Retrofitting and strengthening the ML module system

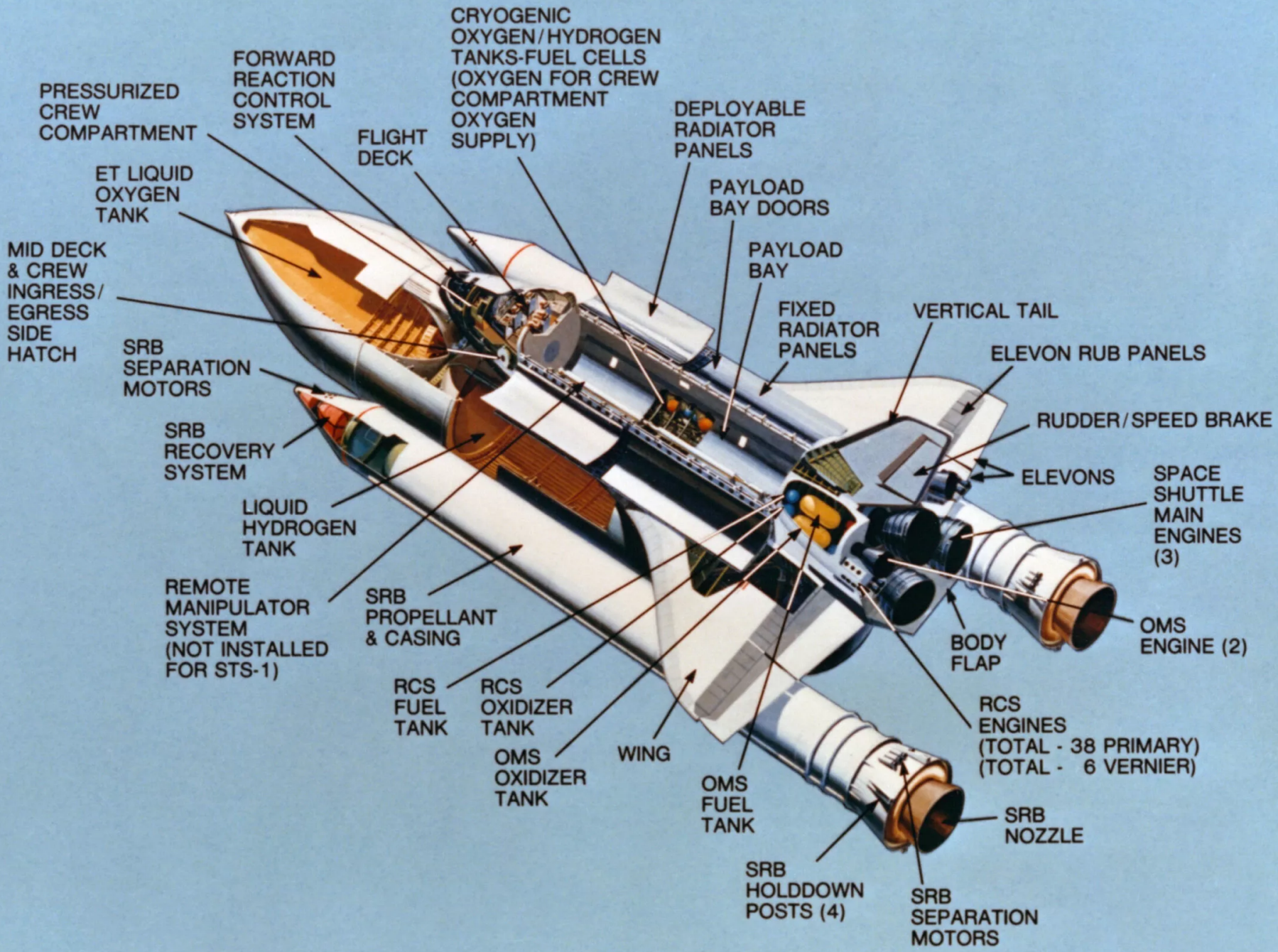
PhD Defense *- 11/12/24*

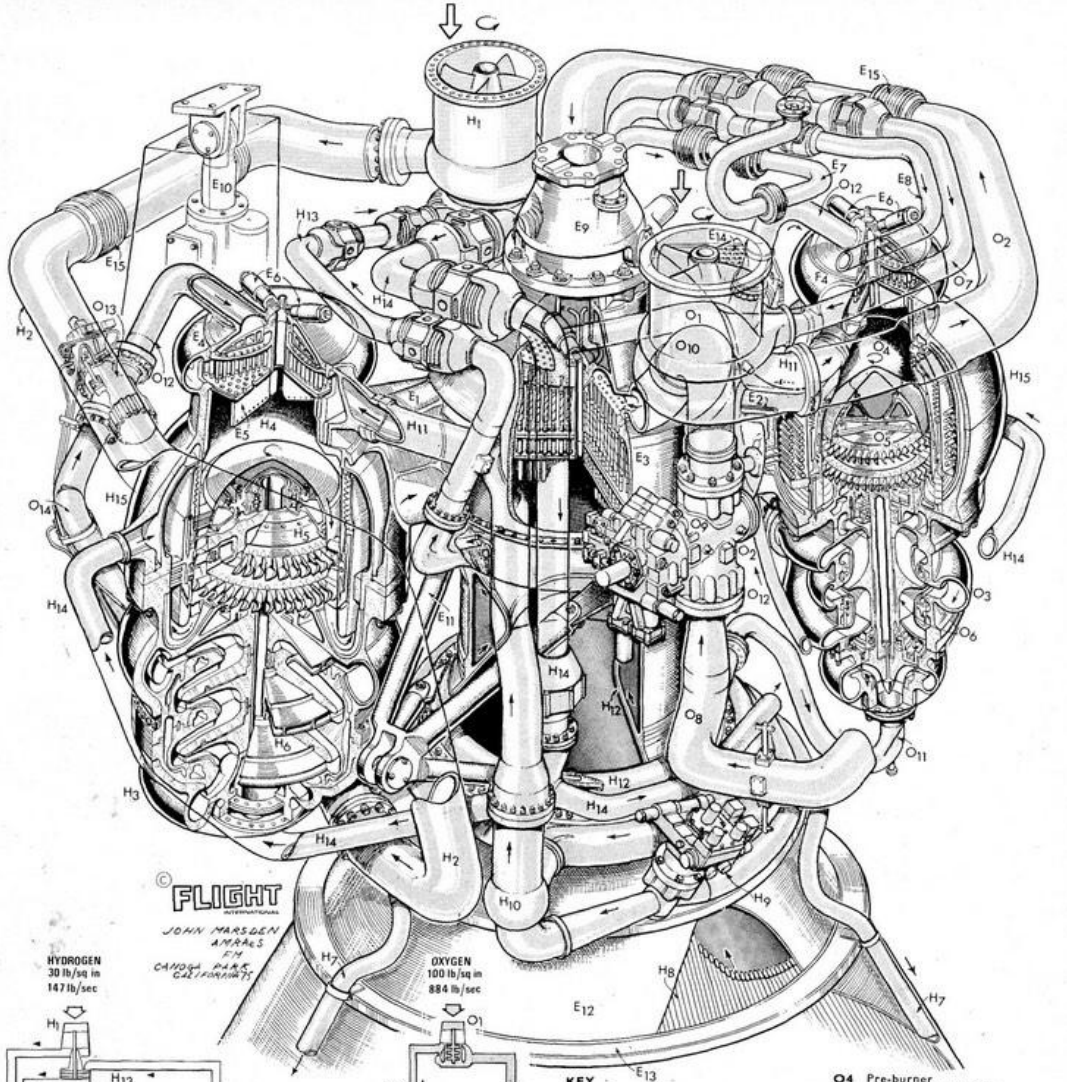
Clément Blaudeau - Université Paris-Cité & Inria (Cambium)

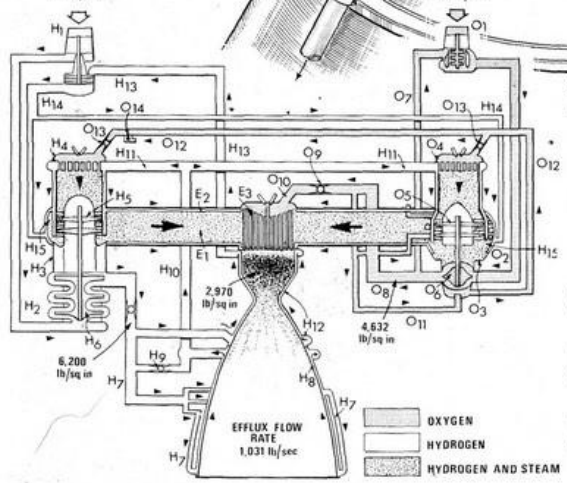
clement.blaudeau [at] inria.fr

Under the supervision of Didier Rémy (Inria) and Gabriel Radanne (Inria)

Disc
covery


PRESSURIZED CREW COMPARTMENT
FORWARD REACTION CONTROL SYSTEM
CRYOGENIC OXYGEN/HYDROGEN TANKS-FUEL CELLS (OXYGEN FOR CREW COMPARTMENT OXYGEN SUPPLY)
FLIGHT DECK
DEPLOYABLE RADIATOR PANELS
ET LIQUID OXYGEN TANK
PAYLOAD BAY DOORS
PAYLOAD BAY
MID DECK & CREW INGRESS/ EGRESS SIDE HATCH
FIXED RADIATOR PANELS
VERTICAL TAIL
ELEVON RUB PANELS
SRB SEPARATION MOTORS
RUDDER/SPEED BRAKE
SRB RECOVERY SYSTEM
ELEVONS
SPACE SHUTTLE MAIN ENGINES (3)
LIQUID HYDROGEN TANK
REMOTE MANIPULATOR SYSTEM (NOT INSTALLED FOR STS-1)
SRB PROPELLANT & CASING
BODY FLAP
OMS ENGINE (2)
RCS FUEL TANK
RCS OXIDIZER TANK
RCS ENGINES (TOTAL - 38 PRIMARY) (TOTAL - 6 VERNIER)
OMS OXIDIZER TANK
WING
OMS FUEL TANK
SRB NOZZLE
SRB HOLDDOWN POSTS (4)
SRB SEPARATION MOTORS





KEY

Hydrogen system

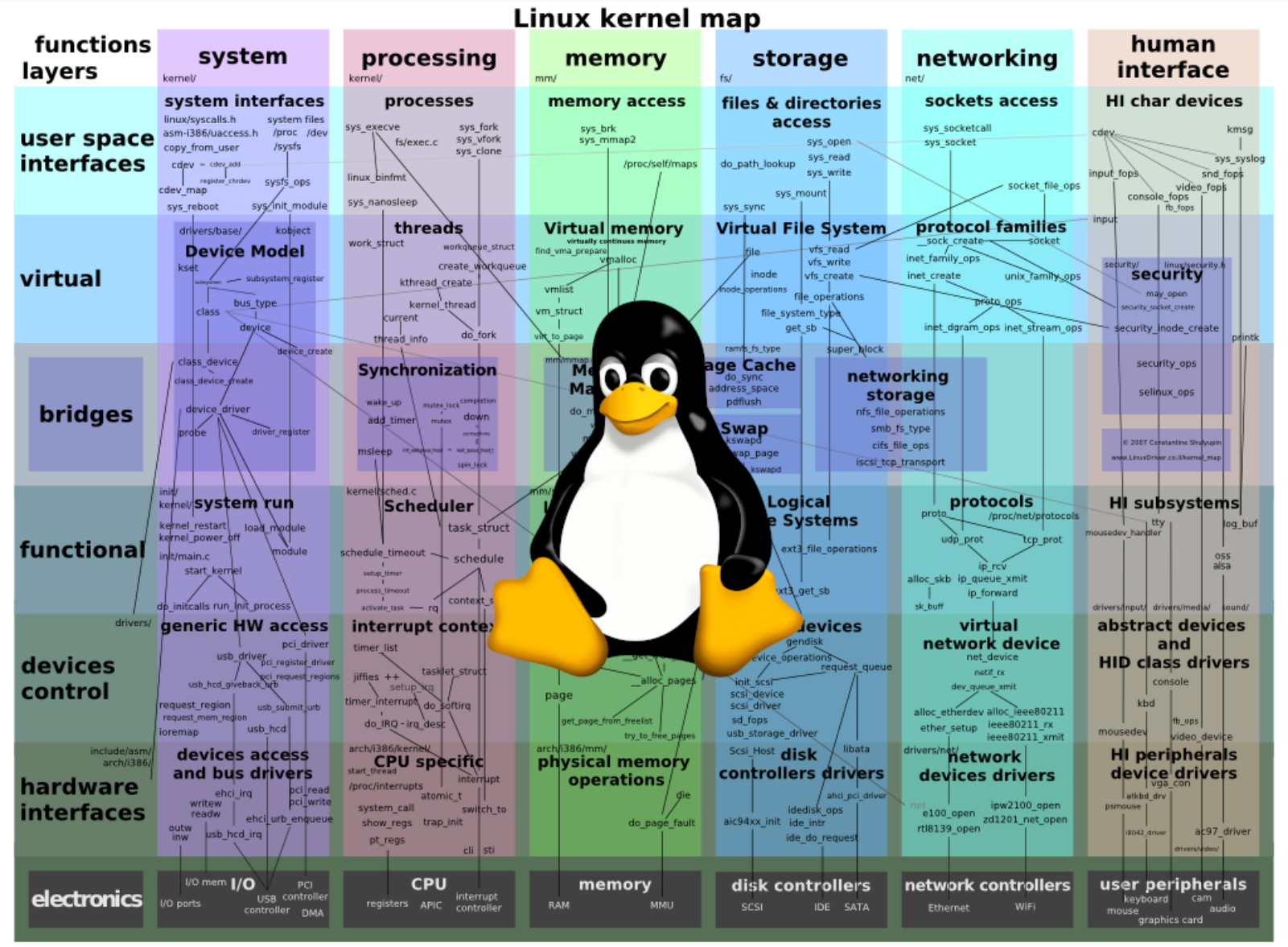
- **H1** Low-pressure turbopump
- **H2** Fuel delivery pipe from l-p pump to high-pressure turbopump
- **H3** High-pressure turbopump
- **H4** Pre-burner
- **H5** High-pressure turbine
- **H6** High-pressure centrifugal compressor
- **H7** Nozzle-coolant supply pipes
- **H8** Nozzle thrust chamber showing tapered tube assembly providing regenerative cooling circuit
- **H9** Coolant bypass control valve
- **H10** Pre-burner supply pipe
- **H11** Pre-burner supply pipe, bifurcated to supply oxygen h-p turbopump pre-burner
- **H12** Main combustion chamber manifold (regeneratively cooled)
- **H13** Delivery pipe to l-p turbopump
- **H14** Coolant delivery pipe from l-p turbine to h-p hot-gas manifold
- **H15** Heat-exchanger

Oxygen system

- **O1** Low-pressure turbopump
- **O2** Fuel delivery pipe from l-p pump to high-pressure turbopump
- **O3** High-pressure turbopump
- **O4** Pre-burner
- **O5** High-pressure turbine
- **O6** High-pressure centrifugal compressor
- **O7** Low-pressure turbine drive pipe
- **O8** High-pressure delivery pipe to combustion chamber
- **O9** Main oxidizer metering valve
- **O10** Delivery pipe to main injector
- **O11** Pre-burner turbopump supply pipe
- **O12** Pre-burner oxygen-supply pipe
- **O13** Pre-burner fuel-control valve
- **O14** Oxidizer bleed valve

Engine ancillaries

- **E1** Hot-gas manifold
- **E2** Internal cooling liner
- **E3** Main dome/injector and baffle elements (total 600)
- **E4** Pre-burner co-axial injector elements
- **E5** Baffle plates (three)
- **E6** Dual spark igniters
- **E7** Fuel bleed pipe
- **E8** Oxygen tank pressurisation pipe
- **E9** Gimbal-bearing thrust housing
- **E10** Gimbal actuator (total two)
- **E12** Gimbal-actuator lower support strut
- **E12** External insulation heat jacket
- **E13** Reinforcing "hat-bands"
- **E14** Solid-state control unit
- **E15** Flexible bellows



~30 million lines of code

Handling complex system

Handling complex system

Modularity

- Structure: gather components into *modules* with limited interactions
- Re-usability: Identify common parts reused throughout the system

Handling complex system

Modularity

- Structure: gather components into *modules* with limited interactions
- Re-usability: Identify common parts reused throughout the system

The tools of modularity

Handling complex system

Modularity

- Structure: gather components into *modules* with limited interactions
- Re-usability: Identify common parts reused throughout the system

The tools of modularity

- Interfaces
- Abstraction
- Name-spaces
- Composition

Handling complex system

Modularity

- Structure: gather components into *modules* with limited interactions
- Re-usability: Identify common parts reused throughout the system

The tools of modularity

- Interfaces
- Abstraction
- Name-spaces
- Composition

In programming languages

- Functions
- Code/Interfaces files
- Libraries
- Package-managers

Handling complex system

Modularity

- Structure: gather components into *modules* with limited interactions
- Re-usability: Identify common parts reused throughout the system

The tools of modularity

- Interfaces
- Abstraction
- Name-spaces
- Composition

In programming languages

- Functions
- Code/Interfaces files
- Libraries
- Package-managers
- OOP (C++, Java, Scala, Python, etc.)

Handling complex system

Modularity

- Structure: gather components into *modules* with limited interactions
- Re-usability: Identify common parts reused throughout the system

The tools of modularity

- Interfaces
- Abstraction
- Name-spaces
- Composition

In programming languages

- Functions
- Code/Interfaces files
- Libraries
- Package-managers
- OOP (C++, Java, Scala, Python, etc.)
- Type-classes (Haskell, Rust, etc.)

Handling complex system

Modularity

- Structure: gather components into *modules* with limited interactions
- Re-usability: Identify common parts reused throughout the system

The tools of modularity

- Interfaces
- Abstraction
- Name-spaces
- Composition

In programming languages

- Functions
- Code/Interfaces files
- Libraries
- Package-managers
- OOP (C++, Java, Scala, Python, etc.)
- Type-classes (Haskell, Rust, etc.)
- Module systems (OCaml, SML, Coq, etc.)

Plan

Plan

1. **The ML[1] module system**
   - Structure, signatures, functors
   - The signature avoidance problem

[1] *Meta-Language*, not machine learning

Plan

1. **The ML[1] module system**
   - Structure, signatures, functors
   - The signature avoidance problem
2. **M$^\omega$**
   - Type system *by translation* to F$^\omega$
   - Transparent existentials
   - OOPSLA 2024 - *Fulfilling OCaml modules with transparency*

[1] *Meta-Language*, not machine learning

Plan

1. **The ML[1] module system**
   - Structure, signatures, functors
   - The signature avoidance problem
2. **M$^\omega$**
   - Type system *by translation* to F$^\omega$
   - Transparent existentials
   - OOPSLA 2024 - *Fulfilling OCaml modules with transparency*
3. **ZipML**
   - Type system based on *paths*
   - Zipper signatures
   - POPL 2025 - *Avoiding signature avoidance in ML modules with zippers*

[1] *Meta-Language*, not machine learning

The ML module system

ML modularity

Building structures

```
module Complex =  struct
  type t = float * float (* polar *)
  let zero = (0., 0.)
  let one = (1., 0.)
  let add c c' = ...
  ...
end
```

ML modularity

Building structures

```
module Complex =  struct
  type t = float * float (* polar *)
  let zero = (0., 0.)
  let one = (1., 0.)
  let add c c' = ...
  ...
end
```

```
let c_two =
 Complex.add Complex.one Complex.one
```

```
val c_two : Complex.t
```

ML modularity

Building structures

```
module Complex =  struct
  type t = float * float (* polar *)
  let zero = (0., 0.)
  let one = (1., 0.)
  let add c c' = ...
  ...
end
```

```
let c_two =
 Complex.add Complex.one Complex.one
```

```
val c_two : Complex.t
```

```
let c_invalid =
 Complex.add (Complex.one) (-1., 0.)
```

```
val c_invalid : Complex.t
```

ML modularity

Building structures

```
module Complex =  struct
  type t = float * float (* polar *)
  let zero = (0., 0.)
  let one = (1., 0.)
  let add c c' = ...
  ...
end
```

```
let c_two =
 Complex.add Complex.one Complex.one
```

```
val c_two : Complex.t
```

```
let c_invalid =
 Complex.add (Complex.one) (-1., 0.)
```

```
val c_invalid : Complex.t
```

Defining signatures

```
module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t → t → t
  ...
end
```

ML modularity

Building structures

```
module Complex = (struct
  type t = float * float (* polar *)
  let zero = (0., 0.)
  let one = (1., 0.)
  let add c c' = ...
  ...
end : Ring )
```

```
let c_two =
 Complex.add Complex.one Complex.one
```

```
val c_two : Complex.t
```

```
let c_invalid =
 Complex.add (Complex.one) (-1., 0.)
```

```
val c_invalid : Complex.t
```

Defining signatures

```
module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t → t → t
  ...
end
```

ML modularity

Building structures

```
module Complex = (struct
  type t = float * float (* polar *)
  let zero = (0., 0.)
  let one = (1., 0.)
  let add c c' = ...
  ...
end : Ring )
```

```
let c_two =
 Complex.add Complex.one Complex.one
```

```
val c_two : Complex.t
```

```
let c_invalid =
 Complex.add (Complex.one) (-1., 0.)
```

```
(-1,0) is not of type Complex.t
```

Defining signatures

```
module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t → t → t
  ...
end
```

ML modularity

Building structures

```
module Complex = (struct
  type t = float * float (* polar *)
  let zero = (0., 0.)
  let one = (1., 0.)
  let add c c' = ...
  ...
end : Ring )
```

Defining signatures

```
module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t → t → t
  ...
end
```

Functors

```
module Polynomials =
```

ML modularity

Building structures

```
module Complex = (struct
  type t = float * float (* polar *)
  let zero = (0., 0.)
  let one = (1., 0.)
  let add c c' = ...
  ...
end : Ring )
```

Defining signatures

```
module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t → t → t
  ...
end
```

Functors

```
module Polynomials =
  functor (R: Ring) → struct
    type t = R.t list
    let zero = []
    let one = [R.one]
    let add p p' = ...
    ...
  end
```

ML modularity

Building structures

```
module Complex = (struct
  type t = float * float (* polar *)
  let zero = (0., 0.)
  let one = (1., 0.)
  let add c c' = ...
  ...
end : Ring )
```

Defining signatures

```
module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t → t → t
  ...
end
```

Functors

```
module Polynomials =
  functor (R: Ring) → struct
    type t = R.t list
    let zero = []
    let one = [R.one]
    let add p p' = ...
    ...
  end
```

Composition

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

```
module CX : sig ... end
```

ML modularity

Building structures

```
module Complex = (struct
  type t = float * float (* polar *)
  let zero = (0., 0.)
  let one = (1., 0.)
  let add c c' = ...
  ...
end : Ring )
```

Defining signatures

```
module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t → t → t
  ...
end
```

Functors

```
module Polynomials =
  functor (R: Ring) → struct
    type t = R.t list
    let zero = []
    let one = [R.one]
    let add p p' = ...
    ...
  end
```

→ Name spaces

Composition

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

```
module CX : sig ... end
```

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

```
module CXY : sig ... end
```

ML modularity

Building structures

```
module Complex = (struct
  type t = float * float (* polar *)
  let zero = (0., 0.)
  let one = (1., 0.)
  let add c c' = ...
  ...
end : Ring )
```

Defining signatures

```
module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t → t → t
  ...
end
```

Functors

```
module Polynomials =
  functor (R: Ring) → struct
    type t = R.t list
    let zero = []
    let one = [R.one]
    let add p p' = ...
    ...
  end
```

Composition

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

```
module CX : sig ... end
```

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

```
module CXY : sig ... end
```

→ Name spaces

→ Interface control

ML modularity

Building structures

```
module Complex = (struct
  type t = float * float (* polar *)
  let zero = (0., 0.)
  let one = (1., 0.)
  let add c c' = ...
  ...
end : Ring )
```

Defining signatures

```
module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t → t → t
  ...
end
```

Functors

```
module Polynomials =
  functor (R: Ring) → struct
    type t = R.t list
    let zero = []
    let one = [R.one]
    let add p p' = ...
    ...
  end
```

→ Name spaces

→ Interface control

Composition

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

```
module CX : sig ... end
```

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

```
module CXY : sig ... end
```

→ Abstraction

ML modularity

Building structures

```
module Complex = (struct
  type t = float * float (* polar *)
  let zero = (0., 0.)
  let one = (1., 0.)
  let add c c' = ...
  ...
end : Ring )
```

Defining signatures

```
module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t → t → t
  ...
end
```

Functors

```
module Polynomials =
  functor (R: Ring) → struct
    type t = R.t list
    let zero = []
    let one = [R.one]
    let add p p' = ...
    ...
  end
```

→ Name spaces

→ Interface control

Composition

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

```
module CX : sig ... end
```

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

```
module CXY : sig ... end
```

→ Abstraction

→ Parameterization

What flavor for your functor ?

Applicative

Stateless, *pure* modules, weak abstraction barrier → same applications produce equal abstract types

What flavor for your functor ?

Applicative

Stateless, *pure* modules, weak abstraction barrier ⟶ same applications produce equal abstract types

```
module Set(E: Comparable) = (... : sig type t ... end)
module S1 = Set(Complex)
module S2 = Set(Complex)
```

```
let f: S1.t ⟶ S2.t = fun x ⟶ x
```

```
val f: S1.t ⟶ S2.t
```

What flavor for your functor ?

Applicative

Stateless, *pure* modules, weak abstraction barrier ⟶ same applications produce equal abstract types

```
1 module Set(E: Comparable) = (... : sig type t ... end)
2 module S1 = Set(Complex)
3 module S2 = Set(Complex)
```

```
1 let f: S1.t ⟶ S2.t = fun x ⟶ x
```

```
val f: S1.t ⟶ S2.t
```

⟶ Especially useful for sharing between common data-structures a posteriori

⟶ Essential for modular implicits (modular type-classes)

What flavor for your functor ?

Applicative

Stateless, pure modules, weak abstraction barrier ⟶ same applications produce equal abstract types

```
1 module Set(E: Comparable) = (... : sig type t ... end)
2 module S1 = Set(Complex)
3 module S2 = Set(Complex)
```

```
1 let f: S1.t ⟶ S2.t = fun x ⟶ x
```

```
val f: S1.t ⟶ S2.t
```

⟶ Especially useful for sharing between common data-structures a posteriori

⟶ Essential for modular implicits (modular type-classes)

Generative

Stateful, effectful modules, strong abstraction barrier ⟶ each application produces new abstract types

What flavor for your functor ?

Applicative

Stateless, pure modules, weak abstraction barrier ⟶ same applications produce equal abstract types

```
module Set(E: Comparable) = (... : sig type t ... end)
module S1 = Set(Complex)
module S2 = Set(Complex)
```

```
let f: S1.t ⟶ S2.t = fun x ⟶ x
```

```
val f: S1.t ⟶ S2.t
```

⟶ Especially useful for sharing between common data-structures a posteriori

⟶ Essential for modular implicits (modular type-classes)

Generative

Stateful, effectful modules, strong abstraction barrier ⟶ each application produces new abstract types

```
module Window (T: Topbar) () = (... : sig type t ... end)
module W1 = Window(T)()
module W2 = Window(T)()
```

```
let f: W1.t ⟶ W2.t = fun x ⟶ x
```

```
W1.t is not equal to W2.t
```

What flavor for your functor ?

Applicative

Stateless, pure modules, weak abstraction barrier → same applications produce equal abstract types

```
module Set(E: Comparable) = (... : sig type t ... end)
module S1 = Set(Complex)
module S2 = Set(Complex)
```

```
let f: S1.t → S2.t = fun x → x
```

```
val f: S1.t → S2.t
```

→ Especially useful for sharing between common data-structures a posteriori

→ Essential for modular implicits (modular type-classes)

Generative

Stateful, effectful modules, strong abstraction barrier → each application produces new abstract types

```
module Window (T: Topbar) () = (... : sig type t ... end)
module W1 = Window(T)()
module W2 = Window(T)()
```

```
let f: W1.t → W2.t = fun x → x
```

```
W1.t is not equal to W2.t
```

The ML module system

The ML module system

Module expressions – `M`

The ML module system

Module expressions – `M`

- Records of values and types — `struct ... end`
- Developer-side abstraction : ascription — `(M : S)`
- Client-side abstraction : functors — `functor (Y:S) → M`, `functor () → M`
- A small calculus (projections, applications, recursion, etc.) — `M.X`, `F(X)`, `...`

The ML module system

Module expressions – `M`

- Records of values and types — `struct ... end`
- Developer-side abstraction : ascription — `(M : S)`
- Client-side abstraction : functors — `functor (Y:S) → M`, `functor () → M`
- A small calculus (projections, applications, recursion, etc.) — `M.X`, `F(X)`, `...`

Signatures – `S`

The ML module system

Module expressions – `M`

- Records of values and types — `struct ... end`
- Developer-side abstraction : ascription — `(M : S)`
- Client-side abstraction : functors — `functor (Y:S) → M`, `functor () → M`
- A small calculus (projections, applications, recursion, etc.) — `M.X`, `F(X)`, `...`

Signatures – `S`

- *Structural* signatures — `sig ... end`
- Functor signatures — `functor (Y:S) → S`, `functor () → S`
- Concrete types — `type t = (int * bool) list`
- Abstract types — `type t`

The ML module system

Module expressions – `M`

- Records of values and types — `struct ... end`
- Developer-side abstraction : ascription — `(M : S)`
- Client-side abstraction : functors — `functor (Y:S) → M`, `functor () → M`
- A small calculus (projections, applications, recursion, etc.) — `M.X`, `F(X)`, ...

Signatures – `S`

- *Structural* signatures — `sig ... end`
- Functor signatures — `functor (Y:S) → S`, `functor () → S`
- Concrete types — `type t = (int * bool) list`
- Abstract types — `type t`

→ ML is two languages in one: modules and core

→ modules: explicit, rich types ↔ core: implicit, simpler types

→ simple type-erasing semantics (records and functions)

A rich history of ML-modules

A rich history of ML-modules

The beginnings (early 90's)

Early SML versions, existential types, strong sums

A rich history of ML-modules

The beginnings (early 90's)

Early SML versions, existential types, strong sums

The golden age (late 90's, 2000's)

OCaml Applicative functors, higher-order functors, 1st-class modules, recursive modules, Moscow ML

A rich history of ML-modules

The beginnings (early 90's)

Early SML versions, existential types, strong sums

The golden age (late 90's, 2000's)

OCaml Applicative functors, higher-order functors, 1st-class modules, recursive modules, Moscow ML

The break-away (2010's)

Theoretical advances: *F-ing* modules, 1ML, MixML

A rich history of ML-modules

The beginnings (early 90's)

Early SML versions, existential types, strong sums

The golden age (late 90's, 2000's)

OCaml Applicative functors, higher-order functors, 1st-class modules, recursive modules, Moscow ML

The break-away (2010's)

Theoretical advances: *F-ing* modules, 1ML, MixML

Today (2020's)

OCaml's approach has proven successful

A rich history of ML-modules

The beginnings (early 90's)

Early SML versions, existential types, strong sums

The golden age (late 90's, 2000's)

OCaml Applicative functors, higher-order functors, 1st-class modules, recursive modules, Moscow ML

The break-away (2010's)

Theoretical advances: *F-ing* modules, 1ML, MixML

Today (2020's)

OCaml's approach has proven successful, but its spec is out of date

A rich history of ML-modules

The beginnings (early 90's)

Early SML versions, existential types, strong sums

The golden age (late 90's, 2000's)

OCaml Applicative functors, higher-order functors, 1st-class modules, recursive modules, Moscow ML

The break-away (2010's)

Theoretical advances: *F-ing* modules, 1ML, MixML

Today (2020's)

OCaml's approach has proven successful, but its spec is out of date : edge cases, un-documented heuristics, missing features

A rich history of ML-modules

The beginnings (early 90's)

Early SML versions, existential types, strong sums

The golden age (late 90's, 2000's)

OCaml Applicative functors, higher-order functors, 1st-class modules, recursive modules, Moscow ML

The break-away (2010's)

Theoretical advances: *F-ing* modules, 1ML, MixML

Today (2020's)

OCaml's approach has proven successful, but its spec is out of date : edge cases, un-documented heuristics, missing features

→ Explore the design space of type-systems for ML modules, with three goal: *explainability*, *usability* (error-messages, performance, etc.) and *retrofit*

A perfectible system

Robustness

Semantically equivalent transformations should not break type-checking

A perfectible system

Robustness

Semantically equivalent transformations should not break type-checking

Typing is not stable under β-reduction

```
1 module X = struct type t end
2 module M = F(X)
```

```
module M : sig ... end ✓
```

A perfectible system

Robustness

Semantically equivalent transformations should not break type-checking

Typing is not stable under β-reduction

```
1 module X = struct type t end
2 module M = F(X)
```

```
module M : sig ... end ✓
```

```
1 (* replaced X by its definition *)
2 module M = F(struct type t end)
```

```
The parameter cannot be
eliminated in the result type.
```

A perfectible system

Robustness

Semantically equivalent transformations should not break type-checking

Typing is not stable under β-reduction

```
1 module X = struct type t end
2 module M = F(X)
```

```
module M : sig ... end ✓
```

```
1 (* replaced X by its definition *)
2 module M = F(struct type t end)
```

```
The parameter cannot be
eliminated in the result type.
```

Typing is not stable under β-expansion

```
1 let f : G(F(X)).t → G(F(X)).t
2   = fun x → x
```

```
val f : G(F(X)).t → G(F(X)).t
```

A perfectible system

Robustness

Semantically equivalent transformations should not break type-checking

Typing is not stable under β-reduction

```
1 module X = struct type t end
2 module M = F(X)
```

```
module M : sig ... end ✓
```

```
1 (* replaced X by its definition *)
2 module M = F(struct type t end)
```

```
The parameter cannot be
eliminated in the result type.
```

Typing is not stable under β-expansion

```
1 let f : G(F(X)).t → G(F(X)).t
2   = fun x → x
```

```
val f : G(F(X)).t → G(F(X)).t
```

```
1 module FX = F(X)
2 let f : G(F(X)).t → G(FX).t
3 = fun x → x
```

```
G(F(X)).t and G(FX).t are not equal
```

A perfectible system

Robustness

Semantically equivalent transformations should not break type-checking

Typing is not stable under β-reduction → Signature avoidance

```
module X = struct type t end
module M = F(X)
```

```
module M : sig ... end ✓
```

```
(* replaced X by its definition *)
module M = F(struct type t end)
```

```
The parameter cannot be
eliminated in the result type.
```

Typing is not stable under β-expansion → Module identities

```
let f : G(F(X)).t → G(F(X)).t
  = fun x → x
```

```
val f : G(F(X)).t → G(F(X)).t
```

```
module FX = F(X)
let f : G(F(X)).t → G(FX).t
= fun x → x
```

```
G(F(X)).t and G(FX).t are not equal
```

The signature avoidance problem

abstract type fields + signatures dependencies + fields getting out of scope

The signature avoidance problem

abstract type fields + signatures dependencies + fields getting out of scope

Type-checking fail (visible error)

```
module N = struct
  type t = A of int | B of bool
  module X = struct let x = (A 1) end
end
module M = N.X
```

```
module M : sig val x : N.t end
```

The signature avoidance problem

abstract type fields + signatures dependencies + fields getting out of scope

Type-checking fail (visible error)

```
1 module N = struct
2   type t = A of int | B of bool
3   module X = struct let x = (A 1) end
4 end
5 module M = N.X
```

```
module M : sig val x : N.t end
```

```
1 (* replaced N by its definition *)
2 module M = (struct
3   type t = A of int | B of bool
4   module X = struct let x = (A 1) end
5 end ).X
```

```
The value x has no valid type
if t is hidden.
```

The signature avoidance problem

abstract type fields + signatures dependencies + fields getting out of scope

Type-checking fail (visible error)

```
module N = struct
  type t = A of int | B of bool
  module X = struct let x = (A 1) end
end
module M = N.X
```

```
module M : sig val x : N.t end
```

```
(* replaced N by its definition *)
module M = (struct
  type t = A of int | B of bool
  module X = struct let x = (A 1) end
end ).X
```

```
The value x has no valid type
if t is hidden.
```

Over-abstraction (silent error!)

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

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

The signature avoidance problem

abstract type fields + signatures dependencies + fields getting out of scope

Type-checking fail (visible error)

```
1 module N = struct
2   type t = A of int | B of bool
3   module X = struct let x = (A 1) end
4 end
5 module M = N.X
```

```
module M : sig val x : N.t end
```

```
1 (* replaced N by its definition *)
2 module M = (struct
3   type t = A of int | B of bool
4   module X = struct let x = (A 1) end
5 end ).X
```

```
The value x has no valid type
if t is hidden.
```

Over-abstraction (silent error!)

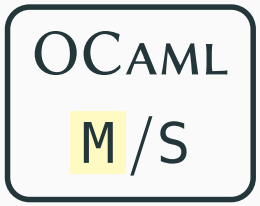
```
1 module N = struct
2   type t
3   module X = struct type u = t
4                     type v = t end
5 end
6 module M = N.X
```

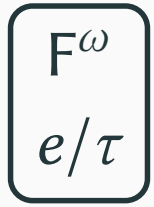
```
module M : sig type u = N.t
               type v = N.t end
```

```
1 (* replaced N by its definition *)
2 module M = (struct
3   type t
4   module X = struct type u = t
5                     type v = t  end
6 end).X
```

```
module M : sig type u
               type v  end
```

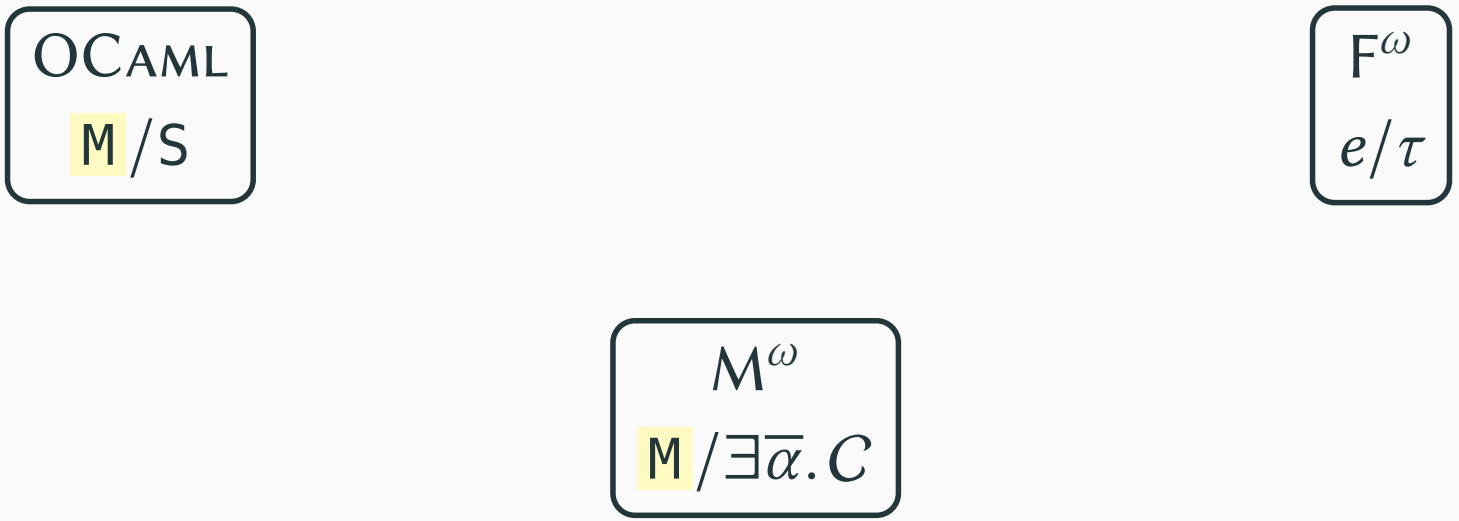
The big picture



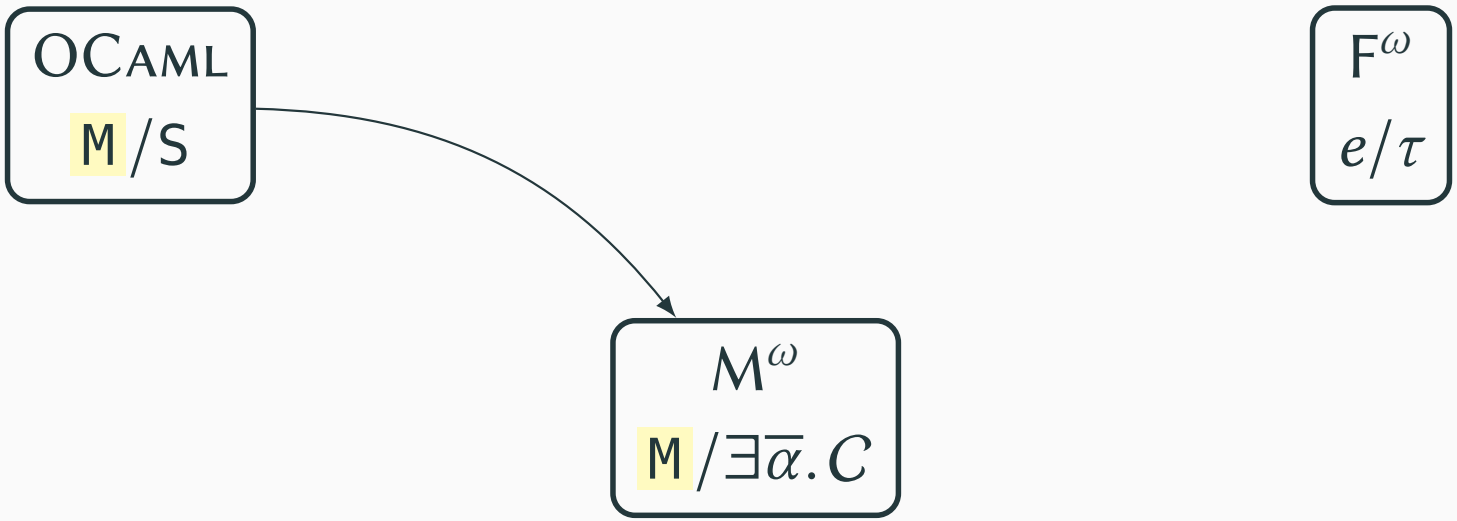


The big picture

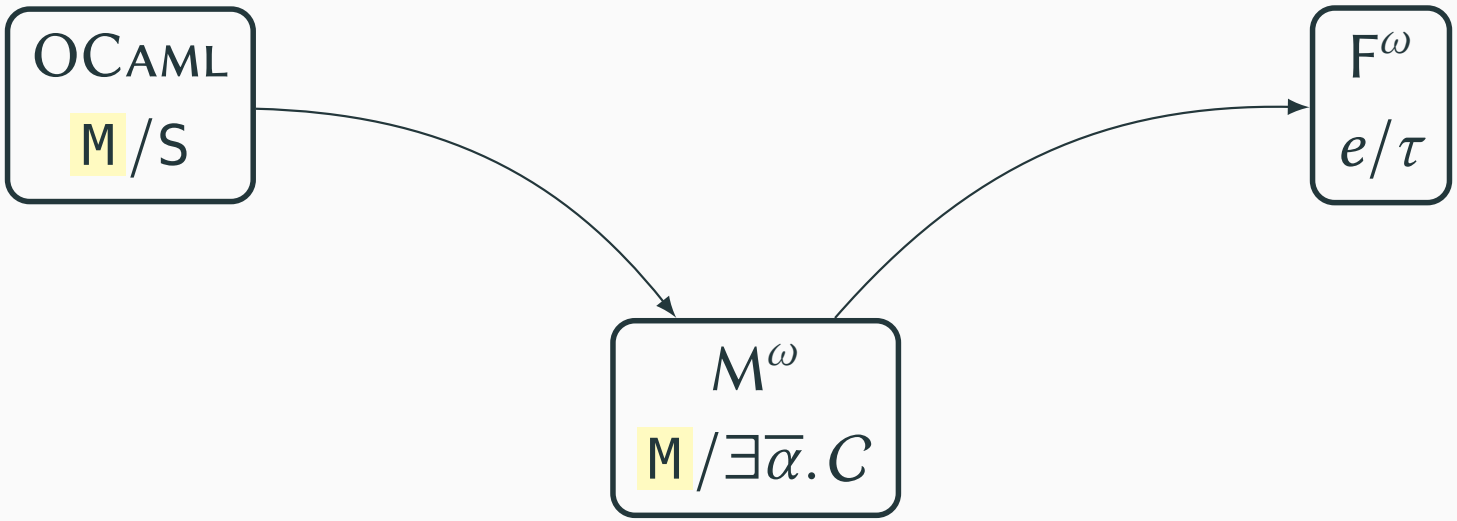
The big picture



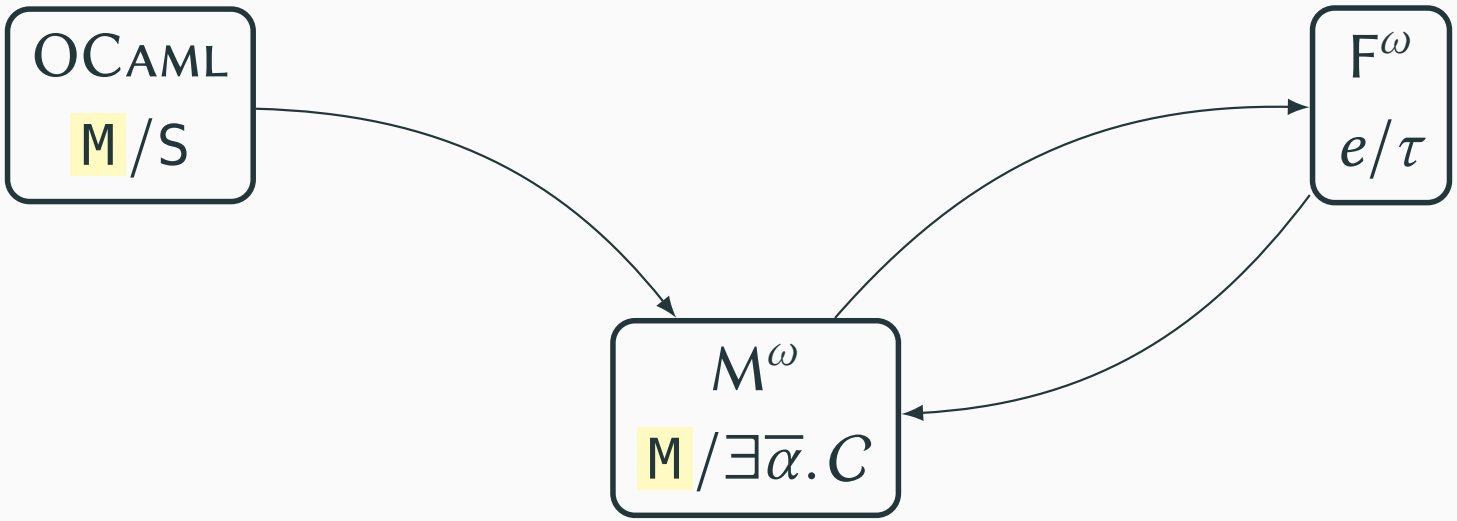
The big picture



The big picture



The big picture



Abstract types have existential type...

Source signature (reference)

- module M
  - module X
    - type t
    - val $x : t$
    - type $u = \text{int} \times t$
  - module F $(Y : \text{sig type } t \text{ end})$
    - type t
    - type $u = Y.t$
  - type $t_1 = F(X).t$
  - type $t_2 = F(X).t$

M^ω signature

- module M
  - module X
    - type t
    - val $x : t$
    - type $u = \text{int} \times t$
  - module F $(Y : \text{sig type } t \text{ end})$
    - type t
    - type $u = Y.t$
  - type $t_1 = F(X).t$
  - type $t_2 = F(X).t$

Abstract types have existential type...

Source signature (reference)

- module M
  - module X
    - type t
    - val $x : t$
    - type $u = \text{int} \times t$
  - module F $(Y : \text{sig type } t \text{ end})$
    - type t
    - type $u = Y.t$
  - type $t_1 = F(X).t$
  - type $t_2 = F(X).t$

M^ω signature

- module M
  - module X
    - type t
    - val $x : t$
    - type $u = \text{int} \times t$
  - module F $(Y : \text{sig type } t \text{ end})$
    - type t
    - type $u = Y.t$
  - type $t_1 = F(X).t$
  - type $t_2 = F(X).t$

Abstract types have existential type...

Source signature (reference)

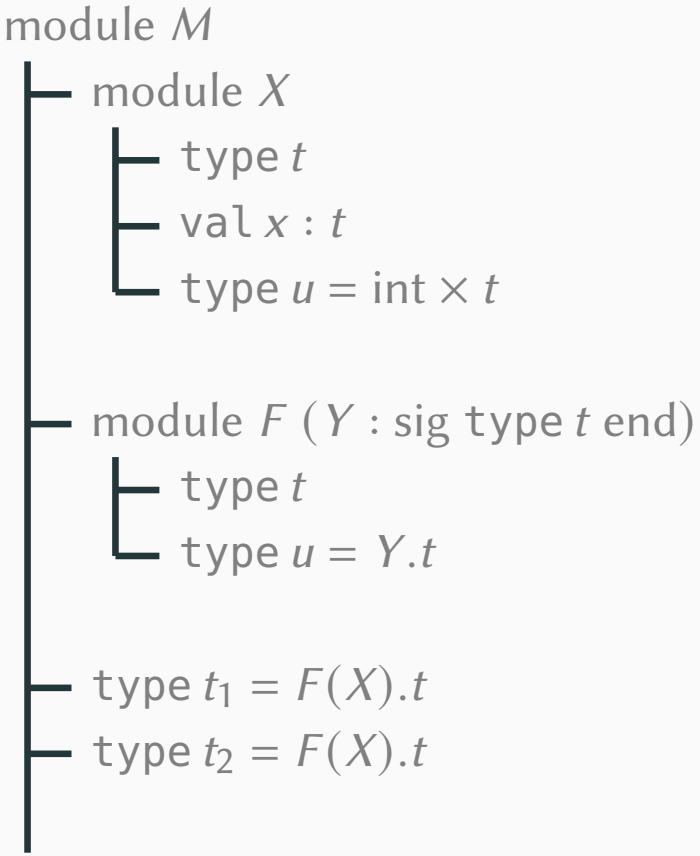
- module M
  - module X
    - type t
    - val $x : t$
    - type $u = \text{int} \times t$
  - module F $(Y : \text{sig type } t \text{ end})$
    - type t
    - type $u = Y.t$
  - type $t_1 = F(X).t$
  - type $t_2 = F(X).t$

M^ω signature

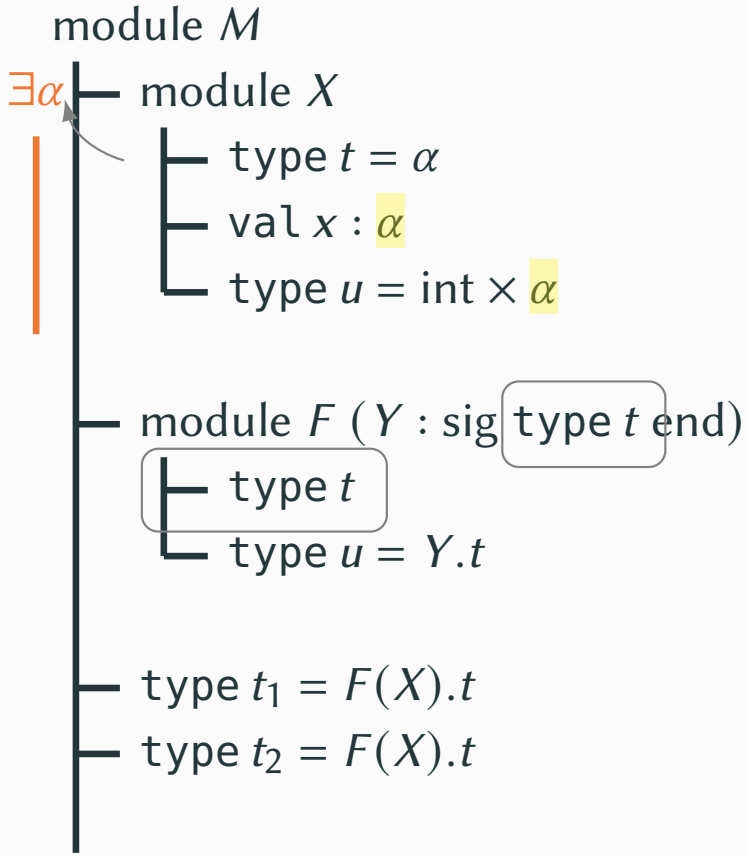
- module M
  - module X
  - $\exists\alpha$
    - type $t = \alpha$
    - val $x : t$
    - type $u = \text{int} \times t$
  - module F $(Y : \text{sig type } t \text{ end})$
    - type t
    - type $u = Y.t$
  - type $t_1 = F(X).t$
  - type $t_2 = F(X).t$

Abstract types have existential type...

Source signature (reference)

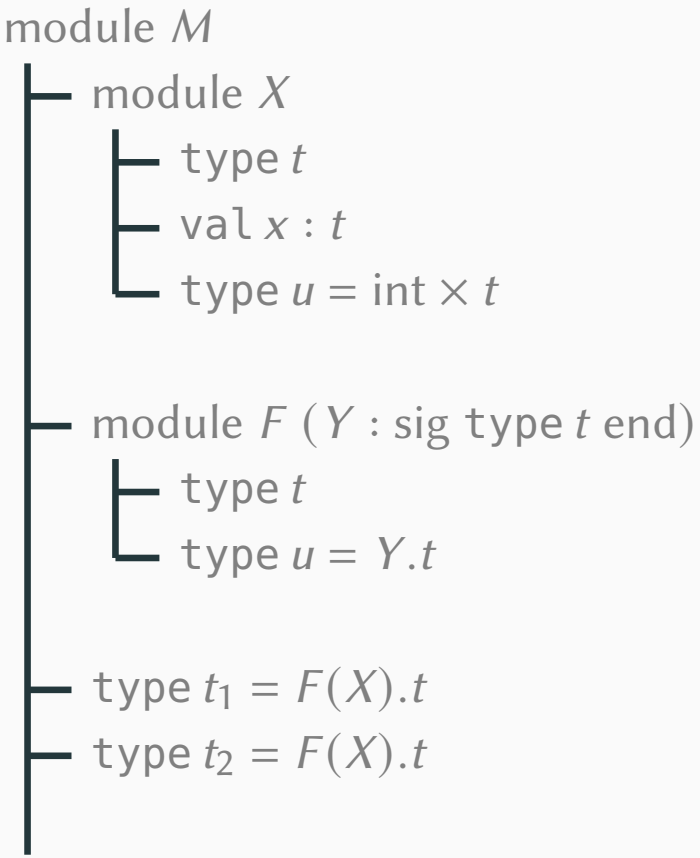


M^{ω} signature

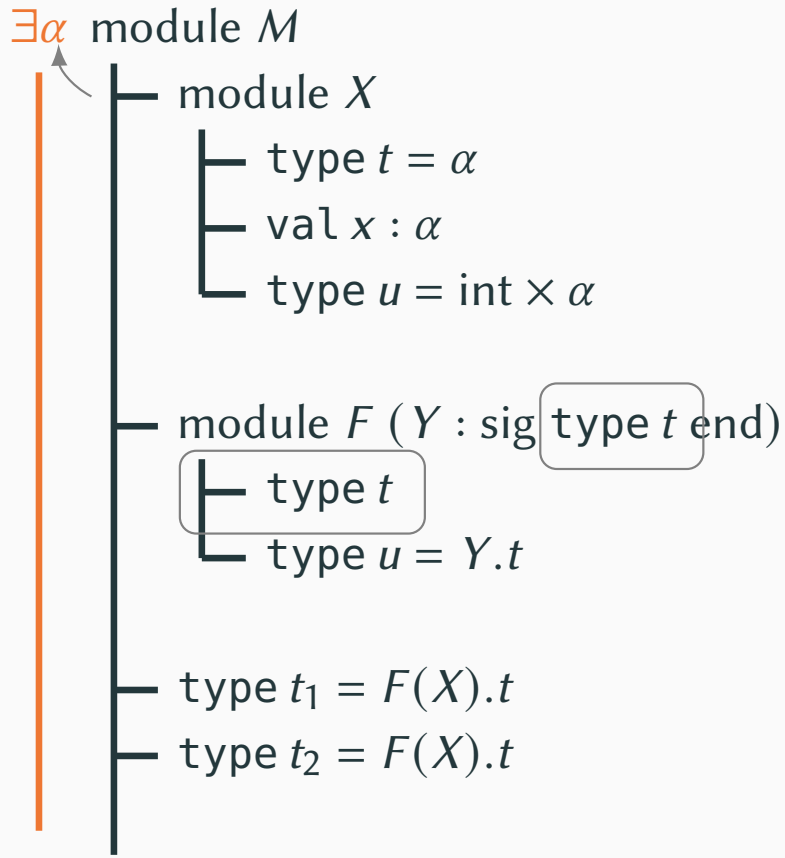


Abstract types have existential type...

Source signature (reference)

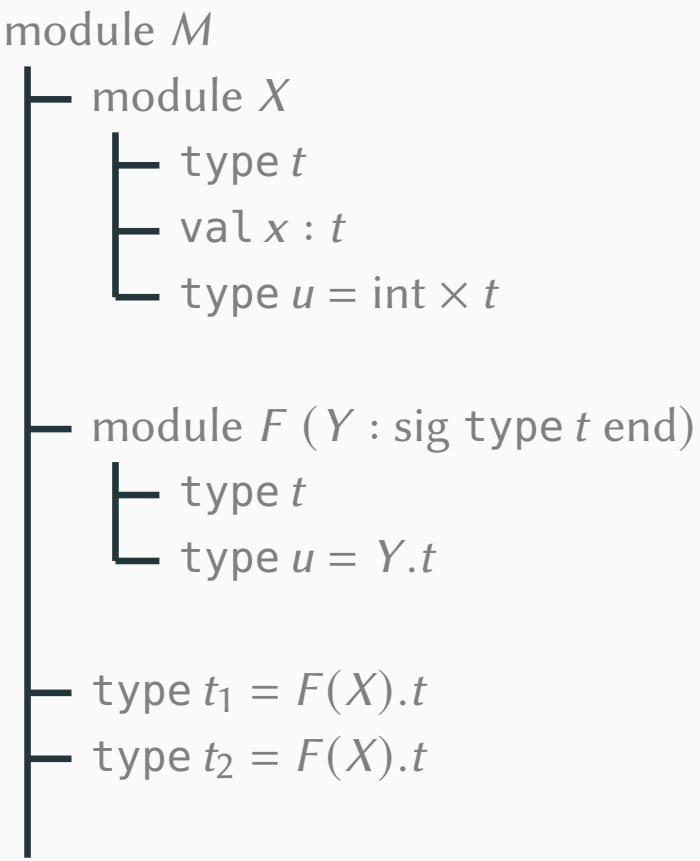


M^{ω} signature

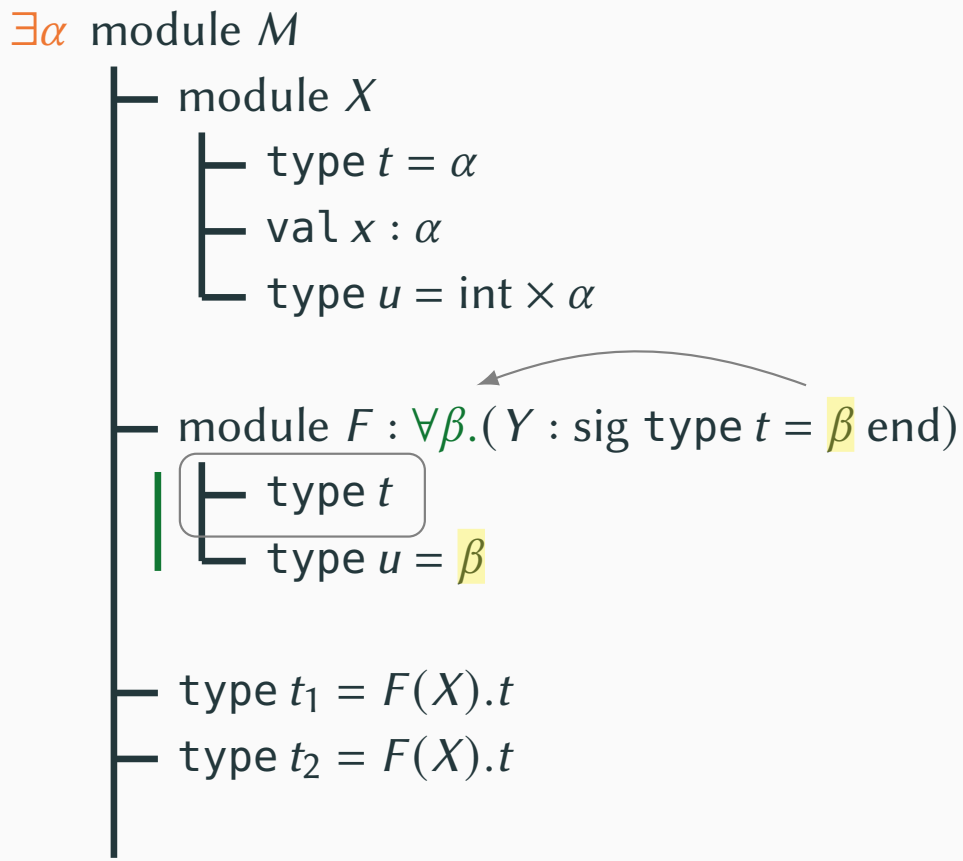


Abstract types have existential type...

Source signature (reference)



M^{ω} signature



Abstract types have existential type...

Source signature (reference)

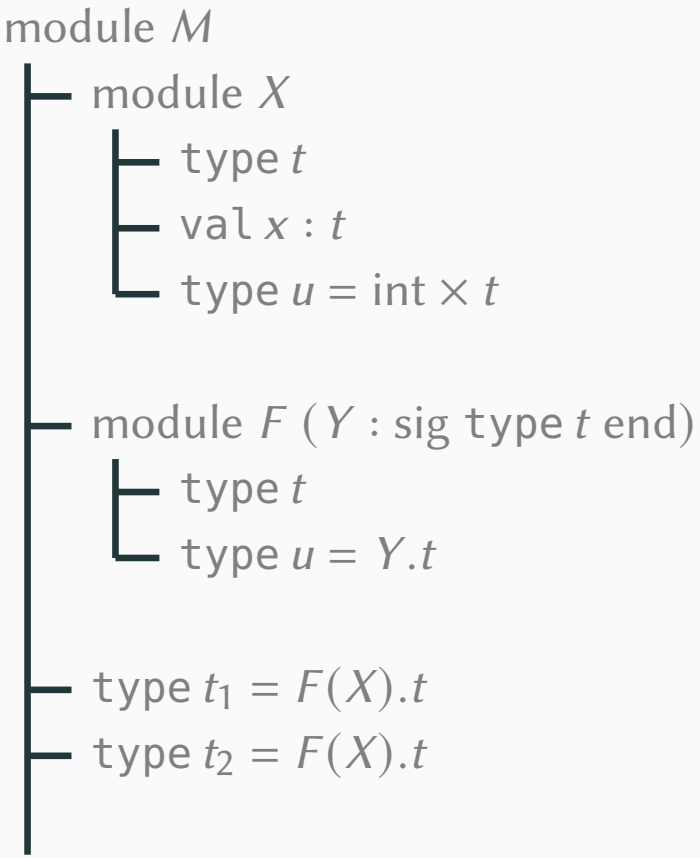
- module M
  - module X
    - type t
    - val $x : t$
    - type $u = \text{int} \times t$
  - module F $(Y : \text{sig type } t \text{ end})$
    - type t
    - type $u = Y.t$
  - type $t_1 = F(X).t$
  - type $t_2 = F(X).t$

M^ω signature

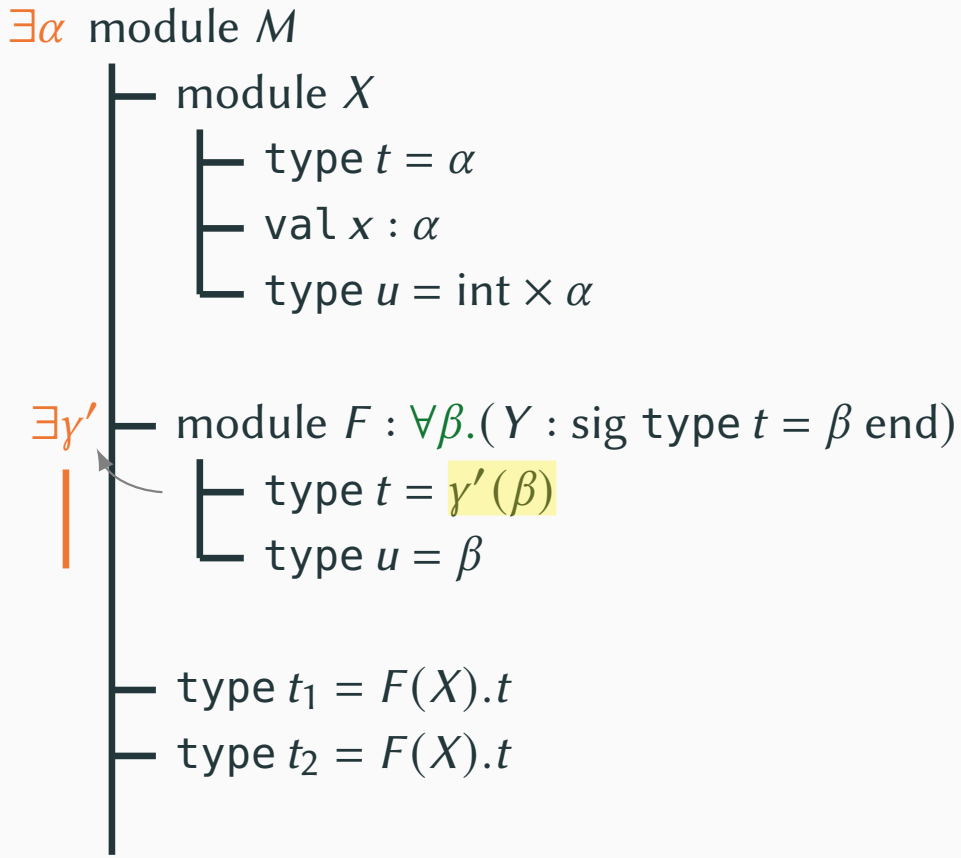
- $\exists\alpha$ module M
  - module X
    - type $t = \alpha$
    - val $x : \alpha$
    - type $u = \text{int} \times \alpha$
  - module $F : \forall\beta.(Y : \text{sig type } t = \beta \text{ end})$
    - $\exists\gamma$ type $t = \gamma$
    - type $u = \beta$
  - type $t_1 = F(X).t$
  - type $t_2 = F(X).t$

Abstract types have existential type...

Source signature (reference)

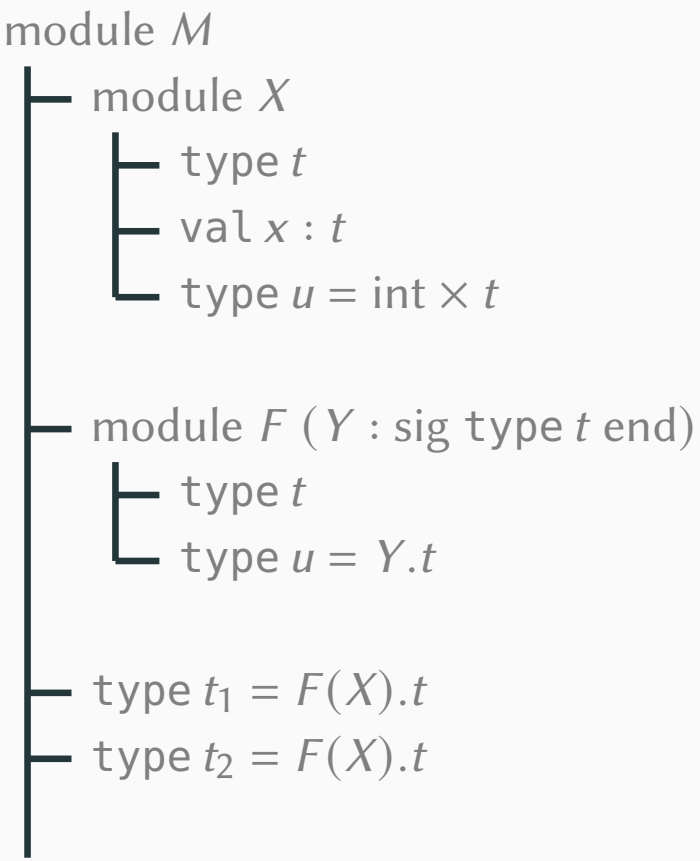


M^ω signature

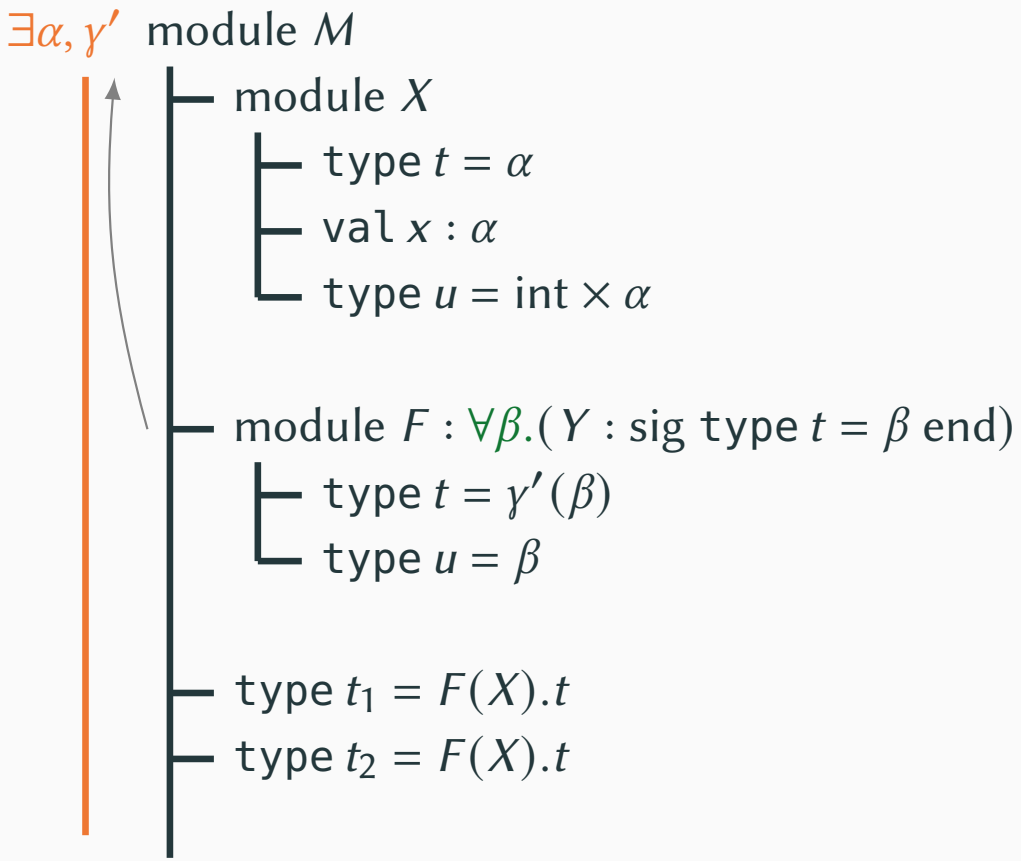


Abstract types have existential type...

Source signature (reference)



M^ω signature



Abstract types have existential type...

Source signature (reference)

- module M
  - module X
    - type t
    - val $x : t$
    - type $u = \text{int} \times t$
  - module F $(Y : \text{sig type } t \text{ end})$
    - type t
    - type $u = Y.t$
  - type $t_1 = F(X).t$
  - type $t_2 = F(X).t$

M^ω signature

$\exists \alpha, \gamma'$ 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 $u = \beta$
- type $t_1 = \gamma'(\alpha)$
- type $t_2 = F(X).t$

Abstract types have existential type...

Source signature (reference)

- module M
  - module X
    - type t
    - val $x : t$
    - type $u = \text{int} \times t$
  - module F $(Y : \text{sig type } t \text{ end})$
    - type t
    - type $u = Y.t$
  - type $t_1 = F(X).t$
  - type $t_2 = F(X).t$

M^ω signature

$\exists \alpha, \gamma'$ 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 $u = \beta$
- type $t_1 = \gamma'(\alpha)$
- type $t_2 = \gamma'(\alpha)$

F^ω overview / M^ω overview

F^ω - with primitive records and existential types

$$\tau := \alpha \mid \tau \to \tau \mid \{\overline{\ell : \tau}\} \mid \forall\alpha.\,\tau \mid \exists\alpha.\tau \mid \lambda\alpha.\,\tau \mid () \qquad \text{(types)}$$

$$e := x \mid \lambda(x : \tau).e \mid e\,e \mid \Lambda\alpha.\,e \mid e\,\tau \mid e \mathbin{@} e \mid \{\overline{\ell = e}\} \mid e.\ell$$

$$\mid \text{pack}\,\langle\tau, e\rangle\,\text{as}\,\exists\alpha.\tau \mid \text{unpack}\,\langle\alpha, x\rangle = e\,\text{in}\,e \mid () \qquad \text{(terms)}$$

F^ω overview / M^ω overview

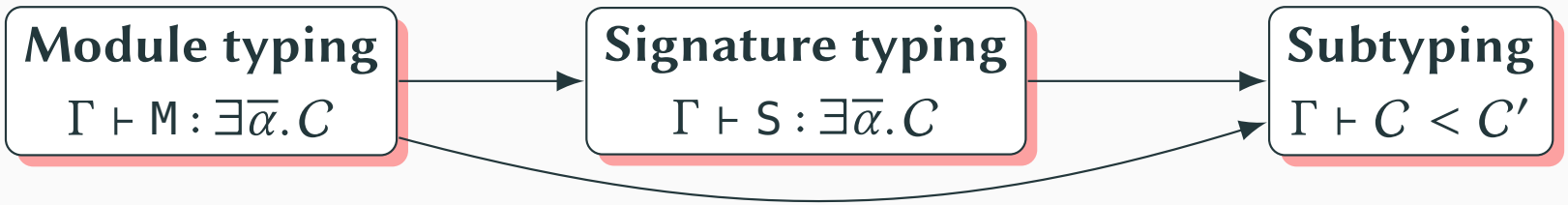
F^ω - with primitive records and existential types

$$\tau := \alpha \mid \tau \to \tau \mid \{\overline{\ell : \tau}\} \mid \forall\alpha.\tau \mid \exists\alpha.\tau \mid \lambda\alpha.\tau \mid () \qquad \text{(types)}$$

$$e := x \mid \lambda(x : \tau).e \mid e\,e \mid \Lambda\alpha.\,e \mid e\,\tau \mid e \mathbin{@} e \mid \{\overline{\ell = e}\} \mid e.\ell$$

$$\mid \text{pack}\,\langle\tau, e\rangle\,\text{as}\,\exists\alpha.\tau \mid \text{unpack}\,\langle\alpha, x\rangle = e\,\text{in}\,e \mid () \qquad \text{(terms)}$$

Momega



F^ω overview / M^ω overview

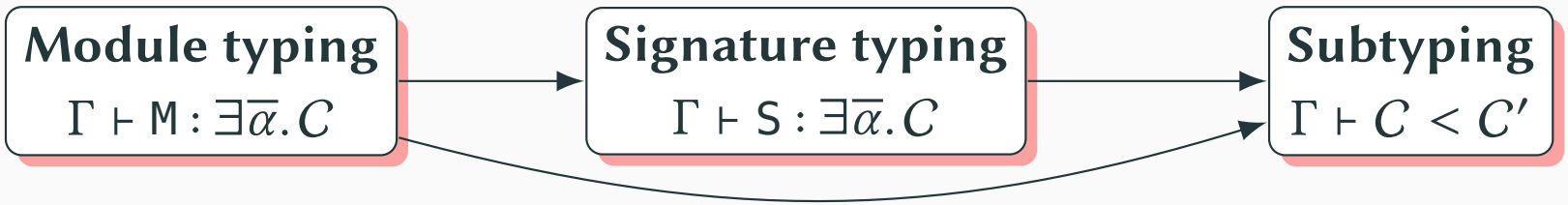
F^ω - with primitive records and existential types

$$\tau := \alpha \mid \tau \to \tau \mid \{\overline{\ell : \tau}\} \mid \forall\alpha.\tau \mid \exists\alpha.\tau \mid \lambda\alpha.\tau \mid () \qquad \text{(types)}$$

$$e := x \mid \lambda(x:\tau).e \mid e\,e \mid \Lambda\alpha.e \mid e\,\tau \mid e \mathbin{@} e \mid \{\overline{\ell = e}\} \mid e.\ell$$

$$\mid \text{pack}\,\langle\tau, e\rangle\,\text{as}\,\exists\alpha.\tau \mid \text{unpack}\,\langle\alpha, x\rangle = e\,\text{in}\,e \mid () \qquad \text{(terms)}$$

Momega



$\to$ M^ω signatures $\exists\overline{\alpha}.C$ are just syntactic sugar for F^ω types

F^ω overview / M^ω overview

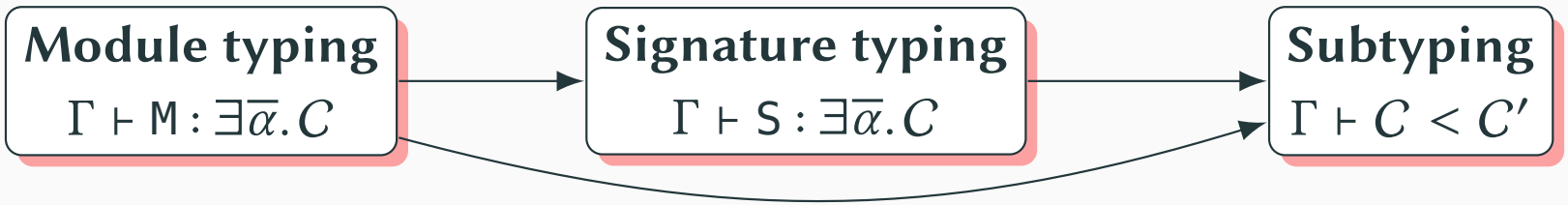
F^ω - with primitive records and existential types

$$\tau := \alpha \mid \tau \to \tau \mid \{\overline{\ell : \tau}\} \mid \forall\alpha.\tau \mid \exists\alpha.\tau \mid \lambda\alpha.\tau \mid () \qquad \text{(types)}$$

$$e := x \mid \lambda(x:\tau).e \mid e\,e \mid \Lambda\alpha.e \mid e\,\tau \mid e \mathbin{@} e \mid \{\overline{\ell = e}\} \mid e.\ell$$

$$\mid \text{pack}\,\langle\tau, e\rangle\,\text{as}\,\exists\alpha.\tau \mid \text{unpack}\,\langle\alpha, x\rangle = e\,\text{in}\,e \mid () \qquad \text{(terms)}$$

Momega



→ M^ω signatures $\exists\overline{\alpha}.C$ are just syntactic sugar for F^ω types

→ Soundness is shown by building a well-typed *evidence term* in F^ω

$$\Gamma \vdash \mathsf{M} : \exists\overline{\alpha}.C \rightsquigarrow e \qquad\qquad \Gamma \vdash C <: C' \rightsquigarrow f$$

Example of encoding into F^{ω}

Source code

```
module type S =
  sig type t val x : t end
```

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

Example of encoding into F^{ω}

Source code

```
module type S =
  sig type t val x : t end
```

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

M^{ω} Signature

$\exists \alpha, \beta.$ **module** M : **sig**
  module X1 : **sig**
    type t = α
    val x : α
  end
  module X2 : **sig**
    type t = β
    val x : β
  end
end

Example of encoding into F^{ω}

Source code

```
module type S =
  sig type t val x : t end
```

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

M^{ω} Signature

```
∃α, β. module M : sig
  module X1 : sig
    type t = α
    val x : α
  end
  module X2 : sig
    type t = β
    val x : β
  end
end
```

F^{ω} type

$$\exists \alpha, \beta. \left\{ \begin{array}{l} \ell_{X_1} : \left\{ \begin{array}{l} \ell_t : \langle\langle \alpha \rangle\rangle \\ \ell_x : \alpha \end{array} \right\} \\ \ell_{X_2} : \left\{ \begin{array}{l} \ell_t : \langle\langle \beta \rangle\rangle \\ \ell_x : \beta \end{array} \right\} \end{array} \right\}$$

where

$$\langle\langle \tau \rangle\rangle \triangleq \forall\, \varphi\, .\, \varphi\, \tau \to \varphi\, \tau$$

Example of encoding into F^{ω}

Source code

```
module type S =
  sig type t val x : t end
```

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

M^{ω} Signature

```
∃α, β. module M : sig
  module X1 : sig
    type t = α
    val x : α
  end
  module X2 : sig
    type t = β
    val x : β
  end
end
```

F^{ω} type

$$\exists \alpha, \beta. \left\{ \begin{array}{l} \ell_{X_1} : \left\{ \begin{array}{l} \ell_t : \langle\langle \alpha \rangle\rangle \\ \ell_x : \alpha \end{array} \right\} \\ \ell_{X_2} : \left\{ \begin{array}{l} \ell_t : \langle\langle \beta \rangle\rangle \\ \ell_x : \beta \end{array} \right\} \end{array} \right\}$$

where

$$\langle\langle \tau \rangle\rangle \triangleq \forall\, \varphi\, .\, \varphi\, \tau \to \varphi\, \tau$$

Encoded module

$$e =$$

Example of encoding into F^{ω}

Source code

```
module type S =
  sig type t val x : t end
```

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

M^{ω} Signature

```
∃α, β. module M : sig
  module X1 : sig
    type t = α
    val x : α
  end
  module X2 : sig
    type t = β
    val x : β
  end
end
```

F^{ω} type

$$\exists \alpha, \beta. \left\{ \begin{array}{l} \ell_{X_1} : \left\{ \begin{array}{l} \ell_t : \langle\langle \alpha \rangle\rangle \\ \ell_x : \alpha \end{array} \right\} \\ \ell_{X_2} : \left\{ \begin{array}{l} \ell_t : \langle\langle \beta \rangle\rangle \\ \ell_x : \beta \end{array} \right\} \end{array} \right\}$$

where

$$\langle\langle \tau \rangle\rangle \triangleq \forall\, \varphi\, .\, \varphi\, \tau \to \varphi\, \tau$$

Encoded module

$e =$ $\{\ell_t = \langle\langle \mathsf{int} \rangle\rangle, \ell_x = 42\}$

Example of encoding into F^{ω}

Source code

```
module type S =
  sig type t val x : t end
```

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

M^{ω} Signature

```
∃α, β. module M : sig
  module X1 : sig
    type t = α
    val x : α
  end
  module X2 : sig
    type t = β
    val x : β
  end
end
```

F^{ω} type

$$\exists \alpha, \beta. \left\{ \begin{array}{l} \ell_{X_1} : \left\{ \begin{array}{l} \ell_t : \langle\langle \alpha \rangle\rangle \\ \ell_x : \alpha \end{array} \right\} \\ \ell_{X_2} : \left\{ \begin{array}{l} \ell_t : \langle\langle \beta \rangle\rangle \\ \ell_x : \beta \end{array} \right\} \end{array} \right\}$$

where

$$\langle\langle \tau \rangle\rangle \triangleq \forall\, \varphi\, .\, \varphi\, \tau \to \varphi\, \tau$$

Encoded module

$$e = \quad \mathsf{pack}\langle \mathtt{int}, \{\ell_t = \langle\langle \mathtt{int} \rangle\rangle, \ell_x = 42\} \rangle$$

Example of encoding into F^{ω}

Source code

```
module type S =
  sig type t val x : t end
```

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

M^{ω} Signature

```
∃α, β. module M : sig
  module X1 : sig
    type t = α
    val x : α
  end
  module X2 : sig
    type t = β
    val x : β
  end
end
```

F^{ω} type

$$\exists \alpha, \beta. \left\{ \begin{array}{l} \ell_{X_1} : \left\{ \begin{array}{l} \ell_t : \langle\langle \alpha \rangle\rangle \\ \ell_x : \alpha \end{array} \right\} \\ \ell_{X_2} : \left\{ \begin{array}{l} \ell_t : \langle\langle \beta \rangle\rangle \\ \ell_x : \beta \end{array} \right\} \end{array} \right\}$$

where

$$\langle\langle \tau \rangle\rangle \triangleq \forall\, \varphi\, .\, \varphi\, \tau \to \varphi\, \tau$$

Encoded module

$$e = \quad \mathsf{pack}\langle \mathsf{int}, \{\ell_t = \langle\langle \mathsf{int} \rangle\rangle, \ell_x = 42\} \rangle$$
$$\{\ell_t = \langle\langle \, ? \, \times \mathsf{bool} \rangle\rangle, \ell_x = (\ ?\ , \bot)\}$$

Example of encoding into F^{ω}

Source code

```
module type S =
  sig type t val x : t end
```

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

M^{ω} Signature

```
∃α, β. module M : sig
  module X1 : sig
    type t = α
    val x : α
  end
  module X2 : sig
    type t = β
    val x : β
  end
end
```

F^{ω} type

$$\exists \alpha, \beta. \left\{ \begin{array}{l} \ell_{X_1} : \left\{ \begin{array}{l} \ell_t : \langle\langle \alpha \rangle\rangle \\ \ell_x : \alpha \end{array} \right\} \\ \ell_{X_2} : \left\{ \begin{array}{l} \ell_t : \langle\langle \beta \rangle\rangle \\ \ell_x : \beta \end{array} \right\} \end{array} \right\}$$

where

$$\langle\langle \tau \rangle\rangle \triangleq \forall\, \varphi\, .\, \varphi\, \tau \to \varphi\, \tau$$

Encoded module

$$e = \mathsf{unpack}\langle \alpha, y_1 \rangle = \mathsf{pack}\langle \mathtt{int}, \{\ell_t = \langle\langle \mathtt{int} \rangle\rangle, \ell_x = 42\} \rangle \ \mathsf{in}$$
$$\{\ell_t = \langle\langle \, ? \, \times \mathtt{bool} \rangle\rangle, \ell_x = (\ ?\ , \bot)\}$$

Example of encoding into F^{ω}

Source code

```
module type S =
  sig type t val x : t end
```

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

M^{ω} Signature

$\exists \alpha, \beta.$ **module** M : **sig**
  module X1 : **sig**
    type t = α
    val x : α
  end
  module X2 : **sig**
    type t = β
    val x : β
  end
end

F^{ω} type

$$\exists \alpha, \beta. \left\{ \begin{array}{l} \ell_{X_1} : \left\{ \begin{array}{l} \ell_t : \langle\langle \alpha \rangle\rangle \\ \ell_x : \alpha \end{array} \right\} \\ \ell_{X_2} : \left\{ \begin{array}{l} \ell_t : \langle\langle \beta \rangle\rangle \\ \ell_x : \beta \end{array} \right\} \end{array} \right\}$$

where

$$\langle\langle \tau \rangle\rangle \triangleq \forall \varphi . \varphi\, \tau \to \varphi\, \tau$$

Encoded module

$$e = \mathsf{unpack}\langle \alpha, y_1 \rangle = \mathsf{pack}\langle \mathtt{int}, \{\ell_t = \langle\langle \mathtt{int} \rangle\rangle, \ell_x = 42\} \rangle \ \mathsf{in}$$
$$\{\ell_t = \langle\langle \alpha \times \mathtt{bool} \rangle\rangle, \ell_x = (y_1.x, \bot)\}$$

Example of encoding into F^{ω}

Source code

```
module type S =
  sig type t val x : t end
```

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

M^{ω} Signature

```
∃α, β. module M : sig
  module X1 : sig
    type t = α
    val x : α
  end
  module X2 : sig
    type t = β
    val x : β
  end
end
```

F^{ω} type

$$\exists \alpha, \beta. \left\{ \begin{array}{l} \ell_{X_1} : \left\{ \begin{array}{l} \ell_t : \langle\langle \alpha \rangle\rangle \\ \ell_x : \alpha \end{array} \right\} \\ \ell_{X_2} : \left\{ \begin{array}{l} \ell_t : \langle\langle \beta \rangle\rangle \\ \ell_x : \beta \end{array} \right\} \end{array} \right\}$$

where

$$\langle\langle \tau \rangle\rangle \triangleq \forall\, \varphi\, .\, \varphi\, \tau \to \varphi\, \tau$$

Encoded module

$$e = \mathsf{unpack}\langle \alpha, y_1 \rangle = \mathsf{pack}\langle \mathtt{int}, \{\ell_t = \langle\langle \mathtt{int} \rangle\rangle, \ell_x = 42\} \rangle \text{ in } \mathsf{pack}\langle \alpha \times \mathtt{bool}, \{\ell_t = \langle\langle \alpha \times \mathtt{bool} \rangle\rangle, \ell_x = (y_1.x, \bot)\} \rangle$$

Example of encoding into F^{ω}

Source code

```
module type S =
  sig type t val x : t end
```

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

M^{ω} Signature

```
∃α, β. module M : sig
  module X1 : sig
    type t = α
    val x : α
  end
  module X2 : sig
    type t = β
    val x : β
  end
end
```

F^{ω} type

$$\exists \alpha, \beta. \left\{ \begin{array}{l} \ell_{X_1} : \left\{ \begin{array}{l} \ell_t : \langle\langle \alpha \rangle\rangle \\ \ell_x : \alpha \end{array} \right\} \\ \ell_{X_2} : \left\{ \begin{array}{l} \ell_t : \langle\langle \beta \rangle\rangle \\ \ell_x : \beta \end{array} \right\} \end{array} \right\}$$

where

$$\langle\langle \tau \rangle\rangle \triangleq \forall\, \varphi\, .\, \varphi\, \tau \to \varphi\, \tau$$

Encoded module

$$e = \begin{array}{l} \mathsf{unpack}\langle \alpha, y_1 \rangle = \mathsf{pack}\langle \mathtt{int}, \{\ell_t = \langle\langle \mathtt{int} \rangle\rangle, \ell_x = 42\} \rangle \ \mathsf{in} \\ \quad \mathsf{unpack}\langle \beta, y_2 \rangle = \mathsf{pack}\langle \alpha \times \mathtt{bool}, \{\ell_t = \langle\langle \alpha \times \mathtt{bool} \rangle\rangle, \ell_x = (y_1.x, \bot)\} \rangle \ \mathsf{in} \end{array}$$

Example of encoding into F^{ω}

Source code

```
module type S =
  sig type t val x : t end
```

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

M^{ω} Signature

```
∃α, β. module M : sig
  module X1 : sig
    type t = α
    val x : α
  end
  module X2 : sig
    type t = β
    val x : β
  end
end
```

F^{ω} type

$$\exists \alpha, \beta. \left\{ \begin{array}{l} \ell_{X_1} : \left\{ \begin{array}{l} \ell_t : \langle\langle \alpha \rangle\rangle \\ \ell_x : \alpha \end{array} \right\} \\ \ell_{X_2} : \left\{ \begin{array}{l} \ell_t : \langle\langle \beta \rangle\rangle \\ \ell_x : \beta \end{array} \right\} \end{array} \right\}$$

where

$$\langle\langle \tau \rangle\rangle \triangleq \forall\, \varphi\, .\, \varphi\, \tau \to \varphi\, \tau$$

Encoded module

$$\begin{aligned} e = \quad & \mathsf{unpack}\langle \alpha, y_1 \rangle = \mathsf{pack}\langle \mathtt{int}, \{\ell_t = \langle\langle \mathtt{int} \rangle\rangle, \ell_x = 42\} \rangle \ \mathsf{in} \\ & \mathsf{unpack}\langle \beta, y_2 \rangle = \mathsf{pack}\langle \alpha \times \mathtt{bool}, \{\ell_t = \langle\langle \alpha \times \mathtt{bool} \rangle\rangle, \ell_x = (y_1.x, \bot)\} \rangle \ \mathsf{in} \\ & \mathsf{pack}\langle \alpha, \beta, \{\ell_{X_1} = y_1, \ell_{X_2} = y_2\} \rangle \end{aligned}$$

Example of encoding into F^{ω}

Source code

```
1  module type S =
2    sig type t val x : t end
```

```
1  module M (Y: S)  = struct
2    module X1 = (struct
3      type t = int
4      let x = 42
5    end : S )
6    module X2 = (struct
7      type t = X1.t * bool
8      let x = (X1.x, false)
9    end : S )
10 end
```

M^{ω} Signature

```
∃α, β. module M : sig
  module X1 : sig
    type t = α
    val x : α
  end
  module X2 : sig
    type t = β
    val x : β
  end
end
```

F^{ω} type

$$\exists \alpha, \beta. \left\{ \begin{array}{l} \ell_{X_1} : \left\{ \begin{array}{l} \ell_t : \langle\langle \alpha \rangle\rangle \\ \ell_x : \alpha \end{array} \right\} \\ \ell_{X_2} : \left\{ \begin{array}{l} \ell_t : \langle\langle \beta \rangle\rangle \\ \ell_x : \beta \end{array} \right\} \end{array} \right\}$$

where

$$\langle\langle \tau \rangle\rangle \triangleq \forall\, \varphi\, .\, \varphi\, \tau \to \varphi\, \tau$$

Encoded module

$$\begin{aligned} e = \quad & \mathsf{unpack}\langle \alpha, y_1 \rangle = \mathsf{pack}\langle \mathtt{int}, \{\ell_t = \langle\langle \mathtt{int} \rangle\rangle, \ell_x = 42\} \rangle \ \mathsf{in} \\ & \mathsf{unpack}\langle \beta, y_2 \rangle = \mathsf{pack}\langle \alpha \times \mathtt{bool}, \{\ell_t = \langle\langle \alpha \times \mathtt{bool} \rangle\rangle, \ell_x = (y_1.x, \bot)\} \rangle \ \mathsf{in} \\ & \mathsf{pack}\langle \alpha, \beta, \{\ell_{X_1} = y_1, \ell_{X_2} = y_2\} \rangle \end{aligned}$$

Example of encoding into F^{ω}

Source code

```
module type S =
  sig type t val x : t end
```

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

M^{ω} Signature

$\exists \alpha', \beta'.$ **module** M :

$\forall \gamma.$(**sig type** t = γ **val** x : γ **end**) $\rightarrow$

```
sig
  module X1 : sig
    type t = α'(γ)
    val x : α'(γ)
  end
  module X2 : sig
    type t = β'(γ)
    val x : β'(γ)
  end
end
```

F^{ω} type

$$\exists \alpha, \beta. \left\{ \begin{array}{l} \ell_{X_1} : \left\{ \begin{array}{l} \ell_t : \langle\langle \alpha \rangle\rangle \\ \ell_x : \alpha \end{array} \right\} \\ \ell_{X_2} : \left\{ \begin{array}{l} \ell_t : \langle\langle \beta \rangle\rangle \\ \ell_x : \beta \end{array} \right\} \end{array} \right\}$$

where

$$\langle\langle \tau \rangle\rangle \triangleq \forall \varphi . \varphi\, \tau \rightarrow \varphi\, \tau$$

Encoded module

$$\begin{aligned} e = \quad & \mathrm{unpack}\langle \alpha, y_1 \rangle = \mathrm{pack}\langle \mathtt{int}, \{\ell_t = \langle\langle \mathtt{int} \rangle\rangle, \ell_x = 42\} \rangle \text{ in} \\ & \mathrm{unpack}\langle \beta, y_2 \rangle = \mathrm{pack}\langle \alpha \times \mathtt{bool}, \{\ell_t = \langle\langle \alpha \times \mathtt{bool} \rangle\rangle, \ell_x = (y_1.x, \bot)\} \rangle \text{ in} \\ & \mathrm{pack}\langle \alpha, \beta, \{\ell_{X_1} = y_1, \ell_{X_2} = y_2\} \rangle \end{aligned}$$

Example of encoding into F^{ω}

Source code

```
module type S =
  sig type t val x : t end
```

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

M^{ω} Signature

$\exists \alpha', \beta'.$ **module** M :

$\forall \gamma.$(**sig type** t = γ **val** x : γ **end**) $\rightarrow$

```
sig
  module X1 : sig
    type t = α'(γ)
    val x : α'(γ)
  end
  module X2 : sig
    type t = β'(γ)
    val x : β'(γ)
  end
end
```

F^{ω} type

$$\exists \alpha', \beta'. \forall \gamma. \{\ell_t : \langle\langle \gamma \rangle\rangle \; ; \; \ell_x : \gamma\} \rightarrow \left\{ \ell_{X_1} : \left\{ \begin{array}{l} \ell_t : \langle\langle \alpha'(\gamma) \rangle\rangle \\ \ell_x : \alpha'(\gamma) \end{array} \right\}, \ell_{X_2} : \left\{ \begin{array}{l} \ell_t : \langle\langle \beta'(\gamma) \rangle\rangle \\ \ell_x : \beta'(\gamma) \end{array} \right\} \right\}$$

where

$\langle\langle \tau \rangle\rangle \triangleq \forall \varphi . \varphi \; \tau \rightarrow \varphi \; \tau$

Encoded module

$$\begin{aligned} e = \; & \text{unpack}\langle \alpha, y_1 \rangle = \text{pack}\langle \texttt{int}, \{\ell_t = \langle\langle \texttt{int} \rangle\rangle, \ell_x = 42\} \rangle \text{ in} \\ & \text{unpack}\langle \beta, y_2 \rangle = \text{pack}\langle \alpha \times \texttt{bool}, \{\ell_t = \langle\langle \alpha \times \texttt{bool} \rangle\rangle, \ell_x = (y_1.x, \bot)\} \rangle \text{ in} \\ & \text{pack}\langle \alpha, \beta, \{\ell_{X_1} = y_1, \ell_{X_2} = y_2\} \rangle \end{aligned}$$

Example of encoding into F^{ω}

Source code

```
module type S =
  sig type t val x : t end
```

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

M^{ω} Signature

$\exists \alpha', \beta'.$ **module** M :

$\forall \gamma.$(**sig type** t = γ **val** x : γ **end**) $\rightarrow$

```
sig
  module X1 : sig
    type t = α'(γ)
    val x : α'(γ)
  end
  module X2 : sig
    type t = β'(γ)
    val x : β'(γ)
  end
end
```

F^{ω} type

$$\exists \alpha', \beta'. \forall \gamma. \{\ell_t : \langle\langle \gamma \rangle\rangle ; \ell_x : \gamma\} \rightarrow \left\{ \ell_{X_1} : \left\{ \begin{array}{l} \ell_t : \langle\langle \alpha'(\gamma) \rangle\rangle \\ \ell_x : \alpha'(\gamma) \end{array} \right\}, \ell_{X_2} : \left\{ \begin{array}{l} \ell_t : \langle\langle \beta'(\gamma) \rangle\rangle \\ \ell_x : \beta'(\gamma) \end{array} \right\} \right\}$$

where

$\langle\langle \tau \rangle\rangle \triangleq \forall \varphi . \varphi\, \tau \rightarrow \varphi\, \tau$

Encoded module

$$\begin{aligned} e = \quad & \mathsf{unpack}\langle \alpha, y_1 \rangle = \mathsf{pack}\langle \mathtt{int}, \{\ell_t = \langle\langle \mathtt{int} \rangle\rangle, \ell_x = 42\} \rangle \ \mathsf{in} \\ & \mathsf{unpack}\langle \beta, y_2 \rangle = \mathsf{pack}\langle \alpha \times \mathtt{bool}, \{\ell_t = \langle\langle \alpha \times \mathtt{bool} \rangle\rangle, \ell_x = (y_1.x, \bot)\} \rangle \ \mathsf{in} \\ & \mathsf{pack}\langle \alpha, \beta, \{\ell_{X_1} = y_1, \ell_{X_2} = y_2\} \rangle \\ e' = \quad & \Lambda \gamma. \lambda (Y : \{\ell_t : \langle\langle \gamma \rangle\rangle, \ell_x : \gamma\}). e \end{aligned}$$

Example of encoding into F^{ω}

Source code

```
1  module type S =
2    sig type t val x : t end
```

```
1  module M (Y: S)  = struct
2    module X1 = (struct
3      type t = int
4      let x = 42
5    end : S )
6    module X2 = (struct
7      type t = X1.t * bool
8      let x = (X1.x, false)
9    end : S )
10 end
```

M^{ω} Signature

$\exists \alpha', \beta'.$ **module** M :

$\forall \gamma.$(**sig type** t = γ **val** x : γ **end**) $\rightarrow$

```
sig
  module X1 : sig
    type t = α'(γ)
    val x : α'(γ)
  end
  module X2 : sig
    type t = β'(γ)
    val x : β'(γ)
  end
end
```

F^{ω} type

$$\exists \alpha', \beta'. \forall \gamma. \{\ell_t : \langle\langle \gamma \rangle\rangle \; ; \; \ell_x : \gamma\} \rightarrow \left\{ \ell_{X_1} : \left\{ \begin{array}{l} \ell_t : \langle\langle \alpha'(\gamma) \rangle\rangle \\ \ell_x : \alpha'(\gamma) \end{array} \right\} \; \ell_{X_2} : \left\{ \begin{array}{l} \ell_t : \langle\langle \beta'(\gamma) \rangle\rangle \\ \ell_x : \beta'(\gamma) \end{array} \right\} \right\}$$

where

$\langle\langle \tau \rangle\rangle \triangleq \forall \varphi . \varphi \; \tau \rightarrow \varphi \; \tau$

Encoded module

$$e = \; \text{unpack}\langle \alpha, y_1 \rangle = \text{pack}\langle \text{int}, \{\ell_t = \langle\langle \text{int} \rangle\rangle, \ell_x = 42\} \rangle \text{ in}$$

$$\text{unpack}\langle \beta, y_2 \rangle = \text{pack}\langle \alpha \times \text{bool}, \{\ell_t = \langle\langle \alpha \times \text{bool} \rangle\rangle, \ell_x = (y_1.x, \bot)\} \rangle \text{ in}$$

$$\text{pack}\langle \alpha, \beta, \{\ell_{X_1} = y_1, \ell_{X_2} = y_2\} \rangle$$

$$e' = \; \Lambda\gamma.\lambda(Y : \{\ell_t : \langle\langle \gamma \rangle\rangle, \ell_x : \gamma\}).e \quad : \quad \forall \gamma.(\{...\}) \rightarrow \exists \alpha, \beta. \{...\} \times$$

Example of encoding into F^{ω}

Source code

```
1  module type S =
2    sig type t val x : t end
```

```
1  module M (Y: S)  = struct
2    module X1 = (struct
3      type t = int
4      let x = 42
5    end : S )
6    module X2 = (struct
7      type t = X1.t * bool
8      let x = (X1.x, false)
9    end : S )
10 end
```

M^{ω} Signature

$\exists \alpha', \beta'.$ **module** M :

$\forall \gamma.$(**sig type** t = γ **val** x : γ **end**) $\rightarrow$

```
sig
  module X1 : sig
    type t = α'(γ)
    val x : α'(γ)
  end
  module X2 : sig
    type t = β'(γ)
    val x : β'(γ)
  end
end
```

F^{ω} type

$$\exists \alpha', \beta'. \forall \gamma. \{\ell_t : \langle\langle \gamma \rangle\rangle \; ; \; \ell_x : \gamma\} \rightarrow \left\{ \ell_{X_1} : \left\{ \begin{array}{l} \ell_t : \langle\langle \alpha'(\gamma) \rangle\rangle \\ \ell_x : \alpha'(\gamma) \end{array} \right\} \; ; \; \ell_{X_2} : \left\{ \begin{array}{l} \ell_t : \langle\langle \beta'(\gamma) \rangle\rangle \\ \ell_x : \beta'(\gamma) \end{array} \right\} \right\}$$

where

$\langle\langle \tau \rangle\rangle \triangleq \forall \varphi . \varphi \, \tau \rightarrow \varphi \, \tau$

Encoded module

$$e = \; \text{unpack}\langle \alpha, y_1 \rangle = \text{pack}\langle \text{int}, \{\ell_t = \langle\langle \text{int} \rangle\rangle, \ell_x = 42\} \rangle \text{ in}$$

$$\text{unpack}\langle \beta, y_2 \rangle = \text{pack}\langle \alpha \times \text{bool}, \{\ell_t = \langle\langle \alpha \times \text{bool} \rangle\rangle, \ell_x = (y_1.x, \bot)\} \rangle \text{ in}$$

$$\text{pack}\langle \alpha, \beta, \{\ell_{X_1} = y_1, \ell_{X_2} = y_2\} \rangle$$

$$e' = \; ?? \; \Lambda\gamma.\lambda(Y : \{\ell_t : \langle\langle \gamma \rangle\rangle, \ell_x : \gamma\}).e \quad : \quad \forall \gamma.(\{...\}) \rightarrow \exists \alpha, \beta. \{...\} \times$$

Extrusion and skolemization

- Through records/product types (extend scope between modules)
  $(\exists \alpha.\tau, \sigma) \rightsquigarrow \exists \alpha.(\tau, \sigma)$ ✓ (*repacking* pattern)

Extrusion and skolemization

- Through records/product types (extend scope between modules)
  $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$ ✓ (*repacking* pattern)
- Through arrow types (extend scope outside of applicative functors):
  $(\tau \to \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \to \sigma)$

Extrusion and skolemization

- Through records/product types (extend scope between modules)
  $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$ ✓ (*repacking* pattern)
- Through arrow types (extend scope outside of applicative functors):
  $(\tau \to \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \to \sigma)$ × unsound in general !

Extrusion and skolemization

- Through records/product types (extend scope between modules)
  $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$ ✓ (*repacking* pattern)
- Through arrow types (extend scope outside of applicative functors):
  $(\tau \to \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \to \sigma)$ × unsound in general !

$$\vdash \lambda b.\ \text{if } b \quad \text{then } (\text{pack } \langle \text{int}, 42 \rangle \text{ as } \exists\alpha.\alpha)$$
$$\text{else } (\text{pack } \langle \text{string}, \text{"Modules"} \rangle \text{ as } \exists\alpha.\alpha)$$

Extrusion and skolemization

- Through records/product types (extend scope between modules)
  $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$ ✓ (*repacking* pattern)
- Through arrow types (extend scope outside of applicative functors):
  $(\tau \to \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \to \sigma)$ × unsound in general !

$$\vdash \lambda b.\ \text{if } b \ \text{ then } (\text{pack } \langle \text{int}, 42\rangle \text{ as } \exists\alpha.\alpha) \qquad : \text{bool} \to \exists\alpha.\alpha$$
$$\text{else } \ (\text{pack } \langle \text{string}, \text{"Modules"}\rangle \text{ as } \exists\alpha.\alpha)$$

Extrusion and skolemization

- Through records/product types (extend scope between modules)
  $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$ ✓ (*repacking* pattern)
- Through arrow types (extend scope outside of applicative functors):
  $(\tau \to \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \to \sigma)$ × unsound in general !

$$\vdash \lambda b.\ \text{if } b \ \text{ then } (\text{pack } \langle \text{int}, 42\rangle \text{ as } \exists\alpha.\alpha) \qquad : \text{bool} \to \exists\alpha.\alpha$$
$$\text{else } \ (\text{pack } \langle \text{string}, \text{"Modules"}\rangle \text{ as } \exists\alpha.\alpha)$$

wrong to give it the type : $\exists\alpha.\text{bool} \to \alpha$

Extrusion and skolemization

- Through records/product types (extend scope between modules)
  $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$ ✓ (*repacking* pattern)
- Through arrow types (extend scope outside of applicative functors):
  $(\tau \to \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \to \sigma)$ × unsound in general !

$$\vdash \lambda b.\ \text{if } b \ \text{ then } (\text{pack } \langle \text{int}, 42\rangle \text{ as } \exists\alpha.\alpha) \qquad : \text{bool} \to \exists\alpha.\alpha$$
$$\text{else } \ (\text{pack } \langle \text{string}, \text{"Modules"}\rangle \text{ as } \exists\alpha.\alpha)$$

  wrong to give it the type : $\exists\alpha.\text{bool} \to \alpha$

- Through forall types (extend scope outside of applicative functors):

Extrusion and skolemization

- Through records/product types (extend scope between modules)
  $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$ ✓ (*repacking* pattern)
- Through arrow types (extend scope outside of applicative functors):
  $(\tau \to \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \to \sigma)$ × unsound in general !

$$\vdash \lambda b.\ \text{if } b \ \text{ then } (\text{pack } \langle \text{int}, 42\rangle \text{ as } \exists\alpha.\alpha) \qquad : \text{bool} \to \exists\alpha.\alpha$$
$$\text{else } \ (\text{pack } \langle \text{string}, \text{"Modules"}\rangle \text{ as } \exists\alpha.\alpha)$$

  wrong to give it the type : $\exists\alpha.\text{bool} \to \alpha$

- Through forall types (extend scope outside of applicative functors):
  $(\forall\beta.\exists\alpha.\tau) \rightsquigarrow \exists\alpha'.(\forall\beta.\tau[\alpha \mapsto \alpha'(\beta)])$

Extrusion and skolemization

- Through records/product types (extend scope between modules)
  $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$ ✓ (*repacking* pattern)
- Through arrow types (extend scope outside of applicative functors):
  $(\tau \to \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \to \sigma)$ × unsound in general !

$$\vdash \lambda b.\ \text{if } b \ \text{ then } (\text{pack } \langle \text{int}, 42\rangle \text{ as } \exists\alpha.\alpha) \qquad : \text{bool} \to \exists\alpha.\alpha$$
$$\text{else } \ (\text{pack } \langle \text{string}, \text{"Modules"}\rangle \text{ as } \exists\alpha.\alpha)$$

  wrong to give it the type : $\exists\alpha.\text{bool} \to \alpha$

- Through forall types (extend scope outside of applicative functors):
  $(\forall\beta.\exists\alpha.\tau) \rightsquigarrow \exists\alpha'.(\forall\beta.\tau[\alpha \mapsto \alpha'(\beta)])$ × not expressible in F^ω

Extrusion and skolemization

- Through records/product types (extend scope between modules)
  $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$ ✓ (*repacking* pattern)
- Through arrow types (extend scope outside of applicative functors):
  $(\tau \to \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \to \sigma)$ × unsound in general !

$$\vdash \lambda b.\ \text{if } b \text{ then } (\text{pack } \langle \text{int}, 42\rangle \text{ as } \exists\alpha.\alpha) \text{ else } (\text{pack } \langle \text{string}, \text{"Modules"}\rangle \text{ as } \exists\alpha.\alpha) : \text{bool} \to \exists\alpha.\alpha$$

  wrong to give it the type : $\exists\alpha.\text{bool} \to \alpha$

- Through forall types (extend scope outside of applicative functors):
  $(\forall\beta.\exists\alpha.\tau) \rightsquigarrow \exists\alpha'.(\forall\beta.\tau[\alpha \mapsto \alpha'(\beta)])$ × not expressible in F^ω

Some existentials are only used for abstraction, not for *heterogeneity*

F^{ω} with transparent existentials - Adding new constructs

$$\tau := \ \ldots \mid \exists^{\blacktriangledown} \alpha . \tau$$

$$\begin{aligned} e := \ & \ldots \mid \text{pack} \langle \tau, e \rangle \text{ as } \exists^{\blacktriangledown} \alpha . \tau \\ & \mid \text{unpack}^{\blacktriangledown} \langle \alpha, x \rangle = e \text{ in } e \end{aligned}$$

F^{ω} with transparent existentials - Adding new constructs

$$
\begin{aligned}
\tau &:= \ \ldots \mid \exists^{\blacktriangledown} \alpha.\, \tau \\
&\quad\ \mid \exists^{\triangledown \tau} \alpha.\, \tau \\
e &:= \ \ldots \mid \text{pack } \langle \tau, e \rangle \text{ as } \exists^{\blacktriangledown} \alpha.\, \tau \\
&\quad\ \mid \text{unpack}^{\blacktriangledown}\ \langle \alpha, x \rangle = e \text{ in } e
\end{aligned}
$$

F^{ω} with transparent existentials - Adding new constructs

$$
\begin{aligned}
\tau ::= \ & \ldots \mid \exists^{\blacktriangledown} \alpha.\, \tau \\
& \mid \exists^{\nabla \tau} \alpha.\, \tau \\
e ::= \ & \ldots \mid \mathsf{pack}\ \langle \tau, e \rangle\ \mathsf{as}\ \exists^{\blacktriangledown} \alpha.\, \tau \\
& \mid \mathsf{unpack}^{\blacktriangledown}\ \langle \alpha, x \rangle = e\ \mathsf{in}\ e \\
& \mid \mathsf{pack}\ e\ \mathsf{as}\ \exists^{\nabla \tau} \alpha.\, \sigma \\
& \mid \mathsf{repack}^{\nabla}\ \langle \alpha, x \rangle = e_1\ \mathsf{in}\ e_2 \\
& \mid \mathsf{seal}\ e \\
& \mid \mathsf{lift}^{\rightarrow} e \\
& \mid \mathsf{lift}^{\forall} e
\end{aligned}
$$

F^{ω} with transparent existentials - Typing rules

F-LiftArr

$$\frac{\Gamma \vdash e : \sigma_1 \to (\exists^{\triangledown\tau} \alpha.\, \sigma_2)}{\Gamma \vdash \mathsf{lift}^{\to} e : \exists^{\triangledown\tau} \alpha.\, (\sigma_1 \to \sigma_2)}$$

F^{ω} with transparent existentials - Typing rules

F-LiftArr

$$\frac{\Gamma \vdash e : \sigma_1 \to (\exists^{\nabla\tau} \alpha.\, \sigma_2)}{\Gamma \vdash \mathsf{lift}^{\to} e : \exists^{\nabla\tau} \alpha.\, (\sigma_1 \to \sigma_2)}$$

F-Repack

$$\frac{\Gamma \vdash e_1 : \exists^{\nabla\tau} \alpha.\, \sigma_1 \qquad \Gamma, \alpha, x : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \mathsf{repack}^{\nabla} \langle \alpha, x \rangle = e_1 \text{ in } e_2 : \exists^{\nabla\tau} \alpha.\, \sigma_2}$$

F^{ω} with transparent existentials - Typing rules

F-LiftArr

$$\frac{\Gamma \vdash e : \sigma_1 \to (\exists^{\nabla\tau} \alpha.\, \sigma_2)}{\Gamma \vdash \mathsf{lift}^{\to} e : \exists^{\nabla\tau} \alpha.\, (\sigma_1 \to \sigma_2)}$$

F-Repack

$$\frac{\Gamma \vdash e_1 : \exists^{\nabla\tau} \alpha.\, \sigma_1 \qquad \Gamma, \alpha, x : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \mathsf{repack}^{\nabla} \langle \alpha, x \rangle = e_1 \text{ in } e_2 : \exists^{\nabla\tau} \alpha.\, \sigma_2}$$

F-Hide

$$\frac{\Gamma \vdash \exists^{\nabla\tau} \alpha.\, \sigma : \star \qquad \Gamma \vdash e : \sigma[\alpha \mapsto \tau]}{\Gamma \vdash \mathsf{pack}\ e\ \mathsf{as}\ \exists^{\nabla\tau} \alpha.\, \sigma : \exists^{\nabla\tau} \alpha.\, \sigma}$$

F-Seal

$$\frac{\Gamma \vdash e : \exists^{\nabla\tau} \alpha.\, \sigma}{\Gamma \vdash \mathsf{seal}\ e : \exists^{\blacktriangledown} \alpha.\, \sigma}$$

F-LiftAll

$$\frac{\Gamma \vdash e : \forall \alpha.\, \exists^{\nabla\tau} \beta.\, \sigma}{\Gamma \vdash \mathsf{lift}^{\forall} e : \exists^{\nabla \lambda \alpha.\, \tau} \beta'.\, \forall \alpha.\, \sigma[\beta \mapsto (\beta'\ \alpha)]}$$

F^{ω} with transparent existentials - is actually vanilla F^{ω}!

$$
\begin{array}{lcl}
e_0 & \triangleq & \left\{ \begin{array}{lcl}
\mathsf{Pack} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x : \varphi\,\alpha).\,x \\
\mathsf{Seal} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x : \varphi\,\alpha).\,\mathsf{pack}\,\langle\alpha, x\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\alpha.\,\varphi\,\alpha \\
\mathsf{Repack} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x : \mathbb{E}\,\alpha\,\varphi).\,\Lambda\psi.\,\lambda(f : \forall\alpha.\,\varphi\,\alpha \to \psi\,\alpha).\,(f\,\alpha\,x) \\
\mathsf{Lift}^{\to} & = & \Lambda\alpha.\,\Lambda\varphi.\,\Lambda\beta.\,\lambda(f : \beta \to \varphi\,\alpha).\,f \\
\mathsf{Lift}^{\forall} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x : (\forall\beta.\,\varphi\,\beta(\alpha\,\beta))).\,x
\end{array} \right\} \\
\tau_0 & \triangleq & \lambda\alpha.\,\lambda\,\varphi\,.\,\varphi\,\alpha \\
\tau_{\mathbb{E}} & \triangleq & \exists^{\blacktriangledown}\mathbb{E}\,.\left\{ \begin{array}{lcl}
\mathsf{Pack} & : & \forall\alpha.\,\forall\,\varphi\,.\,\varphi\,\alpha \to \mathbb{E}\,\alpha\,\varphi \\
\mathsf{Seal} & : & \forall\alpha.\,\forall\,\varphi\,.\,\mathbb{E}\,\alpha\,\varphi \to \exists^{\blacktriangledown}\alpha.\,\varphi\,\alpha \\
\mathsf{Repack} & : & \forall\alpha.\,\forall\,\varphi\,.\,\mathbb{E}\,\alpha\,\varphi \to \forall\,\psi\,.\,(\forall\alpha.\,\varphi\,\alpha \to \psi\,\alpha) \to \mathbb{E}\,\alpha\,\psi \\
\mathsf{Lift}^{\to} & : & \forall\alpha.\,\forall\,\varphi\,.\,\forall\beta.\,(\beta \to \mathbb{E}\,\alpha\,\varphi) \to \mathbb{E}\,\alpha\,(\lambda\alpha.\,\beta \to \varphi\,\alpha) \\
\mathsf{Lift}^{\forall} & : & \forall\alpha.\,\forall\,\varphi\,.\,(\forall\beta.\mathbb{E}\,(\alpha\,\beta)\,\,(\varphi\,\beta)) \to \mathbb{E}\,\alpha\,\,(\lambda\alpha.\forall\beta.\,\varphi\,\beta\,\,(\alpha\,\beta))
\end{array} \right\} \\
e_{\mathbb{E}} & \triangleq & \mathsf{pack}\,\langle\tau_0, e_0\rangle\ \mathsf{as}\ \tau_{\mathbb{E}}
\end{array}
$$

Extrusion and skolemization (with transparent existentials)

- Through records/product types (extend scope between modules)
  $(\exists^{\blacktriangledown}\alpha.\,\tau, \sigma) \rightsquigarrow \exists^{\blacktriangledown}\alpha.\,(\tau, \sigma)$ ✓ (*repacking* pattern)
  $(\exists^{\nabla\rho}\alpha.\,\tau, \sigma) \rightsquigarrow \exists^{\nabla\rho}\alpha.\,(\tau, \sigma)$ ✓ (using repack$^{\nabla}$)

Extrusion and skolemization (with transparent existentials)

- Through records/product types (extend scope between modules)
  $(\exists^{\blacktriangledown}\alpha.\tau, \sigma) \rightsquigarrow \exists^{\blacktriangledown}\alpha.(\tau, \sigma)$ ✓ (*repacking* pattern)
  $(\exists^{\nabla\rho}\alpha.\tau, \sigma) \rightsquigarrow \exists^{\nabla\rho}\alpha.(\tau, \sigma)$ ✓ (using repack^{∇})

- Through arrow types (extend scope outside of applicative functors):
  $(\tau \to \exists^{\blacktriangledown}\alpha.\sigma) \rightsquigarrow \exists^{\blacktriangledown}\alpha.(\tau \to \sigma)$ ×
  $(\tau \to \exists^{\nabla\rho}\alpha.\sigma) \rightsquigarrow \exists^{\nabla\rho}\alpha.(\tau \to \sigma)$ ✓ (using $\text{lift}^{\to}$)

Extrusion and skolemization (with transparent existentials)

- Through records/product types (extend scope between modules)
  $(\exists^{\blacktriangledown}\alpha.\tau, \sigma) \rightsquigarrow \exists^{\blacktriangledown}\alpha.(\tau, \sigma)$ ✓ (*repacking* pattern)
  $(\exists^{\nabla\rho}\alpha.\tau, \sigma) \rightsquigarrow \exists^{\nabla\rho}\alpha.(\tau, \sigma)$ ✓ (using repack$^{\nabla}$)

- Through arrow types (extend scope outside of applicative functors):
  $(\tau \to \exists^{\blacktriangledown}\alpha.\sigma) \rightsquigarrow \exists^{\blacktriangledown}\alpha.(\tau \to \sigma)$ ×
  $(\tau \to \exists^{\nabla\rho}\alpha.\sigma) \rightsquigarrow \exists^{\nabla\rho}\alpha.(\tau \to \sigma)$ ✓ (using lift$^{\to}$)

- Through forall types (extend scope outside of applicative functors):
  $(\forall\beta.\exists^{\blacktriangledown}\alpha.\tau) \rightsquigarrow \exists^{\blacktriangledown}\alpha'.(\forall\beta.\tau[\alpha \mapsto \alpha'(\beta)])$ ×
  $(\forall\beta.\exists^{\nabla\rho}\alpha.\tau) \rightsquigarrow \exists^{\nabla(\lambda\beta.\rho)}\alpha'.(\forall\beta.\tau[\alpha \mapsto (\alpha'\ \beta)])$ ✓ (using lift$^{\forall}$)

M^{ω}- Properties

Theorem (Completeness)

$\Gamma \vdash \mathtt{M} : \exists^{\diamond}\overline{\alpha}.C$ *implies* $\Gamma \vdash \mathtt{M} : \exists^{\diamond}\overline{\alpha}.C \rightsquigarrow e$ *for some* e

M^{ω}- Properties

Theorem (Completeness)

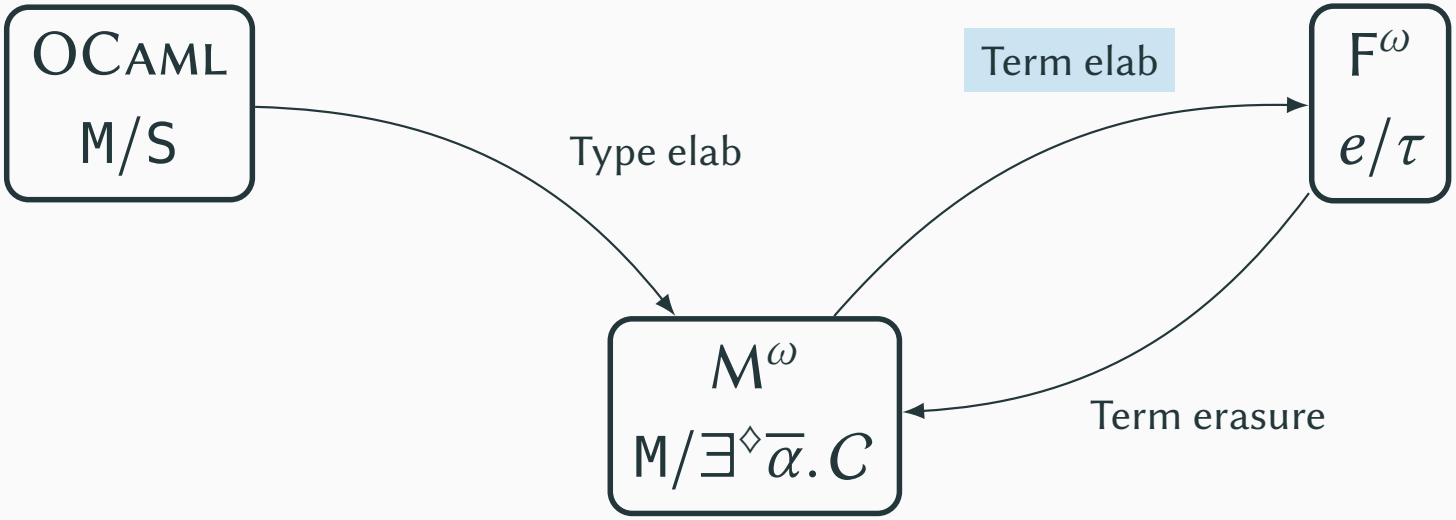
$\Gamma \vdash \mathtt{M} : \exists^{\diamond}\overline{\alpha}.C$ *implies* $\Gamma \vdash \mathtt{M} : \exists^{\diamond}\overline{\alpha}.C \rightsquigarrow e$ *for some* e

Theorem (Soundness)

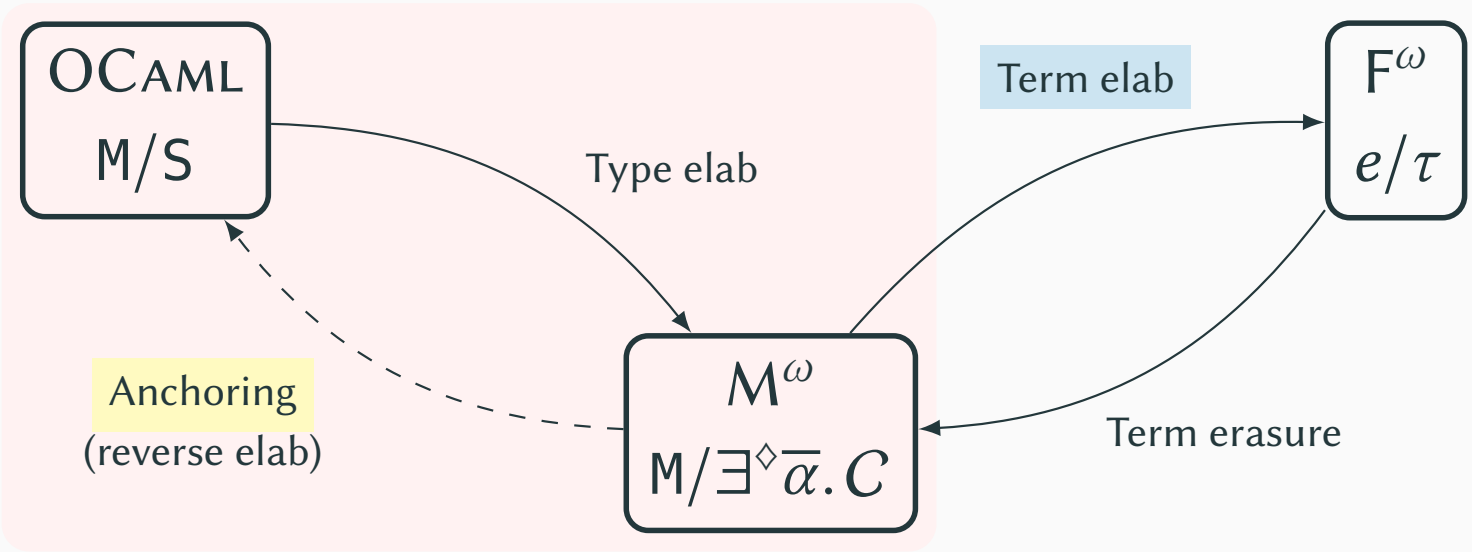
$\Gamma \vdash \mathtt{M} : \exists^{\diamond}\overline{\alpha}.C \rightsquigarrow e$ *implies* $\Gamma \vdash e : \exists^{\diamond}\overline{\alpha}.C$ *(in* F^{ω}*)*

(e *and* $\mathtt{M}$ *have the same untyped semantics)*

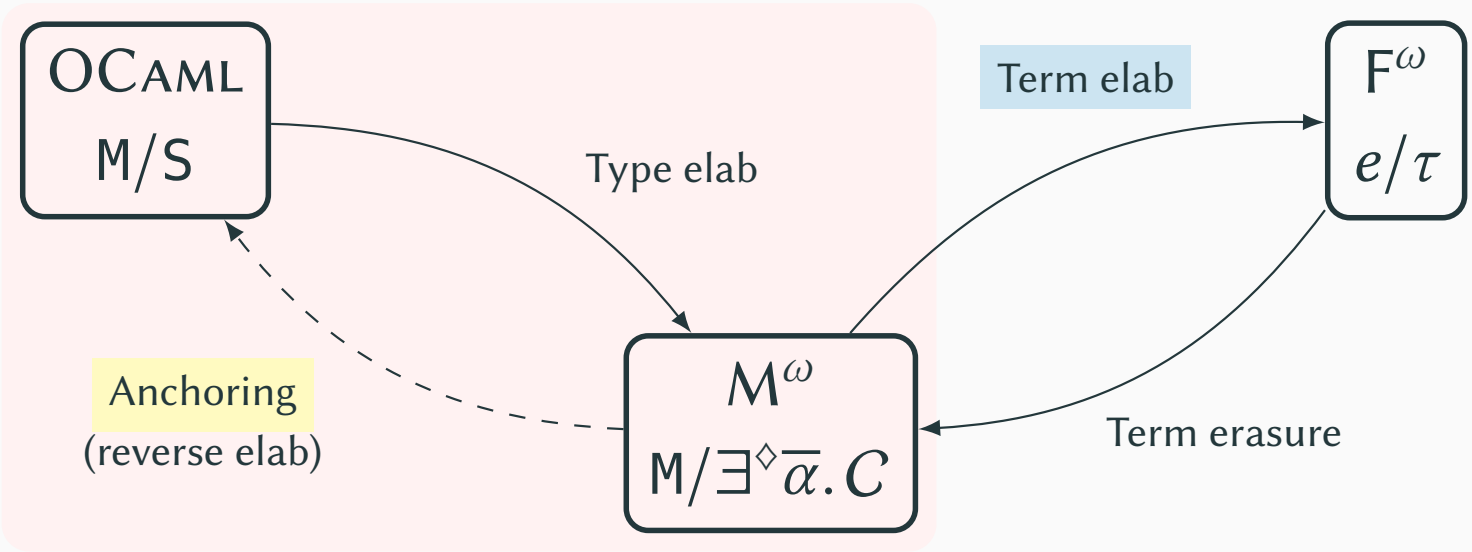
The big picture (again)



The big picture (again)

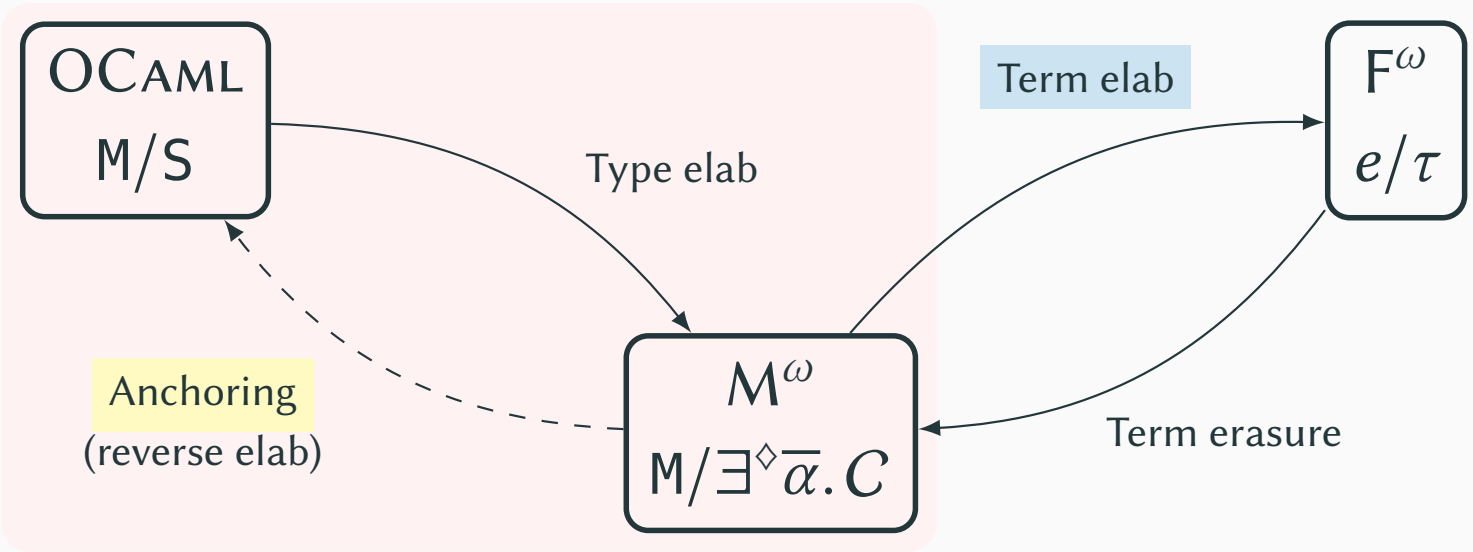


The big picture (again)



- ✓ Significant subset of OCaml
- ✓ Simple, classic scheme
- ✓ Clear treatment of avoidance
- ✓ Transparent signatures (aliasing), abstract signatures, etc.

The big picture (again)



- ✓ Significant subset of OCaml
- ✓ Simple, classic scheme
- ✓ Clear treatment of avoidance
- ✓ Transparent signatures (aliasing), abstract signatures, etc.
- ~ Explainability (elaboration)
- ~ Keeping user-defined names
- × Major change

ZipML

Avoiding signature avoidance

```
(* replaced N by its definition *)
module M = (struct
  type t = A of int | B of bool
  module X = struct let x = (A 1) end
end).X
```

OCaml:

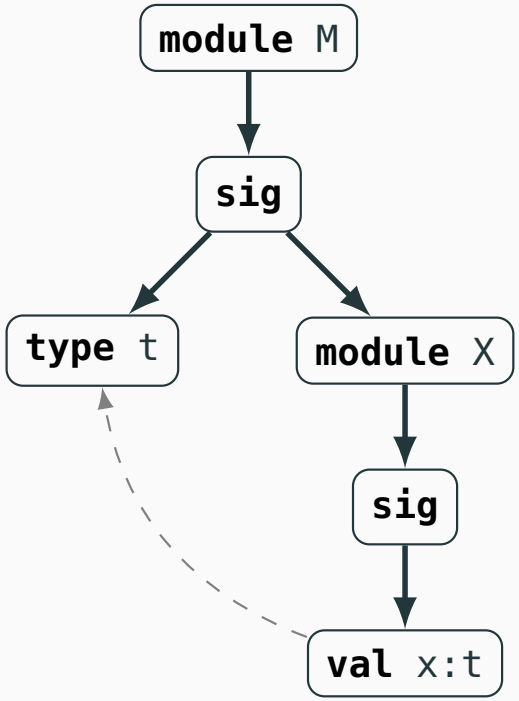
```
The value x has no valid type if t is hidden.
```

Avoiding signature avoidance

```
1 (* replaced N by its definition *)
2 module M = (struct
3   type t = A of int | B of bool
4   module X = struct let x = (A 1) end
5 end).X
```

OCaml:

```
The value x has no valid type if t is hidden.
```

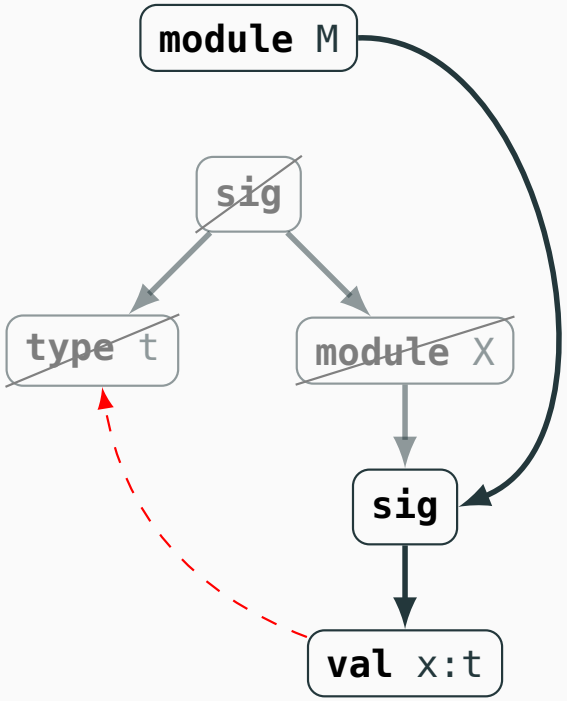


Avoiding signature avoidance

```
1 (* replaced N by its definition *)
2 module M = (struct
3   type t = A of int | B of bool
4   module X = struct let x = (A 1) end
5 end).X
```

OCaml:

```
The value x has no valid type if t is hidden.
```

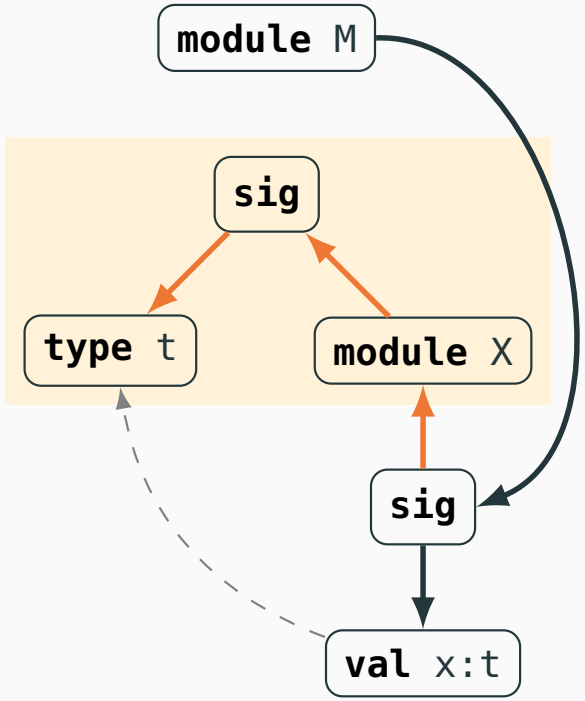


Avoiding signature avoidance

```
1 (* replaced N by its definition *)
2 module M = (struct
3   type t = A of int | B of bool
4   module X = struct let x = (A 1) end
5 end).X
```

OCaml:

```
The value x has no valid type if t is hidden.
```



Avoiding signature avoidance

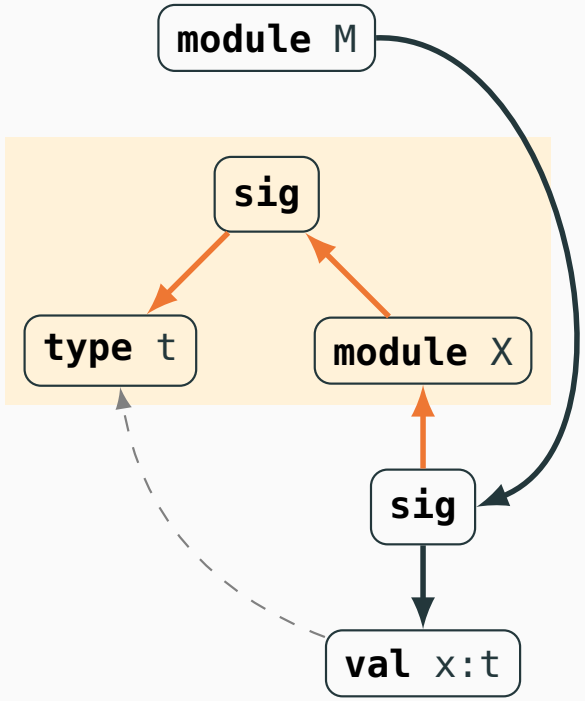
```
1 (* replaced N by its definition *)
2 module M = (struct
3   type t = A of int | B of bool
4   module X = struct let x = (A 1) end
5 end).X
```

OCaml:

```
The value x has no valid type if t is hidden.
```

ZipML:

```
module M :
< type t = A of int | B of bool>
sig val x : t end
```



Avoiding signature avoidance

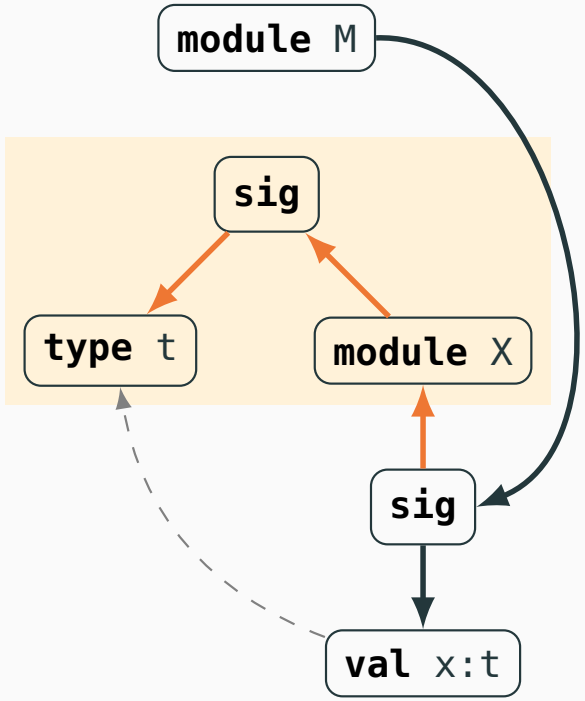
```
1 (* replaced N by its definition *)
2 module M = (struct
3   type t = A of int | B of bool
4   module X = struct let x = (A 1) end
5 end).X
```

OCaml:

```
The value x has no valid type if t is hidden.
```

ZipML:

```
module M :
< type t = A of int | B of bool>
sig val x : t end
```



→ Extend the signature language with zippers – $\langle \gamma \rangle$ S

→ The *zipper context* γ acts as an hidden typing environment attached to S

Zipper simplification

```
(* replaced N by its definition *)
module M = (struct
  type t
  type u = t list
  module X = struct type v = t let x : u = [] end
end).X
```

OCaml:

```
The value x has no valid type if t is hidden.
```

Zipper simplification

```
(* replaced N by its definition *)
module M = (struct
  type t
  type u = t list
  module X = struct type v = t let x : u = [] end
end).X
```

OCaml:

```
The value x has no valid type if t is hidden.
```

ZipML:

```
<type t  type u = t list > sig type v = t val x : u  end
```

Zipper simplification

```
(* replaced N by its definition *)
module M = (struct
  type t
  type u = t list
  module X = struct type v = t let x : u = [] end
end).X
```

OCaml:

```
The value x has no valid type if t is hidden.
```

ZipML:

```
<type t  type u = t list > sig type v = t val x : u  end
<type t  type u = t list > sig type v = t val x : t list end
```

Zipper simplification

```
(* replaced N by its definition *)
module M = (struct
  type t
  type u = t list
  module X = struct type v = t let x : u = [] end
end).X
```

OCaml:

```
The value x has no valid type if t is hidden.
```

ZipML:

```
<type t  type u = t list > sig type v = t val x : u  end
<type t  type u = t list > sig type v = t val x : t list end
<type t >                  sig type v = t val x : t list end
```

Zipper simplification

```
1 (* replaced N by its definition *)
2 module M = (struct
3   type t
4   type u = t list
5   module X = struct type v = t let x : u = [] end
6 end).X
```

OCaml:

```
The value x has no valid type if t is hidden.
```

ZipML:

```
<type t  type u = t list > sig type v = t val x : u  end
<type t  type u = t list > sig type v = t val x : t list end
<type t >                  sig type v = t val x : t list end
                           sig type v   val x : v list end <type t = v >
```

Zipper simplification

```
(* replaced N by its definition *)
module M = (struct
  type t
  type u = t list
  module X = struct type v = t let x : u = [] end
end).X
```

OCaml:

```
The value x has no valid type if t is hidden.
```

ZipML:

```
<type t  type u = t list > sig type v = t val x : u  end
<type t  type u = t list > sig type v = t val x : t list end
<type t >                  sig type v = t val x : t list end
                           sig type v   val x : v list end <type t = v >
                           sig type v   val x : v list end
```

Zipper simplification

```
(* replaced N by its definition *)
module M = (struct
  type t
  type u = t list
  module X = struct type v = t let x : u = [] end
end).X
```

OCaml:

```
The value x has no valid type if t is hidden.
```

ZipML:

```
<type t  type u = t list > sig type v = t val x : u  end
<type t  type u = t list > sig type v = t val x : t list end
<type t >                  sig type v = t val x : t list end
                           sig type v     val x : v list end <type t = v >
                           sig type v     val x : v list end
```

→ valid signature in the source syntax!

Typing with zippers

Zipper introduction

Projections on structure never fails
(no signature avoidance)

$$\textsc{Typ-M-ProjA}\ \frac{\Gamma \vdash \mathtt{M} : \langle \gamma \rangle\ \mathtt{sig}\ \overline{\mathtt{D}}, \mathtt{module}\ X : \mathtt{S}, \overline{\mathtt{D}}'\ \mathtt{end}}{\Gamma \vdash \mathtt{M}.X : \left\langle \gamma \,;\, \overline{\mathtt{D}} \right\rangle \mathtt{S}}$$

Typing with zippers

Zipper introduction

Projections on structure never fails (no signature avoidance)

Typ-M-ProjA

$$\frac{\Gamma \vdash \mathtt{M} : \langle \gamma \rangle\ \mathtt{sig}\ \overline{\mathtt{D}}, \mathtt{module}\ X : \mathtt{S}, \overline{\mathtt{D}}'\ \mathtt{end}}{\Gamma \vdash \mathtt{M}.X : \left\langle \gamma \,;\, \overline{\mathtt{D}} \right\rangle \mathtt{S}}$$

Zipper removal (subtyping)

Zippers can disappear through explicit signature ascription (including `.mli`)

Typ-M-Ascr

$$\frac{\Gamma \vdash \mathtt{S} \qquad \Gamma \vdash \mathtt{M} : \langle \gamma \rangle\, \mathtt{S}' \qquad \Gamma, \gamma \vdash \mathtt{S}' \leqslant \mathtt{S}}{\Gamma \vdash^{\diamond} (\mathtt{M} : \mathtt{S}) : \mathtt{S}}$$

Typing with zippers

Zipper introduction

Projections on structure never fails (no signature avoidance)

Typ-M-ProjA

$$\frac{\Gamma \vdash \mathtt{M} : \langle \gamma \rangle\ \mathtt{sig}\ \overline{\mathtt{D}}, \mathtt{module}\ X : \mathtt{S}, \overline{\mathtt{D}}'\ \mathtt{end}}{\Gamma \vdash \mathtt{M}.X : \left\langle \gamma \,;\, \overline{\mathtt{D}} \right\rangle \mathtt{S}}$$

Zipper removal (subtyping)

Zippers can disappear through explicit signature ascription (including `.mli`)

Typ-M-Ascr

$$\frac{\Gamma \vdash \mathtt{S} \qquad \Gamma \vdash \mathtt{M} : \langle \gamma \rangle\, \mathtt{S}' \qquad \Gamma, \gamma \vdash \mathtt{S}' \leqslant \mathtt{S}}{\Gamma \vdash^{\diamond} (\mathtt{M} : \mathtt{S}) : \mathtt{S}}$$

Zipper simplification

Garbage-collection of useless fields

Typ-M-Simpl

$$\frac{\Gamma \vdash \mathtt{M} : \langle \gamma \rangle\, \mathtt{S} \qquad \Gamma \vdash \langle \gamma \rangle\, \mathtt{S} \rightsquigarrow \left\langle \gamma' \right\rangle \mathtt{S}'}{\Gamma \vdash \mathtt{M} : \left\langle \gamma' \right\rangle \mathtt{S}'}$$

Typing with zippers

Zipper introduction

Projections on structure never fails (no signature avoidance)

Typ-M-ProjA

$$\frac{\Gamma \vdash \mathtt{M} : \langle \gamma \rangle\ \mathtt{sig}\ \overline{\mathtt{D}}, \mathtt{module}\ X : \mathtt{S}, \overline{\mathtt{D}}'\ \mathtt{end}}{\Gamma \vdash \mathtt{M}.X : \left\langle \gamma\,;\overline{\mathtt{D}} \right\rangle \mathtt{S}}$$

Zipper removal (subtyping)

Zippers can disappear through explicit signature ascription (including `.mli`)

Typ-M-Ascr

$$\frac{\Gamma \vdash \mathtt{S} \qquad \Gamma \vdash \mathtt{M} : \langle \gamma \rangle\, \mathtt{S}' \qquad \Gamma, \gamma \vdash \mathtt{S}' \leqslant \mathtt{S}}{\Gamma \vdash^{\diamond} (\mathtt{M} : \mathtt{S}) : \mathtt{S}}$$

Zipper simplification

Garbage-collection of useless fields

Typ-M-Simpl

$$\frac{\Gamma \vdash \mathtt{M} : \langle \gamma \rangle\, \mathtt{S} \qquad \Gamma \vdash \langle \gamma \rangle\, \mathtt{S} \rightsquigarrow \left\langle \gamma' \right\rangle \mathtt{S}'}{\Gamma \vdash \mathtt{M} : \left\langle \gamma' \right\rangle \mathtt{S}'}$$

Theorem (Preservation of typing)

Simplification never prevents further typing (no over-abstraction)

ZipML- Properties

ZipML- Properties

Theorem (Soundness)

Well-typed programs do not crash

ZipML- Properties

Theorem (Soundness)

Well-typed programs do not crash

By elaboration: typing a module expression in ZipML implies typing of the same module expression in M^{ω}

ZipML- Properties

Theorem (Soundness)

Well-typed programs do not crash

By elaboration: typing a module expression in ZipML implies typing of the same module expression in M^{ω}

Conjecture (Completeness)

M^{ω} and ZipML have the same expressivity

ZipML- Properties

Theorem (Soundness)

Well-typed programs do not crash

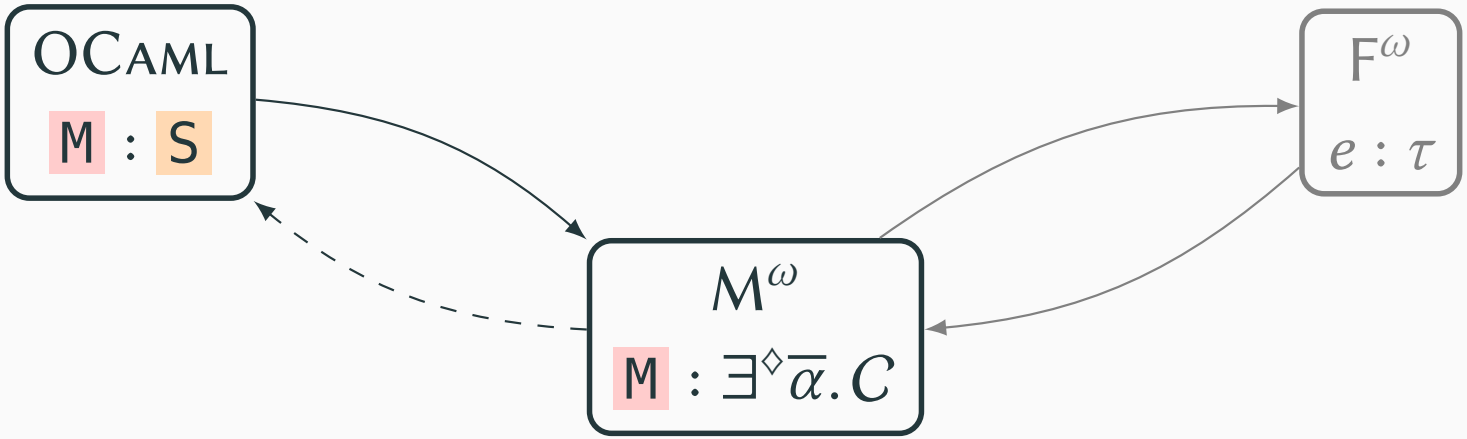
By elaboration: typing a module expression in ZipML implies typing of the same module expression in M^{ω}

Conjecture (Completeness)

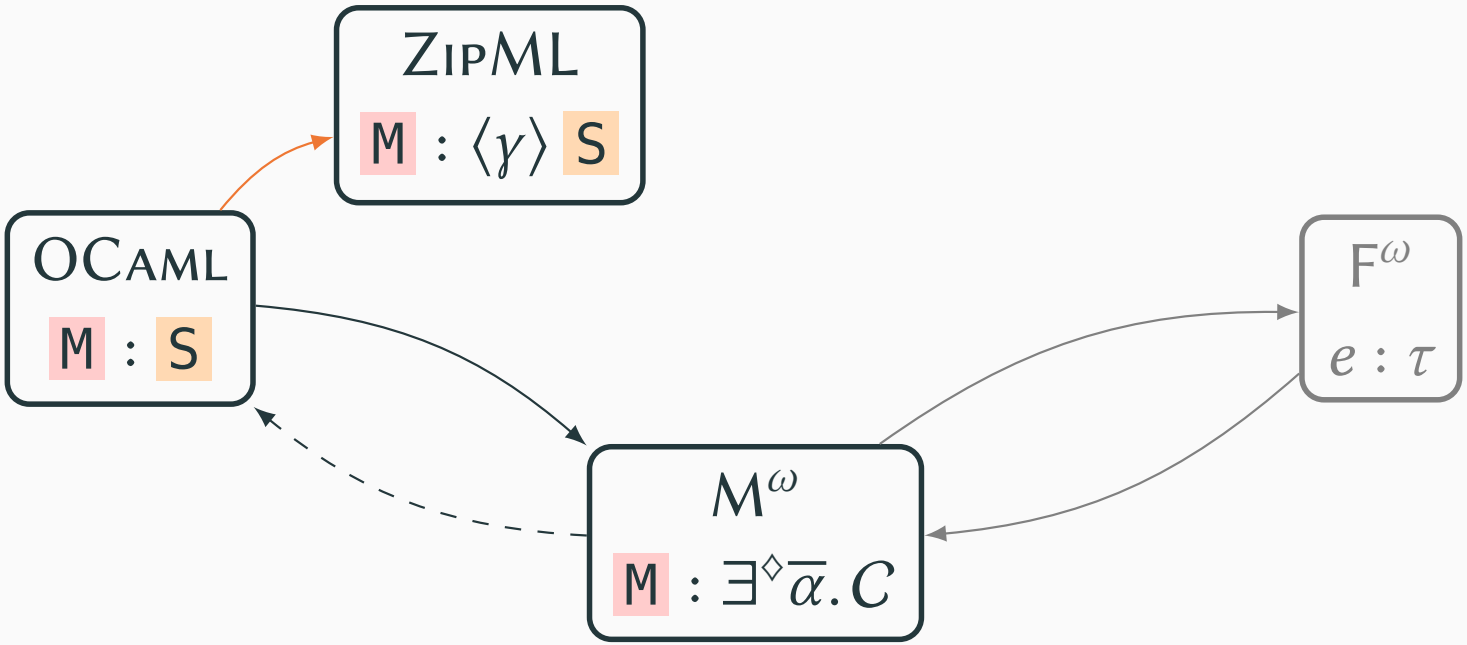
M^{ω} and ZipML have the same expressivity

Zippers can be used to represent quantified variables and to simulate extrusion

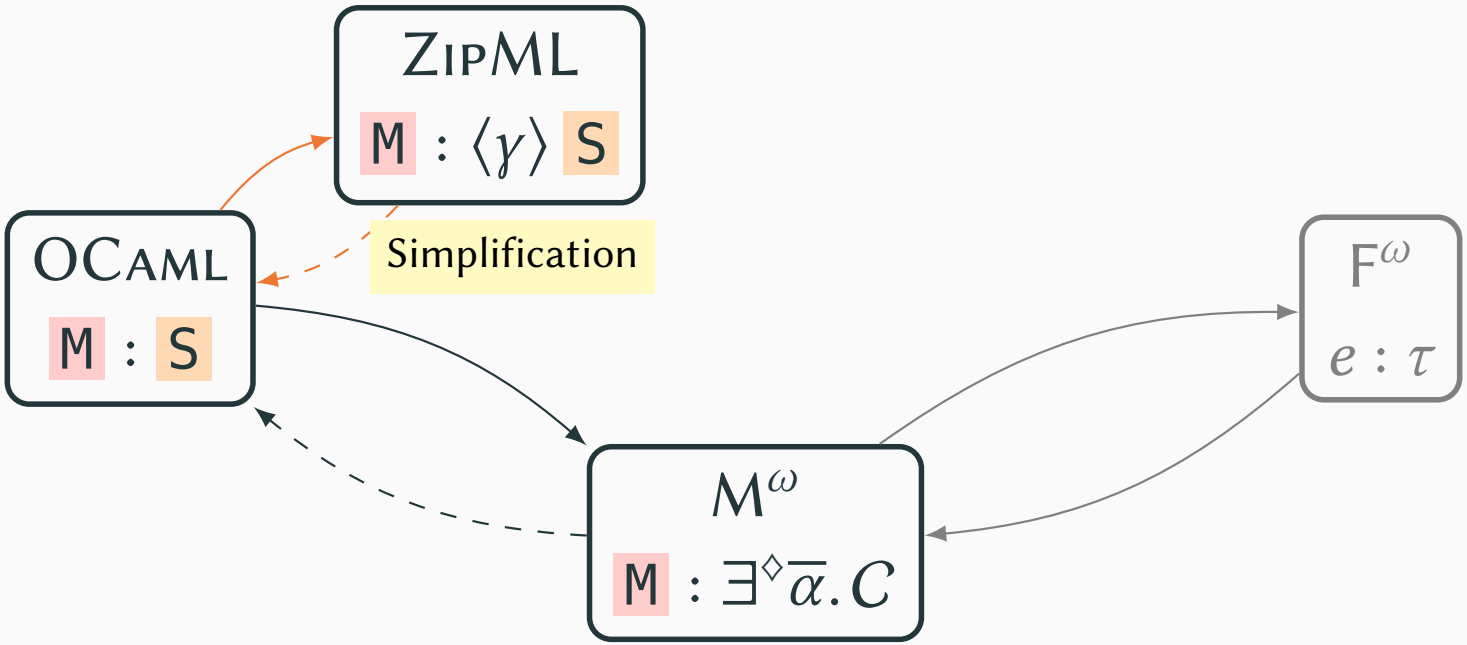
The big picture (again) (again)



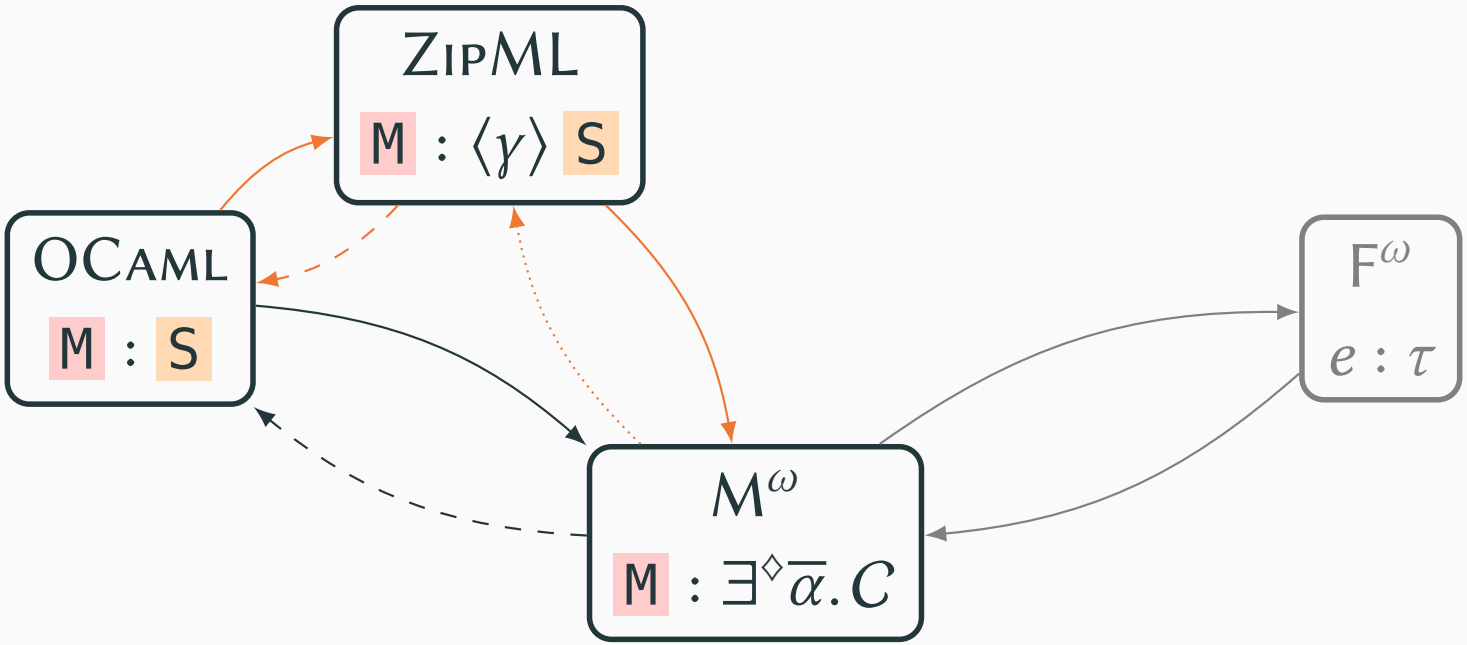
The big picture (again) (again)



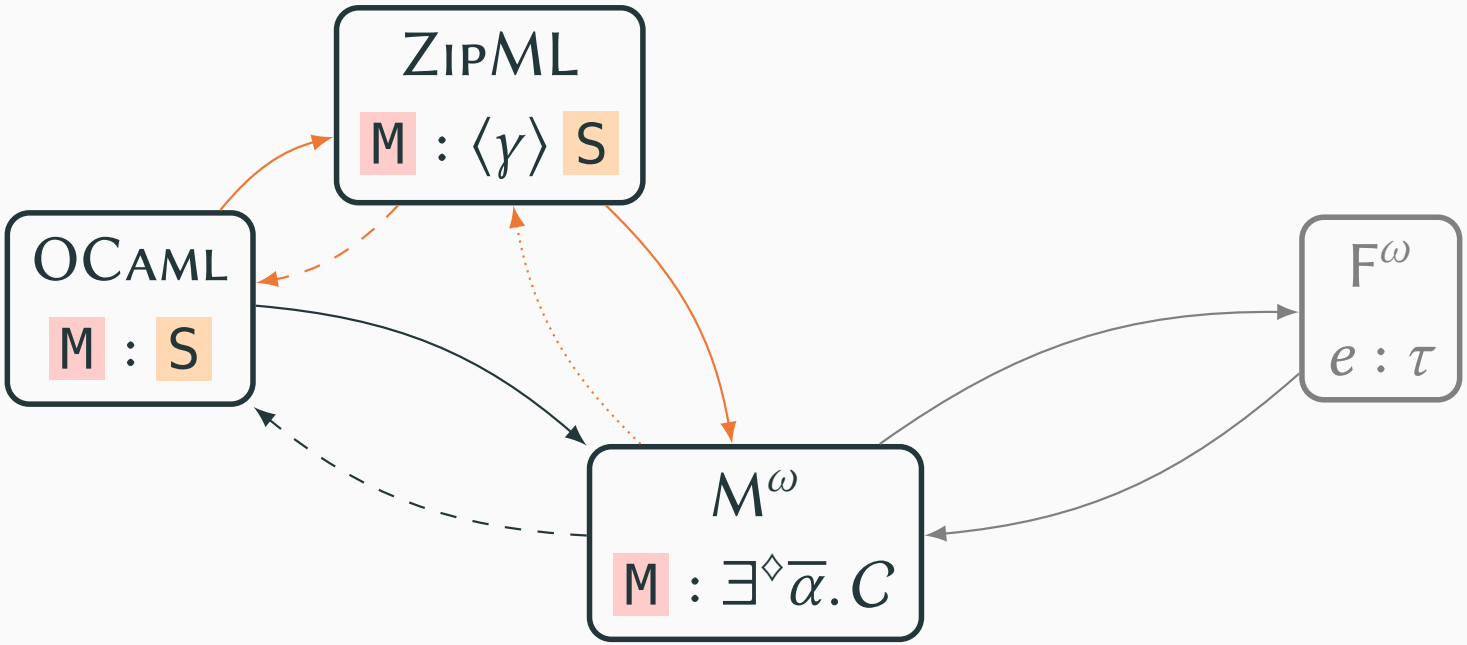
The big picture (again) (again)



The big picture (again) (again)

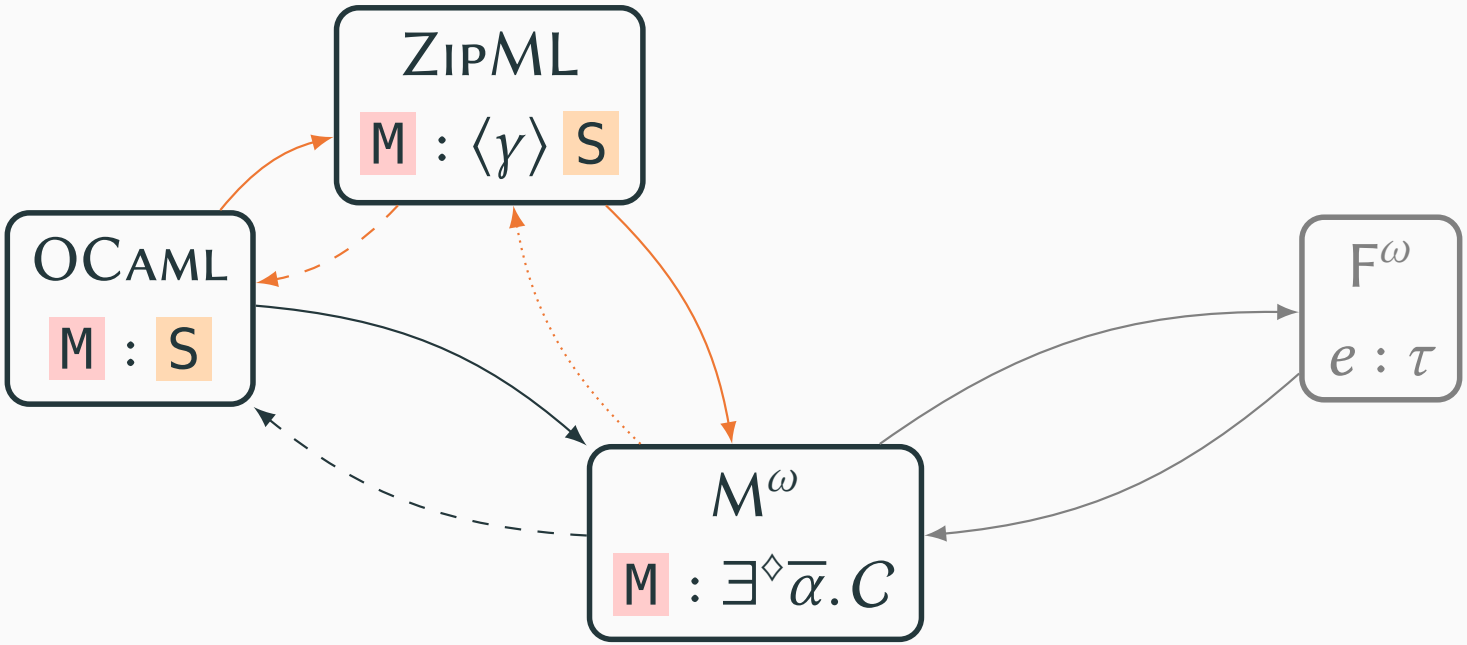


The big picture (again) (again)



- ✓ Significant subset of OCaml
- ✓ Clear treatment of avoidance
- ✓ Module identities, transparent ascription, etc.

The big picture (again) (again)



- ✓ Significant subset of OCaml
- ✓ Clear treatment of avoidance
- ✓ Module identities, transparent ascription, etc.
- ✓ User-defined names
- ✓ Lazyness
- ✓ Retrofit

Conclusion

Take-away

Take-away

$\mathbf{M}^{\omega}$

- A type system based on translation into F^{ω}
- Transparent existentials for sealing inside applicative functors

Take-away

$\mathbf{M}^{\omega}$

- A type system based on translation into F^{ω}
- Transparent existentials for sealing inside applicative functors

OOPSLA 2024 - *Fulfilling OCaml modules with transparency*

Take-away

M^ω

- A type system based on translation into F^ω
- Transparent existentials for sealing inside applicative functors

OOPSLA 2024 - *Fulfilling OCaml modules with transparency*

ZipML

- A (almost) purely-syntactic type system tailored to OCaml
- Zipper signatures, simplification

Take-away

M^ω

- A type system based on translation into F^ω
- Transparent existentials for sealing inside applicative functors

OOPSLA 2024 - *Fulfilling OCaml modules with transparency*

ZipML

- A (almost) purely-syntactic type system tailored to OCaml
- Zipper signatures, simplification

POPL 2025 - *Avoiding signature avoidance in ML modules with zippers*

Take-away

M^ω

- A type system based on translation into F^ω
- Transparent existentials for sealing inside applicative functors

OOPSLA 2024 - *Fulfilling OCaml modules with transparency*

ZipML

- A (almost) purely-syntactic type system tailored to OCaml
- Zipper signatures, simplification

POPL 2025 - *Avoiding signature avoidance in ML modules with zippers*

There's more in the manuscript!

- Transparent signatures (abstraction safety), abstract signatures in M^ω

Take-away

M^ω

- A type system based on translation into F^ω
- Transparent existentials for sealing inside applicative functors

OOPSLA 2024 - *Fulfilling OCaml modules with transparency*

ZipML

- A (almost) purely-syntactic type system tailored to OCaml
- Zipper signatures, simplification

POPL 2025 - *Avoiding signature avoidance in ML modules with zippers*

There's more in the manuscript!

- Transparent signatures (abstraction safety), abstract signatures in M^ω
- Anchoring in M^ω (with properties)

Take-away

M^ω

- A type system based on translation into F^ω
- Transparent existentials for sealing inside applicative functors

OOPSLA 2024 - *Fulfilling OCaml modules with transparency*

ZipML

- A (almost) purely-syntactic type system tailored to OCaml
- Zipper signatures, simplification

POPL 2025 - *Avoiding signature avoidance in ML modules with zippers*

There's more in the manuscript!

- Transparent signatures (abstraction safety), abstract signatures in M^ω
- Anchoring in M^ω (with properties)
- Preservation of user-definitions in ZipML

Take-away

M^ω

- A type system based on translation into F^ω
- Transparent existentials for sealing inside applicative functors

OOPSLA 2024 - *Fulfilling OCaml modules with transparency*

ZipML

- A (almost) purely-syntactic type system tailored to OCaml
- Zipper signatures, simplification

POPL 2025 - *Avoiding signature avoidance in ML modules with zippers*

There's more in the manuscript!

- Transparent signatures (abstraction safety), abstract signatures in M^ω
- Anchoring in M^ω (with properties)
- Preservation of user-definitions in ZipML
- Early and lazy strengthening

Take-away

M^ω

- A type system based on translation into F^ω
- Transparent existentials for sealing inside applicative functors

OOPSLA 2024 - *Fulfilling OCaml modules with transparency*

ZipML

- A (almost) purely-syntactic type system tailored to OCaml
- Zipper signatures, simplification

POPL 2025 - *Avoiding signature avoidance in ML modules with zippers*

There's more in the manuscript!

- Transparent signatures (abstraction safety), abstract signatures in M^ω
- Anchoring in M^ω (with properties)
- Preservation of user-definitions in ZipML
- Early and lazy strengthening
- Overview of the design space of module systems besides abstraction

Future works

Missing features

- The *easy* ones: **`include`**, **`open`**, 1st-class modules, **`module type of`**, signature modifiers (**`with type`**)
- recursive modules, mixins

Future works

Missing features

- The *easy* ones: **`include`**, **`open`**, 1st-class modules, **`module type of`**, signature modifiers (**`with type`**)
- recursive modules, mixins

Implementation

- Split the path from OCaml to ZipML into small steps

Future works

Missing features

- The *easy* ones: **`include`**, **`open`**, 1st-class modules, **`module type of`**, signature modifiers (**`with type`**)
- recursive modules, mixins

Implementation

- Split the path from OCaml to ZipML into small steps

User-surveys

- Embrace the human and sociological aspects of programming

Future works

Missing features

- The *easy* ones: **`include`**, **`open`**, 1st-class modules, **`module type of`**, signature modifiers (**`with type`**)
- recursive modules, mixins

Implementation

- Split the path from OCaml to ZipML into small steps

User-surveys

- Embrace the human and sociological aspects of programming

Mechanization

- F^{ω} with kind functions and parametric kind polymorphism
- Semantic model of ZipML types (for abstraction safety)

Future works

Missing features

- The *easy* ones: `include`, `open`, 1st-class modules, `module type of`, signature modifiers (`with type`)
- recursive modules, mixins

Implementation

- Split the path from OCaml to ZipML into small steps

User-surveys

- Embrace the human and sociological aspects of programming

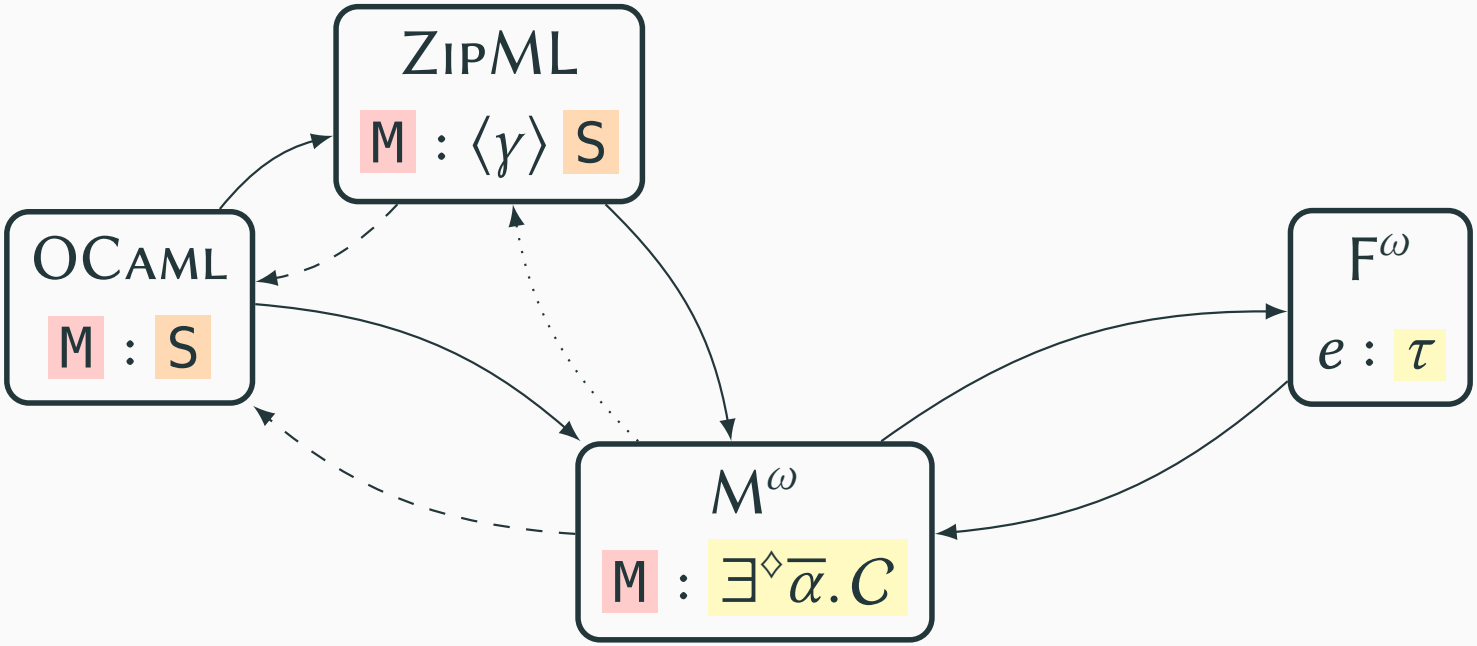
Mechanization

- F^{ω} with kind functions and parametric kind polymorphism
- Semantic model of ZipML types (for abstraction safety)

Research questions

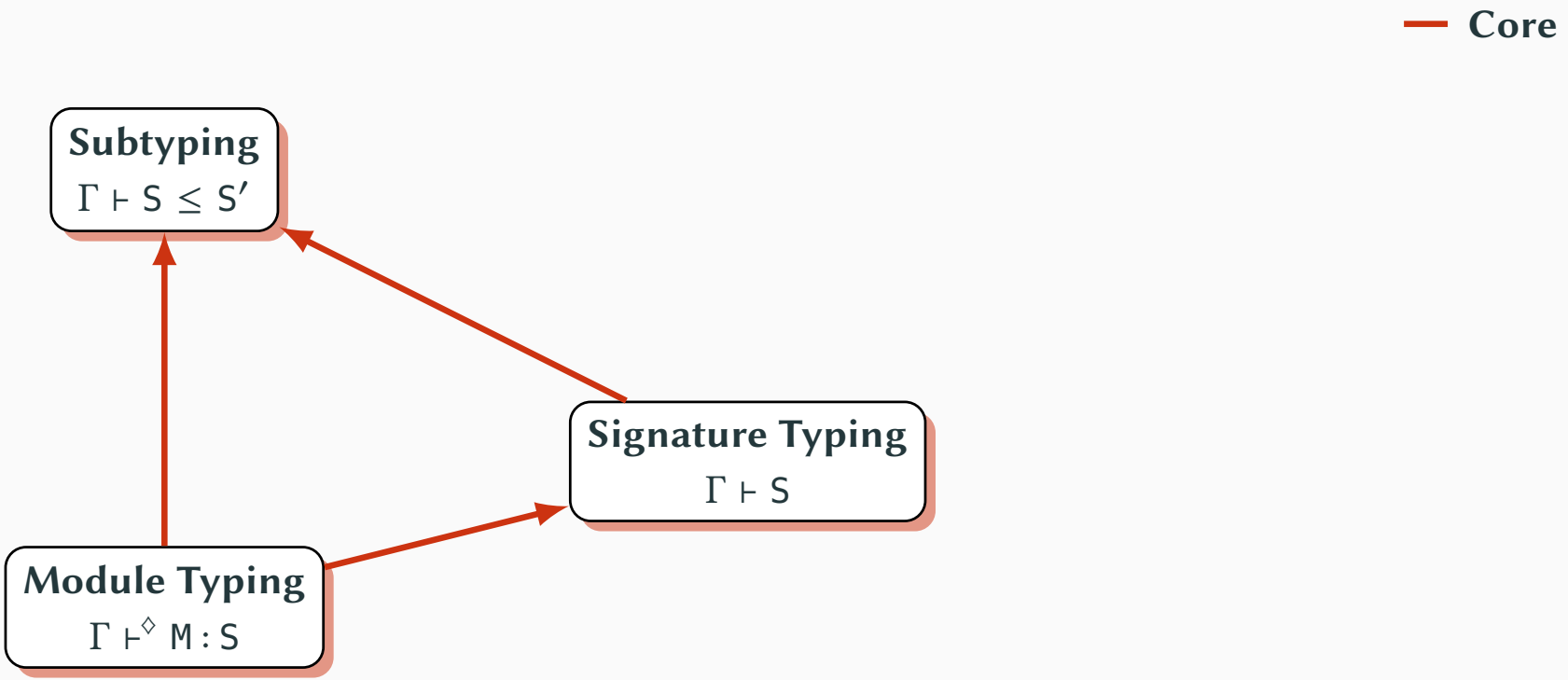
- More subtyping without loss of type-sharing
- Completeness of ZipML with full zippers
- Heuristics of zipper simplification

Questions ?

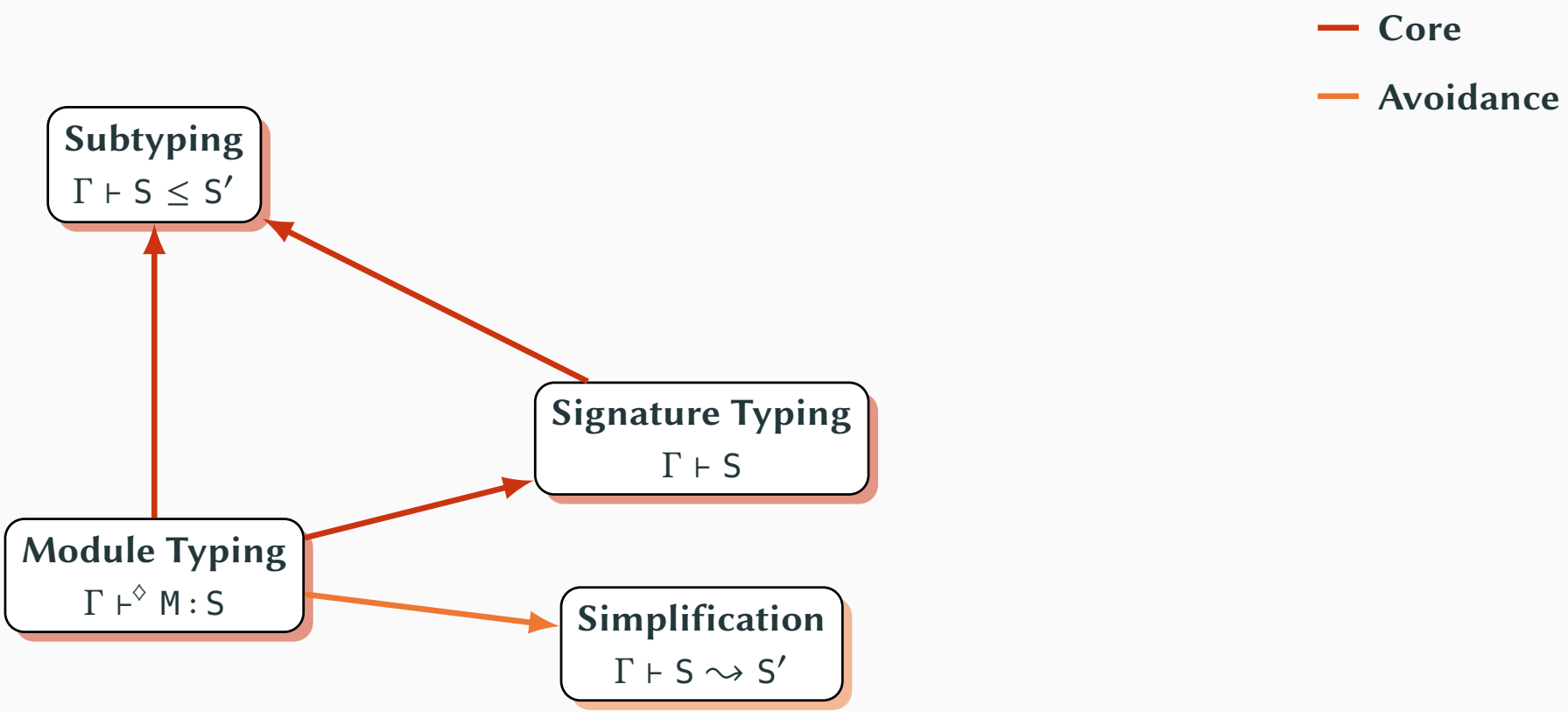


Backups

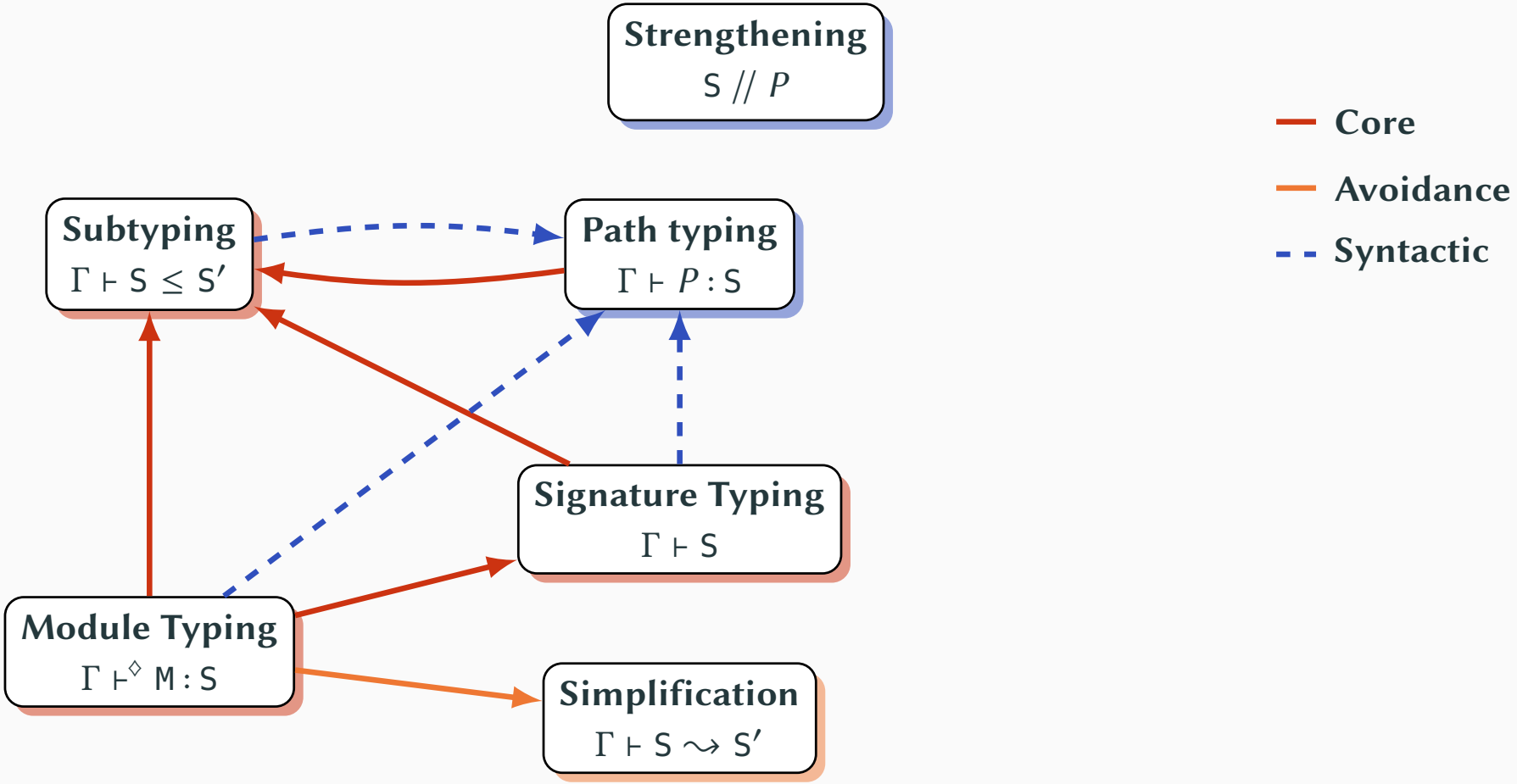
ZipML overview



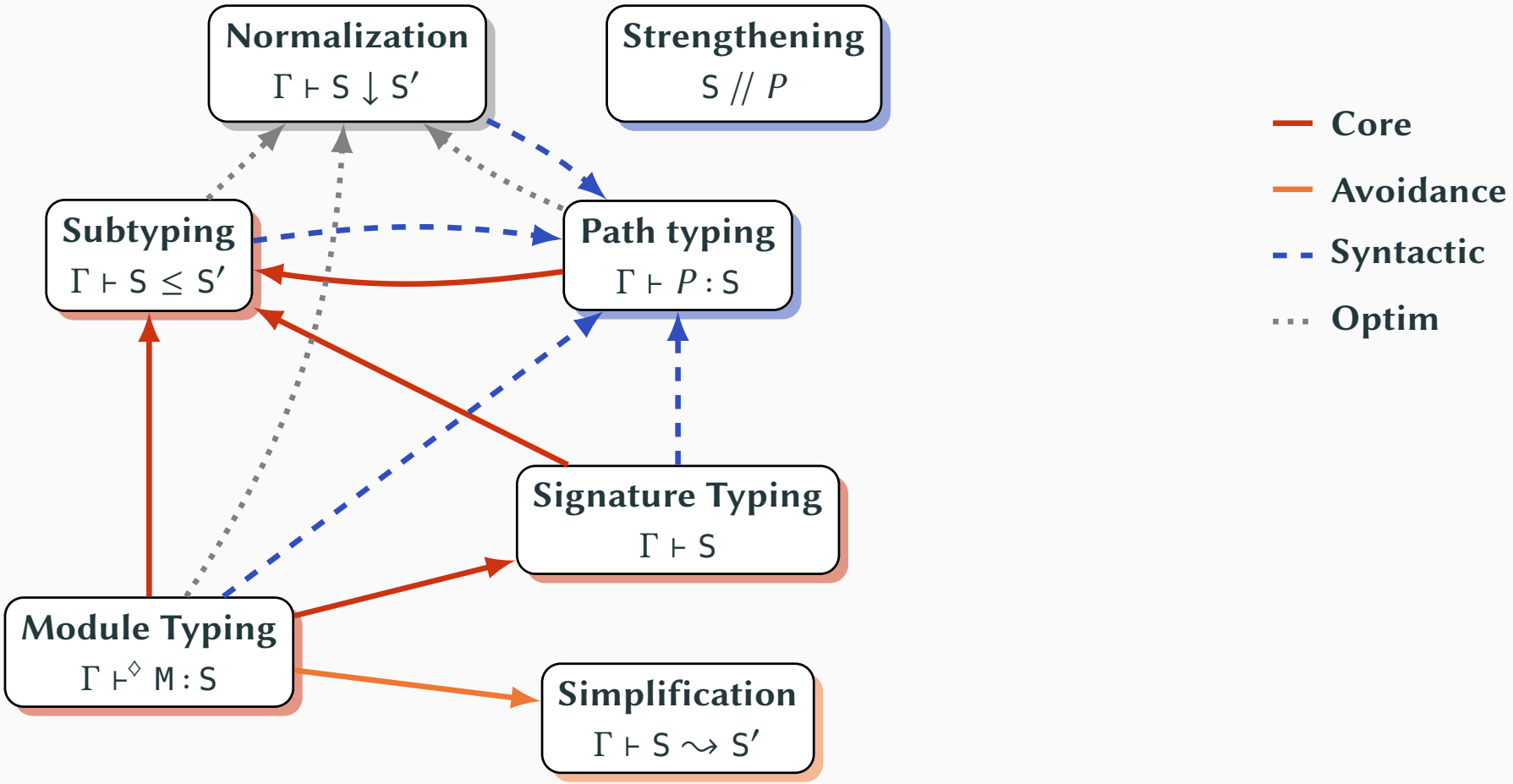
ZipML overview



ZipML overview



ZipML overview



Strengthening in syntactic module systems

```
1 module X  = struct type t end
2 module X' = X
```

```
module X : sig type t end
```

```
module X' : ?
```

Strengthening in syntactic module systems

```
1 module X  = struct type t end
2 module X' = X
```

```
module X : sig type t end
```

```
module X' : ?
```

- Usually, typing variables is easy:

$$\frac{\Gamma(X) = \mathsf{S}}{\Gamma \vdash X : \mathsf{S}}$$

Strengthening in syntactic module systems

```
1 module X  = struct type t end
2 module X' = X
```

```
module X : sig type t end
```

$$\Gamma = (\texttt{X : sig type t end})$$

```
module X' : sig type t end
```

- Usually, typing variables is easy:

$$\frac{\Gamma(X) = \mathsf{S}}{\Gamma \vdash X : \mathsf{S}}$$

Strengthening in syntactic module systems

```
1 module X  = struct type t end
2 module X' = X
```

```
module X : sig type t end
```

Γ = (X : **sig type** t **end**)

```
module X' : sig type t end
X.t and X'.t are not equal
```

- Usually, typing variables is easy:

$$\frac{\Gamma(X) = \mathsf{S}}{\Gamma \vdash X : \mathsf{S}} \times$$

$\longrightarrow$ Duplicating a signature may loose type-sharing

Strengthening in syntactic module systems

```
1 module X  = struct type t end
2 module X' = X
```

```
module X : sig type t end
```

Γ = (`X : sig type t end`)

```
module X' : sig type t end
X.t and X'.t are not equal
```

- Usually, typing variables is easy:

$$\frac{\Gamma(X) = \mathsf{S}}{\Gamma \vdash X : \mathsf{S}} \times$$

→ Duplicating a signature may loose type-sharing

- Need *strengthening* $\mathsf{S} /\!/ X$ to rewrite abstract type fields

Strengthening in syntactic module systems

```
1 module X  = struct type t end
2 module X' = X
```

```
module X : sig type t end
```

Γ = `(X : sig type t end)`

```
module X' : sig type t = X.t end
```

- Usually, typing variables is easy:

$$\frac{\Gamma(X) = \mathtt{S}}{\Gamma \vdash X : \mathtt{S}} \times$$

→ Duplicating a signature may loose type-sharing

- Need *strengthening* $\mathtt{S} \,/\!/\, X$ to rewrite abstract type fields

Strengthening in syntactic module systems

```
1 module X  = struct type t end
2 module X' = X
```

```
module X : sig type t end
```

Γ = `(X : sig type t end)`

```
module X' : sig type t = X.t end
```

- Usually, typing variables is easy:

$$\frac{\Gamma(X) = \mathsf{S}}{\Gamma \vdash X : \mathsf{S}} \times$$

→ Duplicating a signature may loose type-sharing

- Need *strengthening* $\mathsf{S} /\!/ X$ to rewrite abstract type fields

→ Strengthening when accessing variables makes the system irregular

Strengthening in syntactic module systems

```
1 module X  = struct type t end
2 module X' = X
```

```
module X : sig type t end
```

$\Gamma =$ `(X : sig type t = X.t end)`

```
module X' : sig type t = X.t end
```

- Usually, typing variables is easy:

$$\frac{\Gamma(X) = \mathsf{S}}{\Gamma \vdash X : \mathsf{S}} \times$$

→ Duplicating a signature may loose type-sharing

- Need *strengthening* $\mathsf{S} /\!/ X$ to rewrite abstract type fields

→ Strengthening when accessing variables makes the system irregular
→ ZipML uses *early* strengthening

Strengthening in syntactic module systems

```
1 module X  = struct type t end
2 module X' = X
```

```
module X : sig type t end
```

$\Gamma =$ `(X : sig type t = X.t end)`

```
module X' : sig type t = X.t end
```

- Usually, typing variables is easy:

$$\frac{\Gamma(X) = \mathsf{S}}{\Gamma \vdash X : \mathsf{S}} \times$$

→ Duplicating a signature may loose type-sharing

- Need *strengthening* $\mathsf{S} /\!/ X$ to rewrite abstract type fields

→ Strengthening when accessing variables makes the system irregular
→ ZipML uses *early* strengthening

→ Rewriting big signatures is costly (duplication + traversal)

Strengthening in syntactic module systems

```
1 module X  = struct type t end
2 module X' = X
```

```
module X : sig type t end
```

Γ = `(X : sig type t end // X)`

```
module X' : sig type t = X.t end
```

- Usually, typing variables is easy:

$$\frac{\Gamma(X) = \mathsf{S}}{\Gamma \vdash X : \mathsf{S}} \times$$

→ Duplicating a signature may loose type-sharing

- Need *strengthening* $\mathsf{S} \,//\, X$ to rewrite abstract type fields

→ Strengthening when accessing variables makes the system irregular
→ ZipML uses *early* strengthening

→ Rewriting big signatures is costly (duplication + traversal)
→ ZipML uses *lazy* strengthening

Module Magic

```
let module_magic : 'a → 'b =
  fun (x : 'a) →
    let module M = struct
      module rec N : sig val y : unit → 'b end = N
    end
    in M.N.y ()
```

```
val module_magic : 'a → 'b
```