (Y : S) \to M :}$$

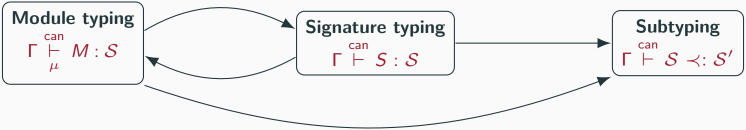
The canonical system



C-Typ-Mod-AppFct

$$\frac{\Gamma \overset{\text{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{C} \qquad Y \notin \Gamma}{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} (Y : S) \to M :}$$

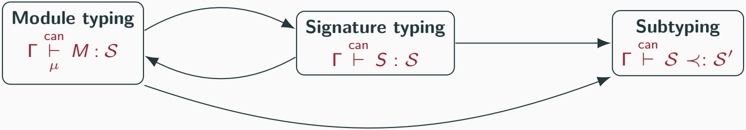
The canonical system



C-Typ-Mod-AppFct

$$\frac{\Gamma \overset{\text{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{C} \qquad \Gamma, \overline{\alpha}, (Y : \mathcal{C}) \overset{\text{can}}{\underset{\text{app}}{\vdash}} M : \exists\overline{\beta}.\,(\tau, \mathcal{R}) \qquad Y \notin \Gamma}{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} (Y : S) \to M :}$$

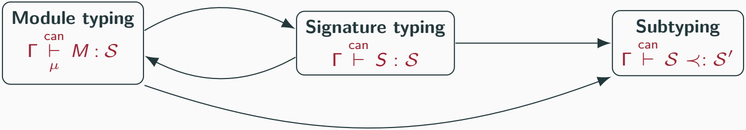
The canonical system



C-Typ-Mod-AppFct

$$\frac{\Gamma \overset{\text{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{C} \qquad \Gamma, \overline{\alpha}, (Y : \mathcal{C}) \overset{\text{can}}{\underset{\text{app}}{\vdash}} M : \exists\overline{\beta}.\,(\tau, \mathcal{R}) \qquad Y \notin \Gamma}{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} (Y : S) \to M : \exists\overline{\beta'}.}$$

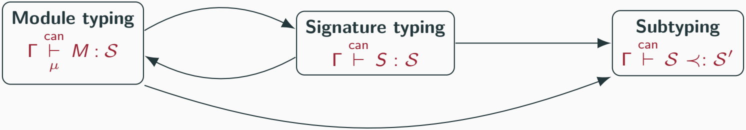
The canonical system



C-Typ-Mod-AppFct

$$\frac{\Gamma \overset{\text{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{C} \qquad \Gamma, \overline{\alpha}, (Y : \mathcal{C}) \overset{\text{can}}{\underset{\text{app}}{\vdash}} M : \exists\overline{\beta}.\,(\tau, \mathcal{R}) \qquad Y \notin \Gamma}{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} (Y : S) \to M : \exists\overline{\beta'}.\,(\qquad , \qquad)\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]}$$

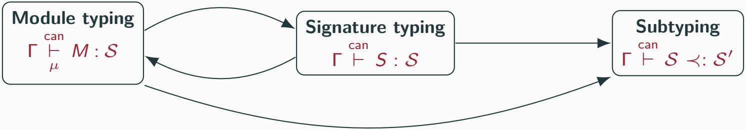
The canonical system



C-Typ-Mod-AppFct

$$\frac{\Gamma \overset{\text{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{C} \qquad \Gamma, \overline{\alpha}, (Y : \mathcal{C}) \overset{\text{can}}{\underset{\text{app}}{\vdash}} M : \exists\overline{\beta}.\,(\tau, \mathcal{R}) \qquad Y \notin \Gamma}{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} (Y : S) \to M : \exists\overline{\beta'}.\,(\qquad , \forall\overline{\alpha}.\,\mathcal{C} \to \mathcal{R})\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]}$$

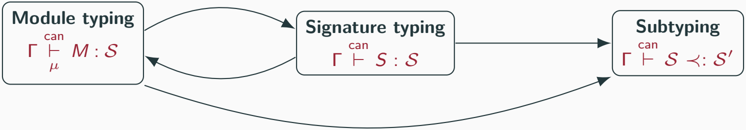
The canonical system



C-Typ-Mod-AppFct

$$\frac{\Gamma \overset{\text{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{C} \qquad \Gamma, \overline{\alpha}, (Y : \mathcal{C}) \overset{\text{can}}{\underset{\text{app}}{\vdash}} M : \exists\overline{\beta}.\,(\tau, \mathcal{R}) \qquad Y \notin \Gamma}{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} (Y : S) \rightarrow M : \exists\overline{\beta'}.\,(\lambda\overline{\alpha}.\tau, \forall\overline{\alpha}.\,\mathcal{C} \rightarrow \mathcal{R}) \left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]}$$

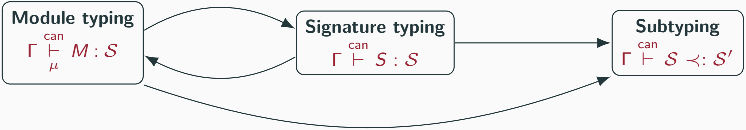
The canonical system



C-Typ-Mod-AppFct

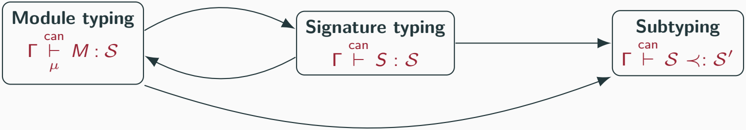
$$\frac{\Gamma \overset{\text{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{C} \qquad \Gamma, \overline{\alpha}, (Y : \mathcal{C}) \overset{\text{can}}{\underset{\text{app}}{\vdash}} M : \exists\overline{\beta}.\,(\tau, \mathcal{R}) \qquad Y \notin \Gamma}{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} (Y : S) \to M : \exists\overline{\beta'}.\,(\lambda\overline{\alpha}.\tau, \forall\overline{\alpha}.\,\mathcal{C} \to \mathcal{R})\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]}$$

The canonical system



C-Typ-Mod-Proj

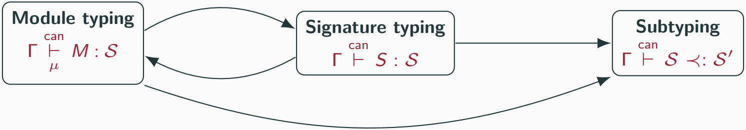
$$\frac{}{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} M.X :}$$



C-Typ-Mod-Proj

$$\frac{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} M : \exists \overline{\alpha}.\left(\tau, \texttt{sig}\ \overline{\mathcal{D}}\ \texttt{end}\right)}{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} M.X :}$$

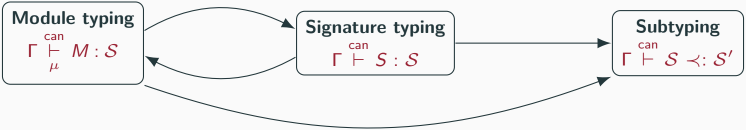
The canonical system



C-TYP-MOD-PROJ

$$\frac{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} M : \exists \overline{\alpha}.\ \left(\tau, \texttt{sig}\ \overline{\mathcal{D}}\ \texttt{end}\right) \qquad \text{module}\, X : \mathcal{C} \in \overline{D}}{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} M.X :}$$

The canonical system



C-Typ-Mod-Proj

$$\frac{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} M : \exists\overline{\alpha}.\left(\tau, \texttt{sig}\ \overline{\mathcal{D}}\ \texttt{end}\right) \qquad \text{module}\, X : \mathcal{C} \in \overline{D}}{\Gamma \overset{\text{can}}{\underset{\mu}{\vdash}} M.X : \exists\overline{\alpha}.\mathcal{C}}$$

A rich history of ML-modules

First formalizations of module systems

A rich history of ML-modules

First formalizations of module systems

- SML modules: Harper et al. [1989]

A rich history of ML-modules

First formalizations of module systems

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

A rich history of ML-modules

First formalizations of module systems

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

Existential types for modules

A rich history of ML-modules

First formalizations of module systems

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

Existential types for modules

- First with dependent types: MacQueen [1986]

A rich history of ML-modules

First formalizations of module systems

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

Existential types for modules

- First with dependent types: MacQueen [1986]
- Then with existential types: Russo [2004]

A rich history of ML-modules

First formalizations of module systems

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

Existential types for modules

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

A rich history of ML-modules

First formalizations of module systems

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

Existential types for modules

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

Formalization via elaboration

A rich history of ML-modules

First formalizations of module systems

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

Existential types for modules

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

Formalization via elaboration

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

A rich history of ML-modules

First formalizations of module systems

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

Existential types for modules

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

Formalization via elaboration

- **Translation of a significant part of SML into F^{ω}: F-ing Rossberg et al. [2014]**
- Core and module languages merged in F^{ω}: 1ML Rossberg [2018]

A user friendly but limited syntax: the source system

Anchoring - a reverse process 1/3

Abstract type fields

```
(* Canonical *)
∃α. module M : sig
  type t = α
  val x : α
  ...
end
```

```
(* Source *)
module M : sig



end
```

Anchoring - a reverse process 1/3

Abstract type fields

```
(* Canonical *)
∃α. module M : sig
  type t = α
  val x : α
  ...
end
```

```
(* Source *)
module M : sig
  type t
  val x : t
  ...
end
```

Anchoring - a reverse process 1/3

Abstract type fields

- Merge quantifier and structural information

```
(* Canonical *)
∃α. module M : sig
  type t = α
  val x : α
  ...
end
```

```
(* Source *)
module M : sig
  type t
  val x : t
  ...
end
```

Anchoring - a reverse process 1/3

Abstract type fields

- Merge quantifier and structural information
- The first *usage point* must be suitable to give a path

```
(* Canonical *)
∃α. module M : sig
  type t = α
  val x : α
  ...
end
```

```
(* Source *)
module M : sig
  type t
  val x : t
  ...
end
```

Anchoring - a reverse process 1/3

Abstract type fields

- Merge quantifier and structural information
- The first *usage point* must be suitable to give a path

```
(* Canonical *)
∃α. module M : sig
  type t = α * bool
  val x : α
  ...
end
```

```
(* Source *)
module M : sig
  type t ✗
  val x : t
  ...
end
```

Anchoring - a reverse process 1/3

Abstract type fields

- Merge quantifier and structural information
- The first *usage point* must be suitable to give a path

> Anchoring map $\theta : \alpha \mapsto P.t$

```
(* Canonical *)
∃α. module M : sig
  type t = α * bool
  val x : α
  ...
end
```

```
(* Source *)
module M : sig
  type t ✗
  val x : t
  ...
end
```

Anchoring - a reverse process 2/3

Module identities

```
(* Canonical *)
∃α_X. module M : sig


    ...
end
```

```
(* Source *)
module M : sig



end
```

Anchoring - a reverse process 2/3

Module identities

```
(* Canonical *)
∃α_X. module M : sig
  module X1 : (α_X, R_1)

    ...
end
```

```
(* Source *)
module M : sig


    ...
end
```

Anchoring - a reverse process 2/3

Module identities

```
(* Canonical *)
∃α_X. module M : sig
  module X1 : (α_X, R_1)
  module X2 : (α_X, R_2)
   ...
end
```

```
(* Source *)
module M : sig


   ...
end
```

Anchoring - a reverse process 2/3

Module identities

- Identity sharing is only expressed as restriction of a common ancestor

```
(* Canonical *)
∃α_X. module M : sig
  module X1 : (α_X, R_1)
  module X2 : (α_X, R_2)
   ...
end
```

$\exists \alpha_X.$ **module** M : **sig** / **module** X1 : $(\alpha_X, \mathcal{R}_1)$ / **module** X2 : $(\alpha_X, \mathcal{R}_2)$

```
(* Source *)
module M : sig
  module X1 : S1

   ...
end
```

Anchoring - a reverse process 2/3

Module identities

- Identity sharing is only expressed as restriction of a common ancestor

```
(* Canonical *)
∃α_X. module M : sig
  module X1 : (α_X, R_1)
  module X2 : (α_X, R_2)
   ...
end
```

```
(* Source *)
module M : sig
  module X1 : S1
  module X2 : (= X1 < S2)
   ...
end
```

Anchoring - a reverse process 2/3

Module identities

- Identity sharing is only expressed as restriction of a common ancestor
- All identities must be ascriptions of the anchoring point

(∗ Canonical ∗)
$\exists \alpha_X$. **module** M : **sig**
  module X1 : $(\alpha_X, \mathcal{R}_1)$
  module X2 : $(\alpha_X, \mathcal{R}_2)$
  ...
end

(∗ Source ∗)
module M : **sig**
  module X1 : S1
  module X2 : $(=$ X1 $<$ S2$)$
  ...
end

Anchoring - a reverse process 2/3

Module identities

- Identity sharing is only expressed as restriction of a common ancestor
- All identities must be ascriptions of the anchoring point

> Anchoring map $\theta : \alpha \mapsto (P, \mathcal{R})$

(* Canonical *)
$\exists \alpha_X$. **module** M : **sig**
  module X1 : $(\alpha_X, \mathcal{R}_1)$
  module X2 : $(\alpha_X, \mathcal{R}_2)$
   ...
end

(* Source *)
module M : **sig**
  module X1 : S1
  module X2 : (= X1 $<$ S2)
   ...
end

Anchoring - a reverse process 3/3

High order abstract types

```
(* Canonical *)
∃γ. module M : sig
end
```

```
(* Source *)
module M : sig
end
```

Anchoring - a reverse process 3/3

High order abstract types

```
(* Canonical *)
∃γ. module M : sig
  module F1 : ∀β, C₁ →



end
```

```
(* Source *)
module M : sig




end
```

Anchoring - a reverse process 3/3

High order abstract types

```
(* Canonical *)
∃γ. module M : sig
  module F1 : ∀β, C_1 →
    sig type t = γ(β) end


end
```

```
(* Source *)
module M : sig





end
```

Anchoring - a reverse process 3/3

High order abstract types

```
(* Canonical *)
∃γ. module M : sig
  module F1 : ∀β, C₁ →
    sig type t = γ(β) end
  module F2 : ∀β, C₂ →

end
```

```
(* Source *)
module M : sig




end
```

High order abstract types

```
(* Canonical *)
∃γ. module M : sig
  module F1 : ∀β,C1 →
    sig type t = γ(β) end
  module F2 : ∀β,C2 →
    sig type t = γ(β) end
end
```

```
(* Source *)
module M : sig




end
```

High order abstract types

- Type operators are only represented as functor applications

```
(* Canonical *)
∃γ. module M : sig
  module F1 : ∀β, C₁ →
    sig type t = γ(β) end
  module F2 : ∀β, C₂ →
    sig type t = γ(β) end
end
```

```
(* Source *)
module M : sig




end
```

Anchoring - a reverse process 3/3

High order abstract types

- Type operators are only represented as functor applications

```
(* Canonical *)
∃γ. module M : sig
  module F1 : ∀β, C₁ →
    sig type t = γ(β) end
  module F2 : ∀β, C₂ →
    sig type t = γ(β) end
end
```

```
(* Source *)
module M : sig
  module F1 : functor (Y:S1) ->



end
```

Anchoring - a reverse process 3/3

High order abstract types

- Type operators are only represented as functor applications

```
(* Canonical *)
∃γ. module M : sig
  module F1 : ∀β, C₁ →
    sig type t = γ(β) end
  module F2 : ∀β, C₂ →
    sig type t = γ(β) end
end
```

```
(* Source *)
module M : sig
  module F1 : functor (Y:S1) ->
    sig type t end


end
```

Anchoring - a reverse process 3/3

High order abstract types

- Type operators are only represented as functor applications

```
(* Canonical *)
∃γ. module M : sig
  module F1 : ∀β, C1 →
    sig type t = γ(β) end
  module F2 : ∀β, C2 →
    sig type t = γ(β) end
end
```

```
(* Source *)
module M : sig
  module F1 : functor (Y:S1) ->
    sig type t end
  module F2 : functor (Y:S2) ->

end
```

Anchoring - a reverse process 3/3

High order abstract types

- Type operators are only represented as functor applications

```
(* Canonical *)
∃γ. module M : sig
  module F1 : ∀β, C_1 →
    sig type t = γ(β) end
  module F2 : ∀β, C_2 →
    sig type t = γ(β) end
end
```

```
(* Source *)
module M : sig
  module F1 : functor (Y:S1) ->
    sig type t end
  module F2 : functor (Y:S2) ->
    sig type t = F1(Y).t end
end
```

Anchoring - a reverse process 3/3

High order abstract types

- Type operators are only represented as functor applications
- All usages must be obtainable by some application of the *anchoring point*

```
(* Canonical *)
∃γ. module M : sig
  module F1 : ∀β, C_1 →
    sig type t = γ(β) end
  module F2 : ∀β, C_2 →
    sig type t = γ(β) end
end
```

```
(* Source *)
module M : sig
  module F1 : functor (Y:S1) ->
    sig type t end
  module F2 : functor (Y:S2) ->
    sig type t = F1(Y).t end
end
```

Anchoring - a reverse process 3/3

High order abstract types

- Type operators are only represented as functor applications
- All usages must be obtainable by some application of the *anchoring point*

> Anchoring map $\theta : \alpha \mapsto \lambda\overline{\alpha}.\lambda\overline{X}.(P.t, \mathcal{R}, \overline{\tau})$

```
(* Canonical *)
∃γ. module M : sig
  module F1 : ∀β, C_1 →
    sig type t = γ(β) end
  module F2 : ∀β, C_2 →
    sig type t = γ(β) end
end
```

```
(* Source *)
module M : sig
  module F1 : functor (Y:S1) ->
    sig type t end
  module F2 : functor (Y:S2) ->
    sig type t = F1(Y).t end
end
```

Challenges for a source system

Challenges for a source system

Signature avoidance

Challenges for a source system

Signature avoidance

- abstract type fields (possibly high-order)

Challenges for a source system

Signature avoidance

- abstract type fields (possibly high-order)
- module identities

Challenges for a source system

Signature avoidance

- abstract type fields (possibly high-order)
- module identities

Implicit quantifiers

Challenges for a source system

Signature avoidance

- abstract type fields (possibly high-order)
- module identities

Implicit quantifiers

- Strengthening (deep rewrites)

Challenges for a source system

Signature avoidance

- abstract type fields (possibly high-order)
- module identities

Implicit quantifiers

- Strengthening (deep rewrites)

Type and signatures equivalence

Challenges for a source system

Signature avoidance

- abstract type fields (possibly high-order)
- module identities

Implicit quantifiers

- Strengthening (deep rewrites)

Type and signatures equivalence

- Types have several available aliases

Challenges for a source system

Signature avoidance

- abstract type fields (possibly high-order)
- module identities

Implicit quantifiers

- Strengthening (deep rewrites)

Type and signatures equivalence

- Types have several available aliases
- Prevent in-lining of module types

Elaboration into F^{ω}: guarantees for the canonical system

Example of encoding into $\mathbf{F}^\omega$

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists \alpha$$

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists\alpha \left\{ \right.$$

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists \alpha \left\{ \quad \ell_{X1} : \left\{ \quad \right. \right.$$

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists\alpha \left\{ \quad \ell_{X1} \colon \left\{ \begin{array}{l} \ell_t : \langle\!\langle \alpha \rangle\!\rangle \\ \\ \end{array} \right. \\ \\ \right.$$

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists \alpha \left\{ \ell_{X1} : \left\{ \ell_t : \forall \beta . \beta\, \alpha \to \beta\, \alpha \right. \right.$$

Example of encoding into F^{ω}

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists \alpha \left\{ \quad \ell_{X1} \colon \begin{cases} \ell_t : \langle\!\langle \alpha \rangle\!\rangle \\ \ell_x : \alpha \\ \end{cases} \right.$$

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists \alpha \left\{ \ell_{X1} : \begin{cases} \ell_t : \langle\!\langle \alpha \rangle\!\rangle \\ \ell_x : \alpha \\ \ell_u : \langle\!\langle \mathtt{int} \times \alpha \rangle\!\rangle \end{cases} \right.$$

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists\alpha \left\{ \quad \ell_{X1} \colon \left\{ \begin{array}{l} \ell_t : \langle\!\langle \alpha \rangle\!\rangle \\ \ell_x : \alpha \\ \ell_u : \langle\!\langle \mathtt{int} \times \alpha \rangle\!\rangle \end{array} \right\} \right.$$

Example of encoding into F^{ω}

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists\alpha \left\{ \begin{array}{l} \ell_{X1} : \left\{ \begin{array}{l} \ell_t : \langle\!\langle \alpha \rangle\!\rangle \\ \ell_x : \alpha \\ \ell_u : \langle\!\langle \mathtt{int} \times \alpha \rangle\!\rangle \end{array} \right\} \\ \\ \ell_{X2} : \left\{ \begin{array}{l} \\ \\ \end{array} \right. \end{array} \right.$$

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists \alpha \left\{ \begin{array}{l} \ell_{X1} : \left\{ \begin{array}{l} \ell_t : \langle\!\langle \alpha \rangle\!\rangle \\ \ell_x : \alpha \\ \ell_u : \langle\!\langle \mathtt{int} \times \alpha \rangle\!\rangle \end{array} \right\} \\ \ell_{X2} : \left\{ \begin{array}{l} \ell_t : \langle\!\langle \alpha \times \mathtt{bool} \rangle\!\rangle \\ \end{array} \right. \end{array} \right.$$

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists\alpha \left\{ \begin{array}{l} \ell_{X1}\colon \left\{ \begin{array}{l} \ell_t : \langle\!\langle \alpha \rangle\!\rangle \\ \ell_x : \alpha \\ \ell_u : \langle\!\langle \mathtt{int} \times \alpha \rangle\!\rangle \end{array} \right\} \\ \ell_{X2}\colon \left\{ \begin{array}{l} \ell_t : \langle\!\langle \alpha \times \mathtt{bool} \rangle\!\rangle \\ \ell_u : \langle\!\langle \alpha \times \mathtt{int} \rangle\!\rangle \end{array} \right. \end{array} \right.$$

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists \alpha \left\{ \begin{array}{l} \ell_{X1}\colon \left\{ \begin{array}{l} \ell_t : \langle\!\langle \alpha \rangle\!\rangle \\ \ell_x : \alpha \\ \ell_u : \langle\!\langle \mathtt{int} \times \alpha \rangle\!\rangle \end{array} \right\} \\ \ell_{X2}\colon \left\{ \begin{array}{l} \ell_t : \langle\!\langle \alpha \times \mathtt{bool} \rangle\!\rangle \\ \ell_u : \langle\!\langle \alpha \times \mathtt{int} \rangle\!\rangle \end{array} \right\} \end{array} \right.$$

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists \alpha \left\{ \begin{array}{l} \text{module } X_1 : \left\{ \begin{array}{l} \text{type } t = \alpha \\ \text{val } x : \alpha \\ \text{type } u = \texttt{int} \times \alpha \end{array} \right\} \\ \text{module } X_2 : \left\{ \begin{array}{l} \text{type } t = \alpha \times \texttt{bool} \\ \text{type } u = \alpha \times \texttt{int} \end{array} \right\} \end{array} \right\}$$

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded module

Encoded signature

$$\Pi = \exists\alpha \left\{ \begin{array}{l} \text{module } X_1 : \left\{ \begin{array}{l} \text{type } t = \alpha \\ \text{val } x : \alpha \\ \text{type } u = \texttt{int} \times \alpha \end{array} \right\} \\ \text{module } X_2 : \left\{ \begin{array}{l} \text{type } t = \alpha \times \texttt{bool} \\ \text{type } u = \alpha \times \texttt{int} \end{array} \right\} \end{array} \right\}$$

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists\alpha \left\{ \begin{array}{l} \text{module } X_1 : \left\{ \begin{array}{l} \text{type } t = \alpha \\ \text{val } x : \alpha \\ \text{type } u = \texttt{int} \times \alpha \end{array} \right\} \\ \text{module } X_2 : \left\{ \begin{array}{l} \text{type } t = \alpha \times \texttt{bool} \\ \text{type } u = \alpha \times \texttt{int} \end{array} \right\} \end{array} \right\}$$

Encoded module

$$E =$$

Example of encoding into F$^\omega$

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists\alpha \left\{ \begin{array}{l} \text{module } X_1 : \left\{ \begin{array}{l} \text{type } t = \alpha \\ \text{val } x : \alpha \\ \text{type } u = \texttt{int} \times \alpha \end{array} \right\} \\ \text{module } X_2 : \left\{ \begin{array}{l} \text{type } t = \alpha \times \texttt{bool} \\ \text{type } u = \alpha \times \texttt{int} \end{array} \right\} \end{array} \right\}$$

Encoded module

$$E = \{\ell_{X1} = \{\ell_t = \langle\langle \texttt{int} \rangle\rangle, \ell_x = 42, \ell_u = \langle\langle \texttt{int} \times \texttt{int} \rangle\rangle\}\}$$

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists \alpha \left\{ \begin{array}{l} \text{module } X_1 : \left\{ \begin{array}{l} \text{type } t = \alpha \\ \text{val } x : \alpha \\ \text{type } u = \texttt{int} \times \alpha \end{array} \right\} \\ \text{module } X_2 : \left\{ \begin{array}{l} \text{type } t = \alpha \times \texttt{bool} \\ \text{type } u = \alpha \times \texttt{int} \end{array} \right\} \end{array} \right\}$$

Encoded module

$$E = \text{pack}\langle \texttt{int}, \{\ell_{X1} = \{\ell_t = \langle\langle \texttt{int} \rangle\rangle, \ell_x = 42, \ell_u = \langle\langle \texttt{int} \times \texttt{int} \rangle\rangle\}\}\rangle$$

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists\alpha \left\{ \begin{array}{l} \text{module } X_1 : \left\{ \begin{array}{l} \text{type } t = \alpha \\ \text{val } x : \alpha \\ \text{type } u = \texttt{int} \times \alpha \end{array} \right\} \\ \text{module } X_2 : \left\{ \begin{array}{l} \text{type } t = \alpha \times \texttt{bool} \\ \text{type } u = \alpha \times \texttt{int} \end{array} \right\} \end{array} \right\}$$

Encoded module

$$E = \text{pack}\langle \texttt{int}, \{\ell_{X1} = \{\ell_t = \langle\!\langle \texttt{int} \rangle\!\rangle, \ell_x = 42, \ell_u = \langle\!\langle \texttt{int} \times \texttt{int} \rangle\!\rangle\}\}\rangle$$
$$\{\ell_{X2} = \{\ell_t = \langle\!\langle \alpha \times \texttt{bool} \rangle\!\rangle, \ell_u = \langle\!\langle \alpha \times \texttt{int} \rangle\!\rangle\}\}$$

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists\alpha \left\{ \begin{array}{l} \text{module } X_1 : \left\{ \begin{array}{l} \text{type } t = \alpha \\ \text{val } x : \alpha \\ \text{type } u = \texttt{int} \times \alpha \end{array} \right\} \\ \text{module } X_2 : \left\{ \begin{array}{l} \text{type } t = \alpha \times \texttt{bool} \\ \text{type } u = \alpha \times \texttt{int} \end{array} \right\} \end{array} \right\}$$

Encoded module

$$\begin{array}{l} E = \text{unpack}\langle \alpha, y_1 \rangle = \text{pack}\langle \texttt{int}, \{\ell_{X1} = \{\ell_t = \langle\!\langle \texttt{int} \rangle\!\rangle, \ell_x = 42, \ell_u = \langle\!\langle \texttt{int} \times \texttt{int} \rangle\!\rangle\}\}\rangle \\ \quad\quad \text{in} \quad\quad\quad\quad \{\ell_{X2} = \{\ell_t = \langle\!\langle \alpha \times \texttt{bool} \rangle\!\rangle, \ell_u = \langle\!\langle \alpha \times \texttt{int} \rangle\!\rangle\}\} \end{array}$$

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists\alpha \left\{ \begin{array}{l} \text{module } X_1 : \left\{ \begin{array}{l} \text{type } t = \alpha \\ \text{val } x : \alpha \\ \text{type } u = \texttt{int} \times \alpha \end{array} \right\} \\ \text{module } X_2 : \left\{ \begin{array}{l} \text{type } t = \alpha \times \texttt{bool} \\ \text{type } u = \alpha \times \texttt{int} \end{array} \right\} \end{array} \right\}$$

Encoded module

$$\begin{array}{l} E = \text{unpack}\langle \alpha, y_1 \rangle = \text{pack}\langle \texttt{int}, \{\ell_{X1} = \{\ell_t = \langle\!\langle \texttt{int} \rangle\!\rangle, \ell_x = 42, \ell_u = \langle\!\langle \texttt{int} \times \texttt{int} \rangle\!\rangle\}\}\rangle \\ \quad \text{in unpack}\langle \varnothing, y_2 \rangle = \{\ell_{X2} = \{\ell_t = \langle\!\langle \alpha \times \texttt{bool} \rangle\!\rangle, \ell_u = \langle\!\langle \alpha \times \texttt{int} \rangle\!\rangle\}\} \\ \quad\quad \text{in} \end{array}$$

Example of encoding into F^{ω}

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists\alpha \left\{ \begin{array}{l} \text{module } X_1 : \left\{ \begin{array}{l} \text{type } t = \alpha \\ \text{val } x : \alpha \\ \text{type } u = \texttt{int} \times \alpha \end{array} \right\} \\ \text{module } X_2 : \left\{ \begin{array}{l} \text{type } t = \alpha \times \texttt{bool} \\ \text{type } u = \alpha \times \texttt{int} \end{array} \right\} \end{array} \right\}$$

Encoded module

$$\begin{array}{l} E = \text{unpack}\langle \alpha, y_1 \rangle = \text{pack}\langle \texttt{int}, \{\ell_{X1} = \{\ell_t = \langle\!\langle \texttt{int} \rangle\!\rangle, \ell_x = 42, \ell_u = \langle\!\langle \texttt{int} \times \texttt{int} \rangle\!\rangle\}\}\rangle \\ \quad \text{in unpack}\langle \varnothing, y_2 \rangle = \{\ell_{X2} = \{\ell_t = \langle\!\langle \alpha \times \texttt{bool} \rangle\!\rangle, \ell_u = \langle\!\langle \alpha \times \texttt{int} \rangle\!\rangle\}\} \\ \quad\quad \text{in pack}\langle \alpha, \{\ell_{X1} = (y_1.\ell_{X1}), \ell_{X2} = (y_2.\ell_{X2})\}\rangle \end{array}$$

The issue of *skolemisation*

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists \alpha . \mathcal{R}$$

Encoded module

$$E = \mathsf{pack}\langle \alpha, \ldots \rangle$$

The issue of *skolemisation*

Source code

```
module M (Y:S) = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists \alpha . \mathcal{R}$$

Encoded module

$$E = \mathsf{pack}\langle \alpha, \dots \rangle$$

The issue of *skolemisation*

Source code

```
module M (Y:S) = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \forall \beta . \mathcal{C} \rightarrow \exists \alpha . \mathcal{R}$$

Encoded module

$$E = \mathsf{pack}\langle \alpha, \ldots \rangle$$

The issue of *skolemisation*

Source code

```
module M (Y:S) = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists \alpha'.\forall \beta.\mathcal{C} \rightarrow \mathcal{R}\Big[\alpha \mapsto \alpha'(\beta)\Big]$$

Encoded module

$$E = \mathsf{pack}\langle \alpha, \ldots \rangle$$

The issue of *skolemisation*

Source code

```
module M (Y:S) = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists\alpha'.\forall\beta.\mathcal{C} \to \mathcal{R}\Big[\alpha \mapsto \alpha'(\beta)\Big]$$

Encoded module

$$E = \Lambda\beta.\lambda(Y : \mathcal{C}).\mathsf{pack}\langle\alpha, \dots\rangle$$

The issue of *skolemisation*

Source code

```
module M (Y:S) = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists \alpha'.\forall \beta.\mathcal{C} \to \mathcal{R}\Big[\alpha \mapsto \alpha'(\beta)\Big]$$

Encoded module

$$E = \Lambda \beta.\lambda(Y : \mathcal{C}).\mathsf{pack}\langle \alpha, \dots \rangle$$

The issue of *skolemisation*

Source code

```
module M (Y:S) = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \forall \beta.\mathcal{C} \rightarrow \exists (\alpha = \tau).\mathcal{R}$$

Encoded module

$$E = \Lambda \beta.\lambda (Y : \mathcal{C}).\mathsf{pack}\langle \alpha, \ldots \rangle$$

The issue of ***skolemisation***

Source code

```
module M (Y:S) = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists(\alpha' = \lambda\beta.\tau).\forall\beta.\mathcal{C} \to \mathcal{R}\left[\alpha \mapsto \alpha'(\beta)\right]$$

Encoded module

$$E = \Lambda\beta.\lambda(Y : \mathcal{C}).\mathsf{pack}\langle\alpha, \dots\rangle$$

The issue of *skolemisation*

Source code

```
module M (Y:S) = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists(\alpha' = \lambda\beta.\tau).\forall\beta.\mathcal{C} \to \mathcal{R}\Big[\alpha \mapsto \alpha'(\beta)\Big]$$

Encoded module

$$E = \mathsf{pack}\langle\alpha, \dots\rangle$$

The issue of *skolemisation*

Source code

```
module M (Y:S) = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists(\alpha' = \lambda\beta.\tau).\forall\beta.\mathcal{C} \to \mathcal{R}\Big[\alpha \mapsto \alpha'(\beta)\Big]$$

Encoded module

$$E = \ldots \text{ as } \exists(\alpha = \tau).\mathcal{R}$$

The issue of *skolemisation*

Source code

```
module M (Y:S) = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = int * t
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * int
  end
end
```

Encoded signature

$$\Pi = \exists(\alpha' = \lambda\beta.\tau).\forall\beta.\mathcal{C} \to \mathcal{R}\Big[\alpha \mapsto \alpha'(\beta)\Big]$$

Encoded module

$$E = \text{skolem}_{\forall}(\Lambda\beta.\text{skolem}_{\to}(\lambda(Y : \mathcal{C}).(\ldots \text{ as } \exists(\alpha = \tau).\mathcal{R})))$$

F^{ω} with skolemisation

New constructs

F^ω with skolemisation

New constructs

$$\begin{aligned} \tau ::= &\ \dots \\ &\mid \exists\alpha.\tau \end{aligned}$$

F^{ω} with skolemisation

New constructs

$$\begin{aligned}\tau ::= &\ \ldots \\ &\mid \exists \alpha . \tau \\ &\mid \exists \left(\alpha = \tau\right) . \tau\end{aligned}$$

F^ω with skolemisation

New constructs

$$
\begin{aligned}
\tau ::= \;& \ldots \\
\mid\;& \exists \alpha . \tau \\
\mid\;& \exists (\alpha = \tau) . \tau \\
e ::= \;& \ldots \\
\mid\;& \mathsf{pack}\, e \\
\mid\;& \mathsf{unpack} \langle \alpha, x \rangle = e \ \mathsf{in}\ e
\end{aligned}
$$

F^ω with skolemisation

New constructs

$$
\begin{aligned}
\tau ::=\ & \dots \\
& \mid \exists \alpha . \tau \\
& \mid \exists \left(\alpha = \tau \right) . \tau \\
e ::=\ & \dots \\
& \mid \mathsf{pack}\, e \\
& \mid \mathsf{unpack} \langle \alpha, x \rangle = e \text{ in } e \\
& \mid e \text{ as } \exists (\alpha = \tau) . \tau
\end{aligned}
$$

F^ω with skolemisation

New constructs

$$
\begin{aligned}
\tau ::= &\ \ldots \\
&\mid \exists \alpha.\tau \\
&\mid \exists\left(\alpha = \tau\right).\tau \\
e ::= &\ \ldots \\
&\mid \mathsf{pack}\ e \\
&\mid \mathsf{unpack}\langle \alpha, x\rangle = e\ \mathsf{in}\ e \\
&\mid e\ \mathsf{as}\ \exists(\alpha = \tau).\tau \\
&\mid \mathsf{hide}\langle \alpha, x\rangle = e\ \mathsf{in}\ e
\end{aligned}
$$

F^ω with skolemisation

New constructs

$$
\begin{aligned}
\tau ::=\;& \ldots \\
& \mid \exists \alpha.\tau \\
& \mid \exists\,(\alpha = \tau)\,.\tau \\
e ::=\;& \ldots \\
& \mid \mathsf{pack}\, e \\
& \mid \mathsf{unpack}\langle \alpha, x\rangle = e \;\mathsf{in}\; e \\
& \mid e \;\mathsf{as}\; \exists(\alpha = \tau).\tau \\
& \mid \mathsf{hide}\langle \alpha, x\rangle = e \;\mathsf{in}\; e \\
& \mid \mathsf{show}\, e
\end{aligned}
$$

$\mathbf{F}^{\omega}$ with skolemisation

New constructs

$$
\begin{aligned}
\tau ::= \; & \ldots \\
& \mid \exists \alpha . \tau \\
& \mid \exists \left(\alpha = \tau \right) . \tau \\
e ::= \; & \ldots \\
& \mid \mathsf{pack}\, e \\
& \mid \mathsf{unpack} \langle \alpha, x \rangle = e \;\mathsf{in}\; e \\
& \mid e \;\mathsf{as}\; \exists (\alpha = \tau) . \tau \\
& \mid \mathsf{hide} \langle \alpha, x \rangle = e \;\mathsf{in}\; e \\
& \mid \mathsf{show}\, e
\end{aligned}
$$

Skolem operators

F^ω with skolemisation

New constructs

$$
\begin{aligned}
\tau ::= &\ \dots \\
&\mid \exists\alpha.\tau \\
&\mid \exists\,(\alpha = \tau)\,.\tau \\
e ::= &\ \dots \\
&\mid \mathsf{pack}\, e \\
&\mid \mathsf{unpack}\langle\alpha, x\rangle = e \;\mathsf{in}\; e \\
&\mid e \;\mathsf{as}\; \exists(\alpha = \tau).\tau \\
&\mid \mathsf{hide}\langle\alpha, x\rangle = e \;\mathsf{in}\; e \\
&\mid \mathsf{show}\, e
\end{aligned}
$$

Skolem operators

- $\mathsf{skolem}_{\rightarrow}$:

$$
\begin{aligned}
\sigma &\quad\rightarrow\quad \exists(\alpha = \tau).\rho \\
\exists(\alpha = \tau).\sigma &\quad\rightarrow\quad \rho
\end{aligned}
$$

F^ω with skolemisation

New constructs

$$
\begin{aligned}
\tau ::=\ & \ldots \\
& \mid \exists\alpha.\tau \\
& \mid \exists\,(\alpha = \tau)\,.\tau \\
e ::=\ & \ldots \\
& \mid \mathsf{pack}\, e \\
& \mid \mathsf{unpack}\langle\alpha, x\rangle = e \;\mathsf{in}\; e \\
& \mid e \;\mathsf{as}\; \exists(\alpha = \tau).\tau \\
& \mid \mathsf{hide}\langle\alpha, x\rangle = e \;\mathsf{in}\; e \\
& \mid \mathsf{show}\, e
\end{aligned}
$$

Skolem operators

- $\mathsf{skolem}_{\rightarrow}$:

$$
\begin{aligned}
\sigma &\quad\rightarrow\quad \exists(\alpha = \tau).\rho \\
\exists(\alpha = \tau).\sigma &\quad\rightarrow\quad \rho
\end{aligned}
$$

- $\mathsf{skolem}_{\forall}$

$$
\begin{aligned}
\forall\beta.&\quad \exists(\alpha = \tau).\sigma \\
\exists(\alpha' = \lambda\beta.\tau).\quad \forall\beta.&\quad \sigma[\alpha \mapsto \alpha'(\beta)]
\end{aligned}
$$

F^ω with skolemisation

New constructs

$$
\begin{aligned}
\tau ::= \ & \dots \\
& \mid \exists \alpha . \tau \\
& \mid \exists \, (\alpha = \tau) \, . \tau \\
e ::= \ & \dots \\
& \mid \text{pack}\, e \\
& \mid \text{unpack}\langle \alpha, x \rangle = e \text{ in } e \\
& \mid e \text{ as } \exists (\alpha = \tau) . \tau \\
& \mid \text{hide}\langle \alpha, x \rangle = e \text{ in } e \\
& \mid \text{show}\, e
\end{aligned}
$$

Skolem operators

- $\text{skolem}_{\to}$:

$$
\begin{aligned}
\sigma \quad & \to \quad \exists (\alpha = \tau) . \rho \\
\exists (\alpha = \tau) . \sigma \quad & \to \quad \rho
\end{aligned}
$$

- $\text{skolem}_{\forall}$

$$
\begin{aligned}
& \forall \beta . \quad \exists (\alpha = \tau) . \sigma \\
\exists (\alpha' = \lambda \beta . \tau) . \quad & \forall \beta . \quad \sigma[\alpha \mapsto \alpha'(\beta)]
\end{aligned}
$$

A restricted encoding

$\to$ without show, only with skolem operators

F^ω with skolemisation

New constructs

$$
\begin{aligned}
\tau ::= \; & \ldots \\
& \mid \exists \alpha . \tau \\
& \mid \exists\, (\alpha = \tau)\, . \tau \\
e ::= \; & \ldots \\
& \mid \mathsf{pack}\ e \\
& \mid \mathsf{unpack}\langle \alpha, x \rangle = e \ \mathsf{in}\ e \\
& \mid e \ \mathsf{as}\ \exists (\alpha = \tau) . \tau \\
& \mid \mathsf{hide}\langle \alpha, x \rangle = e \ \mathsf{in}\ e \\
& \mid \mathsf{show}\ e
\end{aligned}
$$

Skolem operators

- $\mathsf{skolem}_{\rightarrow}$:

$$
\begin{aligned}
\sigma \quad & \rightarrow \quad \exists (\alpha = \tau) . \rho \\
\exists (\alpha = \tau) . \sigma \quad & \rightarrow \quad \rho
\end{aligned}
$$

- $\mathsf{skolem}_{\forall}$

$$
\begin{aligned}
\forall \beta . \quad & \exists (\alpha = \tau) . \sigma \\
\exists (\alpha' = \lambda \beta . \tau) . \quad \forall \beta . \quad & \sigma [\alpha \mapsto \alpha'(\beta)]
\end{aligned}
$$

A restricted encoding

$\rightarrow$ without show, only with skolem operators

$$
\Gamma \overset{\mathsf{can}}{\underset{\mathsf{app}}{\vdash}} M : \exists \overline{\alpha} . \mathcal{C} \iff \exists \overline{\tau}, e .\ \Gamma \overset{\mathsf{elab}}{\vdash} M : \exists (\overline{\alpha} = \overline{\tau}) . \mathcal{C} \rightsquigarrow e
$$

F^ω with skolemisation

New constructs

$$\begin{aligned}\tau ::= &\ \dots\\ &\mid \exists\alpha.\tau\\ &\mid \exists\,(\alpha = \tau)\,.\tau\\ e ::= &\ \dots\\ &\mid \mathsf{pack}\ e\\ &\mid \mathsf{unpack}\langle\alpha, x\rangle = e\ \mathsf{in}\ e\\ &\mid e\ \mathsf{as}\ \exists(\alpha = \tau).\tau\\ &\mid \mathsf{hide}\langle\alpha, x\rangle = e\ \mathsf{in}\ e\\ &\mid \mathsf{show}\ e\end{aligned}$$

Skolem operators

- $\mathsf{skolem}_{\rightarrow}$:

$$\begin{aligned}\sigma &\quad\rightarrow\quad \exists(\alpha = \tau).\rho\\ \exists(\alpha = \tau).\sigma &\quad\rightarrow\quad \rho\end{aligned}$$

- $\mathsf{skolem}_{\forall}$

$$\begin{aligned}&\forall\beta.\quad \exists(\alpha = \tau).\sigma\\ \exists(\alpha' = \lambda\beta.\tau).\quad &\forall\beta.\quad \sigma[\alpha \mapsto \alpha'(\beta)]\end{aligned}$$

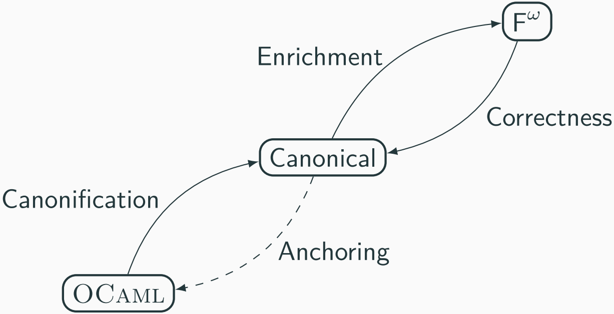
A restricted encoding

$\rightarrow$ without show, only with skolem operators

$$\Gamma \overset{\mathsf{can}}{\underset{\mathsf{app}}{\vdash}} M : \exists\overline{\alpha}.\mathcal{C} \iff \exists\overline{\tau}, e.\ \Gamma \overset{\mathsf{elab}}{\vdash} M : \exists(\overline{\alpha} = \overline{\tau}).\,\mathcal{C} \rightsquigarrow e$$

$$\Gamma \overset{\mathsf{can}}{\underset{\mathsf{gen}}{\vdash}} M : \exists\overline{\alpha}.\mathcal{C} \iff \exists e.\ \Gamma \overset{\mathsf{elab}}{\vdash} M : \exists\overline{\alpha}.\,\mathcal{C} \rightsquigarrow e$$

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
  - 1^{st} Class modules, *with*-constraints, etc.

Future work

- Support a more significant subset of OCaml
  - 1^{st} Class modules, *with*-constraints, etc.
  - Abstract signatures

Future work

- Support a more significant subset of OCaml
  - 1^{st} Class modules, *with*-constraints, etc.
  - Abstract signatures
  - Recursive modules

Future work

- Support a more significant subset of OCaml
  - 1^{st} Class modules, *with*-constraints, etc.
  - Abstract signatures
  - Recursive modules
- Mechanize (Coq)

Future work

- Support a more significant subset of OCaml
  - 1^{st} Class modules, *with*-constraints, etc.
  - Abstract signatures
  - Recursive modules
- Mechanize (Coq)
- Implement canonical-inspired algorithms in the typechecker

Future work

- Support a more significant subset of OCaml
  - 1^{st} Class modules, *with*-constraints, etc.
  - Abstract signatures
  - Recursive modules
- Mechanize (Coq)
- Implement canonical-inspired algorithms in the typechecker
- Explore the challenges of extending the syntax with existentials

Takeway

Takeway

1. An 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 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

Takeway

1. An 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

Takeway

1. An 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
3. A clean framework for other features and future extensions of the OCaml module system

Takeway

1. An 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
3. A clean framework for other features and future extensions of the OCaml module system
4. Formal guarantees through an improved elaboration into F^ω

References

D. DREYER. Recursive type generativity. *Journal of Functional Programming*, 17(4-5): 433–471, 2007. doi: 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`.

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`.

References ii

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.

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.