

An Elaboration-Based Foundation for OCaml Recursive Modules

March 27, 2026

Litao Zhou

Joint work with Didier Rémy

Cambium, Inria Paris

On leave from the University of Hong Kong

Recursive Modules in OCaml

```
module rec X : sig .. (* external signature *) .. end
          = struct .. (* module body *) .. end
and Y : sig .. end = struct .. end
```

Recursive Modules in OCaml

```

1  module rec Tree : sig
2      type t
3      val head : t → t option
4  end = struct
5      type t = Leaf of int
6              | Node of Forest.t
7      let rec head = function
8          | Leaf _ as f → Some f
9          | Node f →
10             Option.bind (Forest.head f) head
11  end

```

```

12  and Forest : sig
13      type t
14      val head : t → Tree.t option
15  end = struct
16      type t = Tree.t list
17      let head = function
18          | [] → None
19          | h::_ → Some h
20  end

```

Recursive Modules in OCaml

```
1  module rec Tree : sig
2    type t
3    val head : t → t option
4  end = struct
5    type t = Leaf of int
6           | Node of Forest.t
7    let rec head = function
8      | Leaf _ as f → Some f
9      | Node f →
10         Option.bind (Forest.head f) head
11  end
```

```
12  and Forest : sig
13    type t
14    val head : t → Tree.t option
15  end = struct
16    type t = Tree.t list
17    let head = function
18      | [] → None
19      | h::_ → Some h
20  end
```

Type checking recursive modules can be subtle.

Recursive Modules in OCaml

```
1 module rec Tree : sig
2   type t
3   val head : t → t option
4 end = struct
5   type t = Leaf of int
6         | Node of Forest.t
7   let rec head (tr : t) : t option =
8     match tr with
9     | Leaf _ as f → Some f
10    | Node f →
11      match Forest.head f with
12      | None → None
13      | Some tr' → head tr'
14 end
```

(Option.bind is expanded for clarity)

```
1 and Forest : sig
2   type t
3   val head : t → Tree.t option
4 end = ...
```

Recursive Modules in OCaml

```
1 module rec Tree : sig
2   type t
3   val head : t → t option
4 end = struct
5   type t = Leaf of int
6         | Node of Forest.t
7   let rec head (tr : t) : t option =
8     match tr with
9     | Leaf _ as f → Some f
10    | Node f →
11      match Forest.head f with
12      | None → None
13      | Some tr' → head tr'
14 end
```

(Option.bind is expanded for clarity)

```
1 and Forest : sig
2   type t
3   val head : t → Tree.t option
4 end = ...
```

- Type checking Tree:

Recursive Modules in OCaml

```
1 module rec Tree : sig
2   type t
3   val head : t → t option
4 end = struct
5   type t = Leaf of int
6         | Node of Forest.t
7   let rec head (tr : t) : t option =
8     match tr with
9     | Leaf _ as f → Some f
10    | Node f →
11      match Forest.head f with
12      | None → None
13      | Some tr' → head tr'
14 end
```

(Option.bind is expanded for clarity)

```
1 and Forest : sig
2   type t
3   val head : t → Tree.t option
4 end = ...
```

- Type checking Tree:
 - Forest.head f returns Tree.t option.

Recursive Modules in OCaml

```

1  module rec Tree : sig
2    type t
3    val head : t → t option
4  end = struct
5    type t = Leaf of int
6          | Node of Forest.t
7    let rec head (tr : t) : t option =
8      match tr with
9      | Leaf _ as f → Some f
10     | Node f →
11       match Forest.head f with
12       | None → None
13       | Some tr' → head tr'
14  end

```

(Option.bind is expanded for clarity)

```

1  and Forest : sig
2    type t
3    val head : t → Tree.t option
4  end = ...

```

- Type checking `Tree`:
 - `Forest.head f` returns `Tree.t option`.
 - Within `Tree`, `head` expects `t → t option`.

Double Vision: type checker needs to know
external abstract type (`Tree.t`)
internal concrete definition (`t`)
 are the same.

OCaml Today: Works, but Fragile

- OCaml performs a strengthening on external signature to match internal definition

Example: Succeeds (Datatype)

```
1 module rec Tree : sig ... end = struct
2   type t = Leaf of int | Node of Forest.t
3   ...
4 end
```

OCaml Today: Works, but Fragile

- OCaml performs a strengthening on external signature to match internal definition
- Works for common cases, but the behavior is fragile: success relies on constructor boxing.

Example: Succeeds (Datatype)

```
1 module rec Tree : sig ... end = struct
2   type t = Leaf of int | Node of Forest.t
3   ...
4 end
```

Example: Fails (Alias)

```
1 module rec Tree : sig ... end = struct
2   type t' = Leaf of int | Node of Forest.t
3   type t = t'
4   let rec head = ...
5     Option.bind (Forest.head f) head
6 end
```

OCaml Today: Works, but Fragile

- OCaml performs a strengthening on external signature to match internal definition
- Works for common cases, but the behavior is fragile: success relies on constructor boxing.

Example: Succeeds (Datatype)

```
1 module rec Tree : sig ... end = struct
2   type t = Leaf of int | Node of Forest.t
3   ...
4 end
```

Example: Fails (Alias)

```
1 module rec Tree : sig ... end = struct
2   type t' = Leaf of int | Node of Forest.t
3   type t = t'
4   let rec head = ...
5     Option.bind (Forest.head f) head
6 end
```

OCaml Error

The value head has type `t -> t option` but an expression was expected of type `Tree.t -> 'a option`.
Type `t` is not compatible with type `Tree.t`

Another Use Case: Bootstrapped Heaps (Okasaki 1998)

```
1 module type ORDERED = sig type t val compare : t → t → int end
2 module type HEAP    = sig type item   type heap
3                       val insert : item → heap → heap ... end
```

A functor `MkHeap` builds a `HEAP` module from an `ORDERED` module.

```
1 module MkHeap (X : ORDERED) : HEAP
2 = struct
3   type item = X.t
4   type heap = Empty
5           | Heap of item * heap
6   let insert x h = ...
7 end
```

Another Use Case: Bootstrapped Heaps (Okasaki 1998)

```
1 module type ORDERED = sig type t val compare : t → t → int end
2 module type HEAP    = sig type item   type heap
3                       val insert : item → heap → heap ... end
```

A functor `MkHeap` builds a `HEAP` module from an `ORDERED` module.

```
1 module MkHeap (X : ORDERED) : HEAP
2 = struct
3   type item = X.t
4   type heap = Empty
5             | Heap of item * heap
6   let insert x h = ...
7 end
```

With recursive modules, it is possible to define **bootstrapped** structures:

```
8 module rec Boot : ORDERED = struct
9   type t = Heap.heap
10  let compare = ... end
11 and Heap : HEAP with type item := Boot.t
12   = MkHeap(Boot)
```

A `Heap` module whose `item` is `heap` itself.

Applicative & Generative Functors

Applicative Functor

- Used in OCAML (default for functors).

Generative Functor

- Used in SML; expressed via `()`-argument in OCAML.

Applicative & Generative Functors

Applicative Functor

- Used in OCAML (default for functors).
- Applying to equal arguments preserves type identity.
- $F(A).t = F(B).t$ whenever $A = B$.

Generative Functor

- Used in SML; expressed via `()`-argument in OCAML.

Applicative & Generative Functors

Applicative Functor

- Used in OCAML (default for functors).
- Applying to equal arguments preserves type identity.
- $F(A).t = F(B).t$ whenever $A = B$.

Generative Functor

- Used in SML; expressed via $()$ -argument in OCAML.
- Each application creates a **fresh** abstract type.
- $G().t \neq G().t$ on two distinct calls.

Applicative & Generative Functors

Applicative Functor

- Used in OCAML (default for functors).
- Applying to equal arguments preserves type identity.
- $F(A).t = F(B).t$ whenever $A = B$.

Generative Functor

- Used in SML; expressed via $()$ -argument in OCAML.
- Each application creates a **fresh** abstract type.
- $G().t \neq G().t$ on two distinct calls.

Prior formal treatments of recursive modules focus on **generative** functors; applicative functors are left unstudied in the context of recursion.

Introduction: Summary

Recursive Modules interact non-trivially with

- **type abstraction**
- **double vision** (partially solved by OCaml)
- **functors** (applicative & generative).

Introduction: Summary

Recursive Modules interact non-trivially with

- **type abstraction**
- **double vision** (partially solved by OCaml)
- **functors** (applicative & generative).

The Goal

Provide a principle foundation for OCaml recursive modules that handles both **generative** and **applicative** functors while resolving **double vision** in practice.

Formal Foundations for OCaml Modules

- Recent efforts have provided formal foundations for OCaml's module system (without recursive modules):
 - **M^ω** (Blaudeau et al. 2024): An elaboration-based approach (F-ING modules).
 - **ZipML** (Blaudeau et al. 2025): A path-based, syntactic approach.

Formal Foundations for OCaml Modules

- Recent efforts have provided formal foundations for OCaml's module system (without recursive modules):
 - **M^ω** (Blaudeau et al. 2024): An elaboration-based approach (F-ING modules).
 - **ZipML** (Blaudeau et al. 2025): A path-based, syntactic approach.
- *Note for this talk:* We build entirely on the **M^ω** foundation.
- F-ING modules: Elaborate source ML modules into the typed core language System F^ω.

Formal Foundations for OCaml Modules

- Recent efforts have provided formal foundations for OCaml's module system (without recursive modules):
 - **M^ω** (Blaudeau et al. 2024): An elaboration-based approach (F-ING modules).
 - **ZipML** (Blaudeau et al. 2025): A path-based, syntactic approach.
- *Note for this talk:* We build entirely on the **M^ω** foundation.
- F-ING modules: Elaborate source ML modules into the typed core language System F^ω.
 - Module **signatures** \rightsquigarrow F^ω **types** Module **expressions** \rightsquigarrow F^ω **terms**

Formal Foundations for OCaml Modules

- Recent efforts have provided formal foundations for OCaml's module system (without recursive modules):
 - **M^ω** (Blaudeau et al. 2024): An elaboration-based approach (F-ING modules).
 - **ZipML** (Blaudeau et al. 2025): A path-based, syntactic approach.
- *Note for this talk:* We build entirely on the **M^ω** foundation.
- F-ING modules: Elaborate source ML modules into the typed core language System F^ω.
 - Module **signatures** \rightsquigarrow F^ω **types** Module **expressions** \rightsquigarrow F^ω **terms**
 - Type abstraction \rightsquigarrow **existential types** Structural signatures \rightsquigarrow record types

Formal Foundations for OCaml Modules

- Recent efforts have provided formal foundations for OCaml's module system (without recursive modules):
 - \mathbf{M}^ω (Blaudeau et al. 2024): An elaboration-based approach (F-ING modules).
 - ZipML** (Blaudeau et al. 2025): A path-based, syntactic approach.
- Note for this talk:* We build entirely on the \mathbf{M}^ω foundation.
- F-ING modules: Elaborate source ML modules into the typed core language System F^ω .
 - Module **signatures** $\rightsquigarrow F^\omega$ **types** Module **expressions** $\rightsquigarrow F^\omega$ **terms**
 - Type abstraction \rightsquigarrow **existential types** Structural signatures \rightsquigarrow record types
 - Type fields \rightsquigarrow dummy identity functions

$$\text{type } t = (\tau : \kappa) \rightsquigarrow \{l_t = \Lambda \varphi . \lambda(x : \varphi \tau) . x : \forall(\varphi : \kappa \rightarrow \star) . \varphi \tau \rightarrow \varphi \tau\}$$

Formal Foundations for OCaml Modules

- Recent efforts have provided formal foundations for OCaml's module system (without recursive modules):
 - M^ω** (Blaudeau et al. 2024): An elaboration-based approach (F-ING modules).
 - ZipML** (Blaudeau et al. 2025): A path-based, syntactic approach.
- Note for this talk:* We build entirely on the **M^ω** foundation.
- F-ING modules: Elaborate source ML modules into the typed core language System F^ω.
 - Module **signatures** \rightsquigarrow F^ω **types** Module **expressions** \rightsquigarrow F^ω **terms**
 - Type abstraction \rightsquigarrow **existential types** Structural signatures \rightsquigarrow record types
 - Type fields \rightsquigarrow dummy identity functions

$$\text{type } t = (\tau : \kappa) \rightsquigarrow \{ \ell_t = \Lambda \varphi . \lambda (x : \varphi \tau) . x : \forall (\varphi : \kappa \rightarrow \star) . \varphi \tau \rightarrow \varphi \tau \}$$

- Soundness & Decidability** come for free from the well-understood meta-theory of core F^ω. No bespoke module-level proof needed.

M^ω : a type system for OCAML modules

Every abstract type field becomes an existential variable, **extruded** to the front of the enclosing signature.

```
1 sig
2   type t           (* abstract *)
3   module X : sig
4     type u         (* abstract *)
5   end
6   val f : t → X.u
7 end
```

M^ω : a type system for OCAML modules

Every abstract type field becomes an existential variable, **extruded** to the front of the enclosing signature.

```
1 sig
2   type t                (* abstract *)
3   module X : sig
4     type u              (* abstract *)
5   end
6   val f : t → X.u
7 end
```

M^ω signature

α (for `t`), β (for `X.u`) are both lifted to the front:

$$\lambda\alpha\beta. \left\{ \begin{array}{l} \text{type } t = \alpha \\ \text{module } X : \{\text{type } u = \beta\} \\ \text{val } f : \alpha \rightarrow \beta \end{array} \right\}$$

M^ω : a type system for OCAML modules

Every abstract type field becomes an existential variable, **extruded** to the front of the enclosing signature.

```

1  sig
2    type t                (* abstract *)
3    module X : sig
4      type u              (* abstract *)
5    end
6    val f : t → X.u
7  end

```

M^ω signature

α (for `t`), β (for `X.u`) are both lifted to the front:

$$\lambda \alpha \beta. \left\{ \begin{array}{l} \text{type } t = \alpha \\ \text{module } X : \{\text{type } u = \beta\} \\ \text{val } f : \alpha \rightarrow \beta \end{array} \right\}$$

Note, M^ω signatures are just syntactic sugars for F^ω types:

$$\lambda \alpha \beta. \left\{ \begin{array}{l} \ell_t : \forall \phi. \phi \alpha \rightarrow \phi \alpha \\ \ell_X : \{\ell_u : \forall \phi. \phi \beta \rightarrow \phi \beta\} \\ \ell_f : \alpha \rightarrow \beta \end{array} \right\}$$

M^ω : Abstract Types as Top-Level Quantifiers

Generative ∇ functors keep abstract variables inside the functor body $() \rightarrow \exists \nabla \bar{\beta}. \mathcal{C}$

Example: Generative functor

```
1 module G ()  
2   : sig type t end = struct  
3   type t = int ref
```

M^ω : Abstract Types as Top-Level Quantifiers

Generative ∇ functors keep abstract variables inside the functor body $() \rightarrow \exists \nabla \bar{\beta}. \mathcal{C}$

Example: Generative functor

```
1 module G ()  
2   : sig type t end = struct  
3   type t = int ref
```

M^ω signature for G:

$G : () \rightarrow \exists \nabla \bar{\beta}. \{\text{type } t = \beta\}$

M^ω : Abstract Types as Top-Level Quantifiers

Generative ∇ functors keep abstract variables inside the functor body $() \rightarrow \exists \nabla \bar{\beta}. \mathcal{C}$

Example: Generative functor

```
1 module G ()  
2   : sig type t end = struct  
3   type t = int ref
```

M^ω signature for G:

$$G : () \rightarrow \exists \nabla \beta. \{\text{type } t = \beta\}$$

β stays **inside** $\exists \nabla$ (generative):

each $G()$ produces a fresh, incompatible type

M^ω : Abstract Types as Top-Level Quantifiers

Generative ∇ functors keep abstract variables inside the functor body $() \rightarrow \exists^\nabla \bar{\beta}. \mathcal{C}$

Applicative ∇ functors **lift** abstract variables to higher-order $\forall \bar{\alpha}. \mathcal{C}_a \rightarrow \mathcal{C}$

Example: Generative functor

```

1 module G ()
2     : sig type t end = struct
3   type t = int ref

```

Example: Applicative functor

```

1 module F (Y : sig type u end)
2     : sig type t end = struct
3   type t = Y.u * Y.u

```

M^ω signature for G:

$$G : () \rightarrow \exists^\nabla \beta. \{\text{type } t = \beta\}$$

β stays **inside** \exists^∇ (generative):

each $G()$ produces a fresh, incompatible type

M^ω : Abstract Types as Top-Level Quantifiers

Generative ∇ functors keep abstract variables inside the functor body $() \rightarrow \exists^\nabla \bar{\beta}. \mathcal{C}$

Applicative ∇ functors **lift** abstract variables to higher-order $\forall \bar{\alpha}. \mathcal{C}_a \rightarrow \mathcal{C}$

Example: Generative functor

```

1 module G ()
2       : sig type t end = struct
3 type t = int ref

```

M^ω signature for G :

$$G : () \rightarrow \exists^\nabla \beta. \{\text{type } t = \beta\}$$

β stays **inside** \exists^∇ (generative):

each $G()$ produces a fresh, incompatible type

Example: Applicative functor

```

1 module F (Y : sig type u end)
2       : sig type t end = struct
3 type t = Y.u * Y.u

```

M^ω signature for F :

$$F : \exists^\nabla \beta. \forall \alpha. \{\text{type } t = \alpha\} \rightarrow \{\text{type } u = \beta(\alpha)\}$$

M^ω : Abstract Types as Top-Level Quantifiers

Generative ∇ functors keep abstract variables inside the functor body $() \rightarrow \exists^\nabla \bar{\beta}. \mathcal{C}$

Applicative ∇ functors **lift** abstract variables to higher-order $\forall \bar{\alpha}. \mathcal{C}_a \rightarrow \mathcal{C}$

Example: Generative functor

```

1 module G ()
2       : sig type t end = struct
3 type t = int ref

```

M^ω signature for G :

$$G : () \rightarrow \exists^\nabla \beta. \{\text{type } t = \beta\}$$

β stays **inside** \exists^∇ (generative):
each $G()$ produces a fresh, incompatible type

Example: Applicative functor

```

1 module F (Y : sig type u end)
2       : sig type t end = struct
3 type t = Y.u * Y.u

```

M^ω signature for F :

$$F : \exists^\nabla \beta. \forall \alpha. \{\text{type } t = \alpha\} \rightarrow \{\text{type } u = \beta(\alpha)\}$$

$\beta : \star \rightarrow \star$ is **skolemized** out (applicative):
 $F(A).t = F(B).t$ when $A = B$

Elaborating applicative functors: transparent existentials

Elaborating applicative functor requires lifting $\mathcal{C}_a \rightarrow \exists\beta.\tau$ into $\exists\beta.\mathcal{C}_a \rightarrow \tau$.

Elaborating applicative functors: transparent existentials

Elaborating applicative functor requires lifting $\mathcal{C}_a \rightarrow \exists\beta.\tau$ into $\exists\beta.\mathcal{C}_a \rightarrow \tau$.

- Lifting assumes a single uniform witness, while standard existentials fail to enforce this:

```

λ(x : bool).if x
  then pack ⟨int, 42⟩ as ∃∇β. β      : bool → ∃∇β.β  ✗  ∃∇β.bool → β
  else pack ⟨bool, true⟩ as ∃∇β. β

```

Elaborating applicative functors: transparent existentials

Elaborating applicative functor requires lifting $\mathcal{C}_a \rightarrow \exists\beta.\tau$ into $\exists\beta.\mathcal{C}_a \rightarrow \tau$.

- Lifting assumes a single uniform witness, while standard existentials fail to enforce this:

```

λ(x : bool).if x
  then pack ⟨int, 42⟩ as ∃∇β. β      : bool → ∃∇β.β  ✗  ∃∇β.bool → β
  else pack ⟨bool, true⟩ as ∃∇β. β

```

- Unless created by generative application, modules always have a unique witness

Elaborating applicative functors: transparent existentials

Elaborating applicative functor requires lifting $\mathcal{C}_a \rightarrow \exists\beta.\tau$ into $\exists\beta.\mathcal{C}_a \rightarrow \tau$.

- Lifting assumes a single uniform witness, while standard existentials fail to enforce this:

```

λ(x : bool).if x
  then pack ⟨int, 42⟩ as ∃∇β. β      : bool → ∃∇β.β  ✗  ∃∇β.bool → β
  else pack ⟨bool, true⟩ as ∃∇β. β

```

- Unless created by generative application, modules always have a unique witness
- **The Solution in M^ω : Transparent existentials (\exists^∇).**
 - Existential types $(\exists^{\nabla\tau}(\beta). \mathcal{C})$ keep track of the witness τ

Elaborating applicative functors: transparent existentials

Elaborating applicative functor requires lifting $\mathcal{C}_a \rightarrow \exists\beta.\tau$ into $\exists\beta.\mathcal{C}_a \rightarrow \tau$.

- Lifting assumes a single uniform witness, while standard existentials fail to enforce this:

$$\begin{array}{l} \lambda(x : \text{bool}).\text{if } x \\ \text{then pack } \langle \text{int}, 42 \rangle \text{ as } \exists^\nabla\beta. \beta \quad : \text{bool} \rightarrow \exists^\nabla\beta. \beta \quad \not\rightarrow \quad \exists^\nabla\beta. \text{bool} \rightarrow \beta \\ \text{else pack } \langle \text{bool}, \text{true} \rangle \text{ as } \exists^\nabla\beta. \beta \end{array}$$

- Unless created by generative application, modules always have a unique witness
- **The Solution in M^ω : Transparent existentials (\exists^∇).**
 - Existential types $(\exists^\nabla\tau(\beta). \mathcal{C})$ keep track of the witness τ
 - Lifting is sound for \exists^∇ because the witness is checked as part of type equality

Elaborating applicative functors: transparent existentials

Elaborating applicative functor requires lifting $\mathcal{C}_a \rightarrow \exists\beta.\tau$ into $\exists\beta.\mathcal{C}_a \rightarrow \tau$.

- Lifting assumes a single uniform witness, while standard existentials fail to enforce this:

```

λ(x : bool).if x
  then pack ⟨int, 42⟩ as ∃∇β. β      : bool → ∃∇β.β  ✗  ∃∇β.bool → β
  else pack ⟨bool, true⟩ as ∃∇β. β

```

- Unless created by generative application, modules always have a unique witness
- **The Solution in M^ω : Transparent existentials (\exists^∇).**
 - Existential types $(\exists^\nabla\tau(\beta).\mathcal{C})$ keep track of the witness τ
 - Lifting is sound for \exists^∇ because the witness is checked as part of type equality
 - Primitives pack^∇ , repack^∇ , $\text{lift}_{\rightarrow}$, ... are encoded as a library in F^ω , sound by construction.

Elaborating applicative functors: transparent existentials

Elaborating applicative functor requires lifting $\mathcal{C}_a \rightarrow \exists\beta.\tau$ into $\exists\beta.\mathcal{C}_a \rightarrow \tau$.

- Lifting assumes a single uniform witness, while standard existentials fail to enforce this:

$$\begin{aligned} &\lambda(x : \text{bool}).\text{if } x \\ &\text{then pack } \langle \text{int}, 42 \rangle \text{ as } \exists^\nabla\beta.\beta \quad : \text{bool} \rightarrow \exists^\nabla\beta.\beta \quad \not\rightarrow \quad \exists^\nabla\beta.\text{bool} \rightarrow \beta \\ &\text{else pack } \langle \text{bool}, \text{true} \rangle \text{ as } \exists^\nabla\beta.\beta \end{aligned}$$

- Unless created by generative application, modules always have a unique witness
- **The Solution in M^ω : Transparent existentials (\exists^∇).**
 - Existential types $(\exists^\nabla\tau(\beta).\mathcal{C})$ keep track of the witness τ
 - Lifting is sound for \exists^∇ because the witness is checked as part of type equality
 - Primitives pack^∇ , repack^∇ , $\text{lift}_{\rightarrow}$, ... are encoded as a library in F^ω , sound by construction.
 - Transparent existentials can be turned into opaque ones via `seal`.

The Elaboration Judgment & Functor Modes in M^ω

Type of a module expression M is an existential signature $\exists^{\bar{\vartheta}} \bar{\alpha}. \mathcal{C}$ where $\vartheta := \nabla \tau \mid \blacktriangledown$

$\bar{\vartheta}$ is required to be homogeneous in \diamond (all \blacktriangledown / all $\nabla \tau$)

$$\Gamma \vdash M : \exists^{\bar{\vartheta}} \bar{\alpha}. \mathcal{C} \rightsquigarrow e$$

The Elaboration Judgment & Functor Modes in M^ω

Type of a module expression M is an existential signature $\exists^{\bar{\vartheta}} \bar{\alpha}. \mathcal{C}$ where $\vartheta := \nabla \tau \mid \blacktriangledown$

$\bar{\vartheta}$ is required to be homogeneous in \diamond (all \blacktriangledown / all $\nabla \tau$)

$$\Gamma \vdash M : \exists^{\bar{\vartheta}} \bar{\alpha}. \mathcal{C} \rightsquigarrow e$$

E-Typ-Mod-GenFct

$$\frac{\Gamma \vdash M : \exists^{\nabla} \bar{\alpha}. \mathcal{C} \rightsquigarrow e}{\Gamma \vdash () \rightarrow M : () \rightarrow \exists^{\nabla} \bar{\alpha}. \mathcal{C} \rightsquigarrow \lambda(_ : ()).e}$$

E-Typ-Mod-Seal

$$\frac{\Gamma \vdash M : \exists^{\nabla \bar{\tau}}(\bar{\alpha}). \mathcal{C} \rightsquigarrow e}{\Gamma \vdash M : \exists^{\nabla} \bar{\alpha}. \mathcal{C} \rightsquigarrow \text{seal}^{|\bar{\alpha}|} e}$$

E-Typ-Mod-Ascr

$$\frac{\Gamma \vdash S : \lambda \bar{\alpha}. \mathcal{C} \quad \Gamma \vdash P : \mathcal{C}' \rightsquigarrow e \quad \Gamma \vdash \mathcal{C}' \prec: \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow f}{\Gamma \vdash (P : S) : \exists^{\nabla \bar{\tau}}(\bar{\alpha}). \mathcal{C} \rightsquigarrow \text{pack}^{\nabla} \langle \bar{\tau}, f e \rangle \text{ as } \exists^{\nabla \bar{\tau}}(\bar{\alpha}). \mathcal{C}} \quad \dots$$

Target Calculus: $F_{\mu}^{\omega} + \exists^{\nabla}$

To support cyclic type definitions we elaborate to F^{ω} extended with **equi-recursive types** $\mu(\alpha : \kappa). \tau$ at any kind κ .

- Recursive types are equal to their infinite unfoldings, we have $\mu\alpha. \tau \equiv \tau[\alpha \mapsto \mu\alpha. \tau]$
- Higher kinded recursion is necessary due to extrusion of abstract variables.

Target Calculus: $F_{\mu}^{\omega} + \exists^{\nabla}$

To support cyclic type definitions we elaborate to F^{ω} extended with **equi-recursive types** $\mu(\alpha : \kappa). \tau$ at any kind κ .

- Recursive types are equal to their infinite unfoldings, we have $\mu\alpha. \tau \equiv \tau[\alpha \mapsto \mu\alpha. \tau]$
- Higher kinded recursion is necessary due to extrusion of abstract variables.
- Well studied meta-theory (Cai et al. 2016):
 - **Type soundness** for recursive types at arbitrary kinds.

Target Calculus: $F_{\mu}^{\omega} + \exists^{\nabla}$

To support cyclic type definitions we elaborate to F^{ω} extended with **equi-recursive types** $\mu(\alpha : \kappa). \tau$ at any kind κ .

- Recursive types are equal to their infinite unfoldings, we have $\mu\alpha. \tau \equiv \tau[\alpha \mapsto \mu\alpha. \tau]$
- Higher kinded recursion is necessary due to extrusion of abstract variables.
- Well studied meta-theory (Cai et al. 2016):
 - **Type soundness** for recursive types at arbitrary kinds.
 - **Decidable type checking** algorithm for recursive types as base kind \star .

Target Calculus: $F_{\mu}^{\omega} + \exists^{\nabla}$

To support cyclic type definitions we elaborate to F^{ω} extended with **equi-recursive types** $\mu(\alpha : \kappa). \tau$ at any kind κ .

- Recursive types are equal to their infinite unfoldings, we have $\mu\alpha. \tau \equiv \tau[\alpha \mapsto \mu\alpha. \tau]$
- Higher kinded recursion is necessary due to extrusion of abstract variables.
- Well studied meta-theory (Cai et al. 2016):
 - **Type soundness** for recursive types at arbitrary kinds.
 - **Decidable type checking** algorithm for recursive types as base kind \star .
 - **Contractiveness** condition to avoid non-productive cycles
(e.g. $\mu\alpha. \alpha$ are valid in syntax, but checked semantically, considered equivalent to \perp)

Target Calculus: $F_{\mu}^{\omega} + \exists^{\nabla}$

To support cyclic type definitions we elaborate to F^{ω} extended with **equi-recursive types** $\mu(\alpha : \kappa). \tau$ at any kind κ .

- Recursive types are equal to their infinite unfoldings, we have $\mu\alpha. \tau \equiv \tau[\alpha \mapsto \mu\alpha. \tau]$
- Higher kinded recursion is necessary due to extrusion of abstract variables.
- Well studied meta-theory (Cai et al. 2016):
 - **Type soundness** for recursive types at arbitrary kinds.
 - **Decidable type checking** algorithm for recursive types as base kind \star .
 - **Contractiveness** condition to avoid non-productive cycles
(e.g. $\mu\alpha. \alpha$ are valid in syntax, but checked semantically, considered equivalent to \perp)
- We use transparent existentials \exists^{∇} to support applicative functors, but indeed they are encodable in F^{ω} , so F_{μ}^{ω} suffices as the target.

Type Checking Recursive Modules

Without double vision, a recursive module $\text{rec } X : S = M$ can be understood as the fixed point of a functor **functor** $(X : S) \rightarrow M$.

Opaque mode is straightforward:

E-Typ-Rec-Opaque

$$\frac{\Gamma \vdash \text{rec } (X : S) : \lambda \bar{\alpha}. \mathcal{C} \quad \Gamma, \bar{\alpha}, X : \mathcal{C} \vdash M : \exists \nabla \bar{\beta}. \mathcal{C}' \rightsquigarrow e \quad \Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec : \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow f}{\Gamma \vdash \text{rec } X : S = M : \exists \nabla \bar{\alpha}. \mathcal{C} \rightsquigarrow \text{fix}(\dots)}$$

Type Checking Recursive Modules

Without double vision, a recursive module $\text{rec } X : S = M$ can be understood as the fixed point of a functor **functor** $(X : S) \rightarrow M$.

Opaque mode is straightforward:

E-Typ-Rec-Opaque

$$\frac{\Gamma \vdash \text{rec } (X : S) : \lambda \bar{\alpha}. \mathcal{C} \quad \Gamma, \bar{\alpha}, X : \mathcal{C} \vdash M : \exists \nabla \bar{\beta}. \mathcal{C}' \rightsquigarrow e \quad \Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec: \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow f}{\Gamma \vdash \text{rec } X : S = M : \exists \nabla \bar{\alpha}. \mathcal{C} \rightsquigarrow \text{fix}(\dots)}$$

Step 1 — Signature

Elaborate $\text{rec } (X : S)$ to get shape \mathcal{C} with abstract vars $\bar{\alpha}$.

Step 1 — External Signature: Shape & Abstract Variables

$$\Gamma \vdash \text{rec } (X : S) : \lambda \bar{\alpha}. \mathcal{C}$$

The external signature elaborates to $\lambda \bar{\alpha}. \mathcal{C}$, where $\bar{\alpha}$ are the *abstract variables* it introduces.

```
1 module rec X : sig
2   type t      (* abstract *)
3   type u = int * X.u
4   (* transparent, cyclic *)
5   val f : u → t
6 end = ...
```

Step 1 — External Signature: Shape & Abstract Variables

$$\Gamma \vdash \text{rec } (X : S) : \lambda \bar{\alpha}. \mathcal{C}$$

The external signature elaborates to $\lambda \bar{\alpha}. \mathcal{C}$, where $\bar{\alpha}$ are the *abstract variables* it introduces.

```

1  module rec X : sig
2    type t      (* abstract *)
3    type u = int * X.u
4    (* transparent, cyclic *)
5    val f : u → t
6  end = ...

```

Abstract type `t`
 \rightsquigarrow fresh variable α

Transparent type `u`
 \rightsquigarrow concrete $\mu\beta. \text{int} \times \beta$
 (by approximation + iteration)

Step 1 — External Signature: Shape & Abstract Variables

$$\Gamma \vdash \text{rec } (X : S) : \lambda \bar{\alpha}. \mathcal{C}$$

The external signature elaborates to $\lambda \bar{\alpha}. \mathcal{C}$, where $\bar{\alpha}$ are the *abstract variables* it introduces.

```

1 module rec X : sig
2   type t      (* abstract *)
3   type u = int * X.u
4   (* transparent, cyclic *)
5   val f : u → t
6 end = ...

```

Abstract type t
 \rightsquigarrow fresh variable α

Transparent type u
 \rightsquigarrow concrete $\mu\beta. \text{int} \times \beta$
 (by approximation + iteration)

```

λ α. sig
  type t = α
  type u = μ β. int × β
  val f : (μ β. int × β) → α
end

```

Step 1 — External Signature: Shape & Abstract Variables

$$\Gamma \vdash \text{rec } (X : S) : \lambda \bar{\alpha}. \mathcal{C}$$

The external signature elaborates to $\lambda \bar{\alpha}. \mathcal{C}$, where $\bar{\alpha}$ are the *abstract variables* it introduces.

```

1  module rec X : sig
2    type t      (* abstract *)
3    type u = int * X.u
4    (* transparent, cyclic *)
5    val f : u → t
6  end = ...

```

Abstract type t
 \rightsquigarrow fresh variable α

Transparent type u
 \rightsquigarrow concrete $\mu\beta. \text{int} \times \beta$
 (by approximation + iteration)

```

λ α. sig
  type t = α
  type u = μ β. int × β
  val f : (μ β. int × β) → α
end

```

Cyclic types are resolved in core F_{μ}^{ω} , no extension needed in M_{μ}^{ω} signature language \mathcal{C} !

Type Checking Recursive Modules

Without double vision, a recursive module $\text{rec } X : S = M$ can be understood as the fixed point of a functor **functor** $(X : S) \rightarrow M$.

Opaque mode is straightforward:

E-Typ-Rec-Opaque

$$\frac{\Gamma \vdash \text{rec } (X : S) : \lambda \bar{\alpha}. \mathcal{C} \quad \Gamma, \bar{\alpha}, X : \mathcal{C} \vdash M : \exists \nabla \bar{\beta}. \mathcal{C}' \rightsquigarrow e \quad \Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec : \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow f}{\Gamma \vdash \text{rec } X : S = M : \exists \nabla \bar{\alpha}. \mathcal{C} \rightsquigarrow \text{fix } (\dots)}$$

Step 1 — Signature

Elaborate $\text{rec } (X : S)$ to get shape \mathcal{C} with abstract vars $\bar{\alpha}$.

Step 2 — Body

Check M with $X : \mathcal{C}$ in scope (the recursive assumption). Body may introduce fresh vars $\bar{\beta}$.

Step 2 — Body Checking: Under the Recursive Assumption

$$\Gamma, \bar{\alpha}, X : \mathcal{C} \vdash M : \exists^{\diamond} \bar{\beta}. C' \rightsquigarrow e$$

```
1  module rec X : sig
2    type t
3    type u = int * X.u
4    val f : u → t
5  end = struct
6    (* G : () → sig type h end *)
7    module H = G()
8    (* t implemented via H *)
9    type t = H.h * X.t
10   type u = int * X.u
11   let f (x : u) : t = ...
12 end
```

Step 2 — Body Checking: Under the Recursive Assumption

$$\Gamma, \bar{\alpha}, X : \mathcal{C} \vdash M : \exists^{\diamond} \bar{\beta}. C' \rightsquigarrow e$$

```

1  module rec X : sig
2    type t
3    type u = int * X.u
4    val f : u → t
5  end = struct
6    (* G : () → sig type h end *)
7    module H = G()
8    (* t implemented via H *)
9    type t = H.h * X.t
10   type u = int * X.u
11   let f (x : u) : t = ...
12 end

```

```

 $\exists^{\nabla} \bar{\beta}. \text{sig}$ 
  module H : sig type h =  $\beta$  end
  type t =  $\beta \times \alpha$ 
  type u = int × ( $\mu\gamma. \text{int} \times \gamma$ )
  val f : (int × ( $\mu\gamma. \text{int} \times \gamma$ )) → ( $\beta \times \alpha$ )
end

```

β : fresh abstract var from generative application `G()`

α (for `t`) comes from recursive assumption

Step 3 — Subtyping: Reconciling Body with Signature

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec: \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow f$$

Find witnesses $\bar{\tau}$ such that body \mathcal{C}' matches external $\mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}]$, by looking up from \mathcal{C}' .

body result \mathcal{C}' (Step 2)

```

sig
  module H : ...
 $\bar{\alpha}, \bar{\beta} \vdash$    type  $t = \beta \times \alpha$ 
                   type  $u = \text{int} \times (\mu\gamma.\text{int} \times \gamma)$ 
                   val  $f : (\text{int} \times (\mu\gamma.\text{int} \times \gamma)) \rightarrow (\beta \times \alpha)$ 
  end

```

Step 3 — Subtyping: Reconciling Body with Signature

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec: \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow f$$

Find witnesses $\bar{\tau}$ such that body \mathcal{C}' matches external $\mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}]$, by looking up from \mathcal{C}' .

body result \mathcal{C}' (Step 2)

```

sig
  module H : ...
 $\bar{\alpha}, \bar{\beta} \vdash$   type t =  $\beta \times \alpha$ 
                 type u = int × (μγ.int × γ)
                 val f : (int × (μγ.int × γ)) → ( $\beta \times \alpha$ )
end

```

external signature \mathcal{C} (Step 1)

```

sig
 $\prec:$   type t =  $\alpha$ 
       type u = (μγ.int × γ)
       val f : (μβ.int × β) → α
end

```

Step 3 — Subtyping: Reconciling Body with Signature

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec: \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow f$$

Find witnesses $\bar{\tau}$ such that body \mathcal{C}' matches external $\mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}]$, by looking up from \mathcal{C}' .

body result \mathcal{C}' (Step 2)

```

sig
  module H : ...
 $\bar{\alpha}, \bar{\beta} \vdash$    type t =  $\beta \times \alpha$ 
                  type u = int × (μγ.int × γ)
                  val f : (int × (μγ.int × γ)) → ( $\beta \times \alpha$ )
end

```

external signature \mathcal{C} (Step 1)

```

sig
 $\prec:$    type t =  $\alpha$ 
        type u = (μγ.int × γ)
        val f : (μβ.int × β) →  $\alpha$ 
end

```

For \mathbf{t} , subtyping holds after $\alpha \mapsto \beta \times \alpha$.

Step 3 — Subtyping: Reconciling Body with Signature

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec: \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow f$$

Find witnesses $\bar{\tau}$ such that body \mathcal{C}' matches external $\mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}]$, by looking up from \mathcal{C}' .

	body result \mathcal{C}' (Step 2)	external signature \mathcal{C} (Step 1)
	sig	sig
$\bar{\alpha}, \bar{\beta} \vdash$	module $H : \dots$	$\prec:$ type $t = \alpha$
	type $t = \beta \times \alpha$	type $u = (\mu\gamma.\text{int} \times \gamma)$
	type $u = \text{int} \times (\mu\gamma.\text{int} \times \gamma)$	val $f : (\mu\beta.\text{int} \times \beta) \rightarrow \alpha$
	val $f : (\text{int} \times (\mu\gamma.\text{int} \times \gamma)) \rightarrow (\beta \times \alpha)$	end
	end	

For t , subtyping holds after $\alpha \mapsto \beta \times \alpha$.

For u , types are equal by equi-recursive unfolding.

Type Checking Recursive Modules

Without double vision, a recursive module $\text{rec } X : S = M$ can be understood as the fixed point of a functor **functor** $(X : S) \rightarrow M$.

Opaque mode is straightforward:

E-Typ-Rec-Opaque

$$\frac{\Gamma \vdash \text{rec } (X : S) : \lambda \bar{\alpha}. \mathcal{C} \quad \Gamma, \bar{\alpha}, X : \mathcal{C} \vdash M : \exists \nabla \bar{\beta}. \mathcal{C}' \rightsquigarrow e \quad \Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \leq: \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow f}{\Gamma \vdash \text{rec } X : S = M : \exists \nabla \bar{\alpha}. \mathcal{C} \rightsquigarrow \text{fix } (\dots)}$$

Step 1 — Signature

Elaborate $\text{rec } (X : S)$ to get shape \mathcal{C} with abstract vars $\bar{\alpha}$.

Step 2 — Body

Check M with $X : \mathcal{C}$ in scope (the recursive assumption). Body may introduce fresh vars $\bar{\beta}$.

Step 3 — Subtyping

Find witnesses $\bar{\tau}$ by *lookup* on type fields. Coercion f checks $\mathcal{C}' \leq \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}]$.

Elaborated Term: in the Generative Mode

Given

$$\Gamma \vdash S : \lambda \bar{\alpha}. \mathcal{C}$$

$$\Gamma, \bar{\alpha}, X : \mathcal{C} \vdash M : \exists^{\nabla} \bar{\beta}. \mathcal{C}' \rightsquigarrow e$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec : \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow f$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \bar{\tau} : wf$$

$$\text{rec } X : S = M \rightsquigarrow$$

$$\text{fix } \lambda(x : \exists^{\nabla} \alpha. \mathcal{C}).$$

$$\text{unpack}^{\nabla} \langle \bar{\alpha}, X \rangle = x \text{ in}$$

$$\text{unpack}^{\nabla} \langle \bar{\beta}, y \rangle = e \text{ in}$$

$$\text{pack } \langle f y, \bar{\tau} \rangle \text{ as } \exists^{\nabla} \alpha. \mathcal{C}$$

Elaborated Term: in the Generative Mode

Given

$$\Gamma \vdash S : \lambda \bar{\alpha}. \mathcal{C}$$

$$\Gamma, \bar{\alpha}, X : \mathcal{C} \vdash M : \exists^{\nabla} \bar{\beta}. \mathcal{C}' \rightsquigarrow e$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec : \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow f$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \bar{\tau} : wf$$

$$\text{rec } X : S = M \rightsquigarrow$$

$$\text{fix } \lambda(x : \exists^{\nabla} \alpha. \mathcal{C}).$$

$$\text{unpack}^{\nabla} \langle \bar{\alpha}, X \rangle = x \text{ in}$$

$$\text{unpack}^{\nabla} \langle \bar{\beta}, y \rangle = e \text{ in}$$

$$\text{pack } \langle f y, \bar{\tau} \rangle \text{ as } \exists^{\nabla} \alpha. \mathcal{C}$$

- The elaborated term is well typed at $\exists^{\nabla} \bar{\alpha}. \mathcal{C}$:

Elaborated Term: in the Generative Mode

Given

$$\Gamma \vdash S : \lambda \bar{\alpha}. \mathcal{C}$$

$$\Gamma, \bar{\alpha}, X : \mathcal{C} \vdash M : \exists^{\nabla} \bar{\beta}. \mathcal{C}' \rightsquigarrow e$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec : \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow f$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \bar{\tau} : wf$$

$$\text{rec } X : S = M \rightsquigarrow$$

$$\text{fix } \lambda(x : \exists^{\nabla} \alpha. \mathcal{C}).$$

$$\text{unpack}^{\nabla} \langle \bar{\alpha}, X \rangle = x \text{ in}$$

$$\text{unpack}^{\nabla} \langle \bar{\beta}, y \rangle = e \text{ in}$$

$$\text{pack } \langle f y, \bar{\tau} \rangle \text{ as } \exists^{\nabla} \alpha. \mathcal{C}$$

- The elaborated term is well typed at $\exists^{\nabla} \bar{\alpha}. \mathcal{C}$:
 - fixpoint over the abstract signature $\exists^{\nabla} \bar{\alpha}. \mathcal{C}$.

Elaborated Term: in the Generative Mode

Given

$$\Gamma \vdash S : \lambda \bar{\alpha}. \mathcal{C}$$

$$\Gamma, \bar{\alpha}, X : \mathcal{C} \vdash M : \exists^{\nabla} \bar{\beta}. \mathcal{C}' \rightsquigarrow e$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec : \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow f$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \bar{\tau} : wf$$

$$\text{rec } X : S = M \rightsquigarrow$$

$$\text{fix } \lambda(x : \exists^{\nabla} \alpha. \mathcal{C}).$$

$$\text{unpack}^{\nabla} \langle \bar{\alpha}, X \rangle = x \text{ in}$$

$$\text{unpack}^{\nabla} \langle \bar{\beta}, y \rangle = e \text{ in}$$

$$\text{pack } \langle f y, \bar{\tau} \rangle \text{ as } \exists^{\nabla} \alpha. \mathcal{C}$$

- The elaborated term is well typed at $\exists^{\nabla} \bar{\alpha}. \mathcal{C}$:
 - fixpoint over the abstract signature $\exists^{\nabla} \bar{\alpha}. \mathcal{C}$.
 - Unpacking $\bar{\alpha}$: $\alpha, (X : \mathcal{C}) \vdash \dots$

Elaborated Term: in the Generative Mode

Given

$$\Gamma \vdash S : \lambda \bar{\alpha}. \mathcal{C}$$

$$\Gamma, \bar{\alpha}, X : \mathcal{C} \vdash M : \exists^{\nabla} \bar{\beta}. \mathcal{C}' \rightsquigarrow e$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec : \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow f$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \bar{\tau} : wf$$

$$\text{rec } X : S = M \rightsquigarrow$$

$$\text{fix } \lambda(x : \exists^{\nabla} \alpha. \mathcal{C}).$$

$$\text{unpack}^{\nabla} \langle \bar{\alpha}, X \rangle = x \text{ in}$$

$$\text{unpack}^{\nabla} \langle \bar{\beta}, y \rangle = e \text{ in}$$

$$\text{pack } \langle f y, \bar{\tau} \rangle \text{ as } \exists^{\nabla} \alpha. \mathcal{C}$$

- The elaborated term is well typed at $\exists^{\nabla} \bar{\alpha}. \mathcal{C}$:
 - fixpoint over the abstract signature $\exists^{\nabla} \bar{\alpha}. \mathcal{C}$.
 - Unpacking $\bar{\alpha}$: $\alpha, (X : \mathcal{C}) \vdash \dots$
 - Unpacking $\bar{\beta}$: $\alpha, (X : \mathcal{C}), \bar{\beta}, (y : \mathcal{C}') \vdash \dots$

Elaborated Term: in the Generative Mode

Given

$$\Gamma \vdash S : \lambda \bar{\alpha}. \mathcal{C}$$

$$\Gamma, \bar{\alpha}, X : \mathcal{C} \vdash M : \exists^{\nabla} \bar{\beta}. \mathcal{C}' \rightsquigarrow e$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec : \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow f$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \bar{\tau} : wf$$

$$\text{rec } X : S = M \rightsquigarrow$$

$$\text{fix } \lambda(x : \exists^{\nabla} \alpha. \mathcal{C}).$$

$$\text{unpack}^{\nabla} \langle \bar{\alpha}, X \rangle = x \text{ in}$$

$$\text{unpack}^{\nabla} \langle \bar{\beta}, y \rangle = e \text{ in}$$

$$\text{pack} \langle f y, \bar{\tau} \rangle \text{ as } \exists^{\nabla} \alpha. \mathcal{C}$$

- The elaborated term is well typed at $\exists^{\nabla} \bar{\alpha}. \mathcal{C}$:
 - fixpoint over the abstract signature $\exists^{\nabla} \bar{\alpha}. \mathcal{C}$.
 - Unpacking $\bar{\alpha}$: $\alpha, (X : \mathcal{C}) \vdash \dots$
 - Unpacking $\bar{\beta}$: $\alpha, (X : \mathcal{C}), \bar{\beta}, (y : \mathcal{C}') \vdash \dots$
 - $f y : \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}]$, which packs to $\exists^{\nabla} \bar{\alpha}. \mathcal{C}$.

Elaborated Term: in the Generative Mode

Given

$$\Gamma \vdash S : \lambda \bar{\alpha}. \mathcal{C}$$

$$\Gamma, \bar{\alpha}, X : \mathcal{C} \vdash M : \exists^{\nabla} \bar{\beta}. \mathcal{C}' \rightsquigarrow e$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec: \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow f$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \bar{\tau} : wf$$

$$\text{rec } X : S = M \rightsquigarrow$$

$$\text{fix } \lambda(x : \exists^{\nabla} \alpha. \mathcal{C}).$$

$$\text{unpack}^{\nabla} \langle \bar{\alpha}, X \rangle = x \text{ in}$$

$$\text{unpack}^{\nabla} \langle \bar{\beta}, y \rangle = e \text{ in}$$

$$\text{pack } \langle f y, \bar{\tau} \rangle \text{ as } \exists^{\nabla} \alpha. \mathcal{C}$$

- The elaborated term is well typed at $\exists^{\nabla} \bar{\alpha}. \mathcal{C}$:
 - fixpoint over the abstract signature $\exists^{\nabla} \bar{\alpha}. \mathcal{C}$.
 - Unpacking $\bar{\alpha}$: $\alpha, (X : \mathcal{C}) \vdash \dots$
 - Unpacking $\bar{\beta}$: $\alpha, (X : \mathcal{C}), \bar{\beta}, (y : \mathcal{C}') \vdash \dots$
 - $f y : \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}]$, which packs to $\exists^{\nabla} \bar{\alpha}. \mathcal{C}$.
- **No new μ -type created**: Even though the witness can be cyclic (e.g. $\alpha \mapsto \beta \times \alpha$), it is hidden by the pack.

Challenges in the Applicative Mode

Given

$$\Gamma \vdash S : \lambda \bar{\alpha}. \mathcal{C}$$

$$\Gamma, \bar{\alpha}, X : \mathcal{C} \vdash M : \exists^{\nabla \tau_1}(\bar{\beta}). \mathcal{C}' \rightsquigarrow e$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec: \mathcal{C}[\bar{\alpha} \mapsto \tau_2] \rightsquigarrow f$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \tau_2 : wf$$

$$\text{rec } X : S = M \rightsquigarrow$$

$$\text{fix } \lambda(x : \exists^{\nabla \tau_2}(\alpha). \mathcal{C}).$$

$$\text{unpack}^{\nabla} \langle \bar{\alpha}, X \rangle = x \text{ in}$$

$$\text{unpack}^{\nabla} \langle \bar{\beta}, y \rangle = e \text{ in}$$

$$\text{pack } \langle f y, \tau_2 \rangle \text{ as } \exists^{\nabla \tau_2}(\alpha). \mathcal{C}$$

Opaque \exists^{∇} : packing hides the cyclic witness
Standard unpack+pack suffices here.

Challenges in the Applicative Mode

Given

$$\Gamma \vdash S : \lambda \bar{\alpha}. \mathcal{C}$$

$$\Gamma, \bar{\alpha}, X : \mathcal{C} \vdash M : \exists^{\nabla \tau_1}(\bar{\beta}). \mathcal{C}' \rightsquigarrow e$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec: \mathcal{C}[\bar{\alpha} \mapsto \tau_2] \rightsquigarrow f$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \tau_2 : wf$$

$$\text{rec } X : S = M \rightsquigarrow$$

$$\text{fix } \lambda(x : \exists^{\nabla ?\tau_2}(\alpha). \mathcal{C}).$$

$$\text{unpack}^{\nabla} \langle \bar{\alpha}, X \rangle = x \text{ in}$$

$$\text{unpack}^{\nabla} \langle \bar{\beta}, y \rangle = e \text{ in}$$

$$\text{pack } \langle f y, \tau_2 \rangle \text{ as } \exists^{\nabla ?\tau_2}(\alpha). \mathcal{C}$$

Opaque \exists^{∇} : packing hides the cyclic witness
Standard unpack+pack suffices here.

Transparent \exists^{∇} : The witness type $? \tau_2$ is *part of the type* — it must be a well-defined μ -type from the start. We must build it explicitly.

New Primitive: fixre^∇ — Fixpoint over Existential Packages

Goal: build the fixpoint on an existential whose witness type is recursive $\exists^\nabla \mu \bar{\alpha} . \bar{\tau}(\bar{\alpha}) . \mathcal{C}$.

F-Fixre

$$\frac{\Gamma, \beta \vdash \sigma : \text{wf} \quad \Gamma, \alpha \vdash \tau : \text{wf} \quad \Gamma, \alpha, (x : \sigma[\beta \mapsto \alpha]) \vdash e : \exists^\nabla \tau(\beta) . \sigma}{\Gamma \vdash \text{fixre}^\nabla \langle \alpha, x : \exists^\nabla \mu \alpha . \tau(\beta) . \sigma \rangle . e : \exists^\nabla \mu \alpha . \tau(\beta) . \sigma}$$

- Can be implemented in F_μ^ω type checker, sound by construction.

New Primitive: fixre^∇ — Fixpoint over Existential Packages

Goal: build the fixpoint on an existential whose witness type is recursive $\exists^\nabla \mu \bar{\alpha}. \bar{\tau}(\bar{\alpha}). \mathcal{C}$.

F-Fixre

$$\frac{\Gamma, \beta \vdash \sigma : \text{wf} \quad \Gamma, \alpha \vdash \tau : \text{wf} \quad \Gamma, \alpha, (x : \sigma[\beta \mapsto \alpha]) \vdash e : \exists^\nabla \tau(\beta). \sigma}{\Gamma \vdash \text{fixre}^\nabla \langle \alpha, x : \exists^\nabla \mu \alpha. \tau(\beta). \sigma \rangle . e : \exists^\nabla \mu \alpha. \tau(\beta). \sigma}$$

- Can be implemented in F_μ^ω type checker, sound by construction.
- Soundness: instantiate the premise with $[\alpha \mapsto \mu \alpha. \tau]$

$$\Gamma, x : \sigma[\beta \mapsto \mu \alpha. \tau] \vdash e : \exists^\nabla \tau[\alpha \mapsto \mu \alpha. \tau](\beta). \sigma = \exists^\nabla \mu \alpha. \tau(\beta). \sigma$$

New Primitive: fixre^∇ — Fixpoint over Existential Packages

Goal: build the fixpoint on an existential whose witness type is recursive $\exists^\nabla \mu \bar{\alpha}. \bar{\tau}(\bar{\alpha}). \mathcal{C}$.

F-Fixre

$$\frac{\Gamma, \beta \vdash \sigma : \text{wf} \quad \Gamma, \alpha \vdash \tau : \text{wf} \quad \Gamma, \alpha, (x : \sigma[\beta \mapsto \alpha]) \vdash e : \exists^\nabla \tau(\beta). \sigma}{\Gamma \vdash \text{fixre}^\nabla \langle \alpha, x : \exists^\nabla \mu \alpha. \tau(\beta). \sigma \rangle . e : \exists^\nabla \mu \alpha. \tau(\beta). \sigma}$$

- Can be implemented in F_μ^ω type checker, sound by construction.
- Soundness: instantiate the premise with $[\alpha \mapsto \mu \alpha. \tau]$

$$\Gamma, x : \sigma[\beta \mapsto \mu \alpha. \tau] \vdash e : \exists^\nabla \tau[\alpha \mapsto \mu \alpha. \tau](\beta). \sigma = \exists^\nabla \mu \alpha. \tau(\beta). \sigma$$

- For \exists^∇ : a new primitive in transparent existential library

New Primitive: fixre^∇ — Fixpoint over Existential Packages

Goal: build the fixpoint on an existential whose witness type is recursive $\exists^\nabla \mu \bar{\alpha}. \bar{\tau}(\bar{\alpha}). \mathcal{C}$.

F-Fixre

$$\frac{\Gamma, \beta \vdash \sigma : \text{wf} \quad \Gamma, \alpha \vdash \tau : \text{wf} \quad \Gamma, \alpha, (x : \sigma[\beta \mapsto \alpha]) \vdash e : \exists^\nabla \tau(\beta). \sigma}{\Gamma \vdash \text{fixre}^\nabla \langle \alpha, x : \exists^\nabla \mu \alpha. \tau(\beta). \sigma \rangle . e : \exists^\nabla \mu \alpha. \tau(\beta). \sigma}$$

- Can be implemented in F_μ^ω type checker, sound by construction.
- Soundness: instantiate the premise with $[\alpha \mapsto \mu \alpha. \tau]$

$$\Gamma, x : \sigma[\beta \mapsto \mu \alpha. \tau] \vdash e : \exists^\nabla \tau[\alpha \mapsto \mu \alpha. \tau](\beta). \sigma = \exists^\nabla \mu \alpha. \tau(\beta). \sigma$$

- For \exists^∇ : a new primitive in transparent existential library
- For \exists^∇ : defined as a syntactic sugar: $\text{fix}(\lambda(x : \exists^\nabla \beta. \sigma). \text{unpack}^\nabla \langle \beta, x \rangle = x \text{ in } \dots)$.

New Primitive: compact^∇ — Propagating Witnesses

Given

$$\Gamma \vdash S : \lambda \bar{\alpha}. \mathcal{C}$$

$$\Gamma, \bar{\alpha}, \bar{X} : \mathcal{C} \vdash M : \exists^{\nabla \bar{\tau}_1} (\bar{\beta}). \mathcal{C}' \rightsquigarrow e$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec : \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}_2] \rightsquigarrow f$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \bar{\tau}_2 : wf$$

$$\text{rec } X : S = M \rightsquigarrow \text{fixre}^\nabla \langle \bar{\alpha}, x : \exists^{\nabla \mu \bar{\alpha}.?} (\bar{\alpha}). \mathcal{C} \rangle .?$$

New Primitive: compact^∇ — Propagating Witnesses

Given

$$\Gamma \vdash S : \lambda \bar{\alpha}. \mathcal{C}$$

$$\Gamma, \bar{\alpha}, \bar{X} : \mathcal{C} \vdash M : \exists^{\nabla \bar{\tau}_1}(\bar{\beta}). \mathcal{C}' \rightsquigarrow e$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec: \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}_2] \rightsquigarrow f$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \bar{\tau}_2 : \text{wf}$$

$$\text{rec } X : S = M \rightsquigarrow \text{fixre}^\nabla \langle \bar{\alpha}, x : \exists^{\nabla \mu \bar{\alpha}.?}(\bar{\alpha}). \mathcal{C} \rangle .?$$

$$\text{repack} \langle \bar{\beta}, y \rangle = e \text{ in } f y : \exists^{\nabla \bar{\tau}_1}(\bar{\beta}). \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}_2]$$

New Primitive: compact^∇ — Propagating Witnesses

Given

$$\Gamma \vdash S : \lambda \bar{\alpha}. \mathcal{C}$$

$$\Gamma, \bar{\alpha}, \bar{X} : \mathcal{C} \vdash M : \exists^{\nabla \bar{\tau}_1}(\bar{\beta}). \mathcal{C}' \rightsquigarrow e$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec: \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}_2] \rightsquigarrow f$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \bar{\tau}_2 : \text{wf}$$

$$\text{rec } X : S = M \rightsquigarrow \text{fixre}^\nabla \langle \bar{\alpha}, x : \exists^{\nabla \mu \bar{\alpha}.?}(\bar{\alpha}). \mathcal{C} \rangle . ?$$

$$\text{repack} \langle \bar{\beta}, y \rangle = e \text{ in } f y : \exists^{\nabla \bar{\tau}_1}(\bar{\beta}). \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}_2]$$

The external signature expects $\exists^{\nabla?}(\bar{\alpha}). \mathcal{C}$. We achieve it with a new primitive compact^∇ .

F-Compact

$$\frac{\Gamma \vdash e : \exists^{\nabla \bar{\tau}_1}(\bar{\beta}). \sigma[\bar{\alpha} \mapsto \bar{\tau}_2] \quad \Gamma, \bar{\alpha} \vdash \sigma : \text{wf} \quad \Gamma, \bar{\beta} \vdash \bar{\tau}_2 : \text{wf}}{\text{compact}^\nabla \langle \bar{\beta}, \bar{\tau}_2, e \rangle \text{ as } \exists^{\nabla (\bar{\tau}_2[\bar{\beta} \mapsto \bar{\tau}_1])}(\bar{\alpha}). \sigma : \exists^{\nabla (\bar{\tau}_2[\bar{\beta} \mapsto \bar{\tau}_1])}(\bar{\alpha}). \sigma}$$

New Primitive: compact^∇ — Propagating Witnesses

Given

$$\Gamma \vdash S : \lambda \bar{\alpha}. \mathcal{C}$$

$$\Gamma, \bar{\alpha}, \bar{X} : \mathcal{C} \vdash M : \exists^{\nabla \bar{\tau}_1}(\bar{\beta}). \mathcal{C}' \rightsquigarrow e$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec: \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}_2] \rightsquigarrow f$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \bar{\tau}_2 : \text{wf}$$

$$\text{rec } X : S = M \rightsquigarrow \text{fixre}^\nabla \langle \bar{\alpha}, x : \exists^{\nabla \mu \bar{\alpha}.?}(\bar{\alpha}). \mathcal{C} \rangle . ?$$

$$\text{repack} \langle \bar{\beta}, y \rangle = e \text{ in } f y : \exists^{\nabla \bar{\tau}_1}(\bar{\beta}). \mathcal{C}[\bar{\alpha} \mapsto \bar{\tau}_2]$$

The external signature expects $\exists^{\nabla?}(\bar{\alpha}). \mathcal{C}$. We achieve it with a new primitive compact^∇ .

F-Compact

$$\frac{\Gamma \vdash e : \exists^{\nabla \bar{\tau}_1}(\bar{\beta}). \sigma[\bar{\alpha} \mapsto \bar{\tau}_2] \quad \Gamma, \bar{\alpha} \vdash \sigma : \text{wf} \quad \Gamma, \bar{\beta} \vdash \bar{\tau}_2 : \text{wf}}{\text{compact}^\nabla \langle \bar{\beta}, \bar{\tau}_2, e \rangle \text{ as } \exists^{\nabla (\bar{\tau}_2[\bar{\beta} \mapsto \bar{\tau}_1])}(\bar{\alpha}). \sigma : \exists^{\nabla (\bar{\tau}_2[\bar{\beta} \mapsto \bar{\tau}_1])}(\bar{\alpha}). \sigma}$$

- For \exists^∇ : a primitive that propagates transparency.

New Primitive: compact^∇ — Propagating Witnesses

Given

$$\Gamma \vdash S : \lambda \bar{\alpha}. \mathcal{C}$$

$$\Gamma, \bar{\alpha}, X : \mathcal{C} \vdash M : \exists^{\nabla \tau_1}(\bar{\beta}). \mathcal{C}' \rightsquigarrow e$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \mathcal{C}' \prec: \mathcal{C}[\bar{\alpha} \mapsto \tau_2] \rightsquigarrow f$$

$$\Gamma, \bar{\alpha}, \bar{\beta} \vdash \tau_2 : \text{wf}$$

$$\text{rec } X : S = M \rightsquigarrow \text{fixre}^\nabla \langle \bar{\alpha}, x : \exists^{\nabla \mu \bar{\alpha}.?}(\bar{\alpha}). \mathcal{C} \rangle . ?$$

$$\text{repack} \langle \bar{\beta}, y \rangle = e \text{ in } f y : \exists^{\nabla \tau_1}(\bar{\beta}). \mathcal{C}[\bar{\alpha} \mapsto \tau_2]$$

The external signature expects $\exists^{\nabla?}(\bar{\alpha}). \mathcal{C}$. We achieve it with a new primitive compact^∇ .

F-Compact

$$\frac{\Gamma \vdash e : \exists^{\nabla \tau_1}(\bar{\beta}). \sigma[\bar{\alpha} \mapsto \tau_2] \quad \Gamma, \bar{\alpha} \vdash \sigma : \text{wf} \quad \Gamma, \bar{\beta} \vdash \tau_2 : \text{wf}}{\text{compact}^\nabla \langle \bar{\beta}, \tau_2, e \rangle \text{ as } \exists^{\nabla(\tau_2[\bar{\beta} \mapsto \tau_1])}(\bar{\alpha}). \sigma : \exists^{\nabla(\tau_2[\bar{\beta} \mapsto \tau_1])}(\bar{\alpha}). \sigma}$$

- For \exists^∇ : a primitive that propagates transparency.
- For \exists^∇ : defined as syntactic sugar: $\text{unpack}^\nabla \langle \bar{\beta}, x \rangle = e \text{ in } \text{pack}^\nabla \langle \tau_2, x \rangle \text{ as } \exists^\nabla \bar{\alpha}. \sigma$

The Full Elaboration Rule, applicative mode

E-Typ-Rec

$$\frac{\underbrace{\Gamma \vdash S : \lambda \bar{\alpha}. C}_{(1) \text{ signature}} \quad \underbrace{\Gamma, \bar{\alpha}, X : C \vdash M : \exists^{\nabla \bar{\sigma}}(\bar{\beta}). C' \rightsquigarrow e}_{(2) \text{ body}} \quad \underbrace{\Gamma, \bar{\alpha}, \bar{\beta} \vdash C' \prec : C[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow f}_{(3) \text{ subtyping}}}{\Gamma \vdash \text{rec } X : S = M : \exists^{\nabla \mu \bar{\alpha}. \bar{\tau}[\bar{\beta} \mapsto \bar{\sigma}]}(\bar{\alpha}). C \rightsquigarrow \text{fixre}^{\nabla} \langle \bar{\alpha}, X \rangle . \text{compact}^{\nabla} \langle \bar{\beta}, \bar{\tau}, \text{repack} \langle \bar{\beta}, y \rangle = e \text{ in } f y \rangle}$$

$\text{repack} \langle \bar{\beta}, y \rangle = e \text{ in } f y$ Applies the coercion f inside the body's package.

$\text{compact}^{\nabla} \langle \bar{\beta}, \bar{\tau}, \dots \rangle$ Re-packs the body's $\exists \bar{\beta}$ into the external $\exists \bar{\alpha}$.

$\text{fixre}^{\nabla} \langle \bar{\alpha}, X \rangle . \dots$ Ties the knot: witness type is $\mu \bar{\alpha}. \bar{\tau}[\bar{\beta} \mapsto \bar{\sigma}]$

A unified treatment: The same rule after changing $\nabla \tau$ to \blacktriangledown also works for the generative mode.

Double Vision: a two-pass elaboration strategy

“Single Vision” is not enough

✓ Bootstrapped Heaps × Tree/Forest

Recursive knot is tied after sealing, no way to realize `Tree.t = t`

Double Vision: a two-pass elaboration strategy

“Single Vision” is not enough

✓ Bootstrapped Heaps × Tree/Forest

Recursive knot is tied after sealing, no way to realize `Tree.t = t`

Inspired by Dreyer (2007), we introduce a **Static Pass** to M_{μ}^{ω} elaboration:

Double Vision: a two-pass elaboration strategy

“Single Vision” is not enough

✓ Bootstrapped Heaps × Tree/Forest

Recursive knot is tied after sealing, no way to realize `Tree.t = t`

Inspired by Dreyer (2007), we introduce a **Static Pass** to M_{μ}^{ω} elaboration:

1. **Static Checking:** Check the module body but **ignore terms**.
 - `let x = e ~> val x : ⊥`
 - Only type definitions are kept.

Double Vision: a two-pass elaboration strategy

“Single Vision” is not enough

✓ Bootstrapped Heaps × Tree/Forest

Recursive knot is tied after sealing, no way to realize `Tree.t = t`

Inspired by Dreyer (2007), we introduce a **Static Pass** to M_{μ}^{ω} elaboration:

1. **Static Checking:** Check the module body but **ignore terms**.
 - `let x = e ~> val x : ⊥`
 - Only type definitions are kept.
2. **Equation Discovery:** Collect internal type definitions from the static result.

Double Vision: Step 1 & 2 — Static Pass

Goal: discover internal type definitions *without* checking terms.

Source code (from introduction)

```
1 module rec Tree : sig
2   type t (* ~>  $\alpha$  *)
3   val head : t → t option
4 end = struct
5   type t = Leaf of int
6         | Node of Forest.t
7   let rec head = ... (* ignored *)
8 end
9 and Forest : sig
10  type t (* ~>  $\beta$  *)
11  val head : t → Tree.t option
12 end = struct
13  type t = Tree.t list
14  let head = ... (* ignored *)
15 end
16
```

Static type checking

(External Signature)

$$\left\{ \begin{array}{l} \text{Tree.t} = \alpha \\ \text{Forest.t} = \beta \end{array} \right\}$$

Double Vision: Step 1 & 2 — Static Pass

Goal: discover internal type definitions *without* checking terms.

Source code (from introduction)

```

1  module rec Tree : sig
2    type t                (* ~→ α *)
3    val head : t → t option
4  end = struct
5    type t = Leaf of int
6           | Node of Forest.t
7    let rec head = ...    (* ignored *)
8  end
9  and Forest : sig
10   type t                (* ~→ β *)
11   val head : t → Tree.t option
12 end = struct
13   type t = Tree.t list
14   let head = ...        (* ignored *)
15 end
16
```

Static type checking

(External Signature)

$$\left\{ \begin{array}{l} \text{Tree.t} = \alpha \\ \text{Forest.t} = \beta \end{array} \right\} \vdash_{\text{static}}$$

Tree : sig type $t = \text{int} + \beta$ end
 Forest : sig type $t = \text{list}(\alpha)$ end

Double Vision: Step 1 & 2 — Static Pass

Goal: discover internal type definitions *without* checking terms.

Source code (from introduction)

```

1  module rec Tree : sig
2    type t                (* ~→ α *)
3    val head : t → t option
4  end = struct
5    type t = Leaf of int
6           | Node of Forest.t
7    let rec head = ...    (* ignored *)
8  end
9  and Forest : sig
10   type t                (* ~→ β *)
11   val head : t → Tree.t option
12 end = struct
13   type t = Tree.t list
14   let head = ...        (* ignored *)
15 end
16
```

Static type checking

(External Signature)

$$\left\{ \begin{array}{l} \text{Tree.t} = \alpha \\ \text{Forest.t} = \beta \end{array} \right\} \vdash_{\text{static}}$$

Tree : sig type $t = \text{int} + \beta$ end
 Forest : sig type $t = \text{list}(\alpha)$ end

Equation discovery:

- For Tree: $\alpha \mapsto \text{int} + \beta$
- For Forest: $\beta \mapsto \text{list}(\alpha)$

Double Vision: a two-pass elaboration strategy

Inspired by Dreyer (2007), we introduce a **Static Pass** to M_μ^ω elaboration:

1. **Static Checking:** Check the module body but **ignore terms**.
 - `let x = e ~> val x : ⊥`
 - Only type definitions are kept.
2. **Equation Discovery:** Collect internal type definitions from the static result.
3. **Context Strengthening:** Refine the recursive assumptions with these discovered identities.

Double Vision: Step 3 — Context Strengthening

Each submodule is re-checked under a **partially instantiated** environment.

Internal view for Tree

(Tree.t(α) concrete, Forest.t(β) abstract)

$$\text{Tree} : \left\{ \begin{array}{l} \text{type } t = \text{int} + \beta; \\ \text{val } \textit{head} : (\text{int} + \beta) \rightarrow (\text{int} + \beta) \text{ option} \end{array} \right\}$$
$$\text{Forest} : \left\{ \begin{array}{l} \text{type } t = \beta; \\ \text{val } \textit{head} : \beta \rightarrow (\text{int} + \beta) \text{ option} \end{array} \right\}$$

Double Vision: Step 3 — Context Strengthening

Each submodule is re-checked under a **partially instantiated** environment.

Internal view for `Tree`

(`Tree.t` (α) concrete, `Forest.t` (β) abstract)

$$\text{Tree} : \left\{ \begin{array}{l} \text{type } t = \text{int} + \beta; \\ \text{val } \text{head} : (\text{int} + \beta) \rightarrow (\text{int} + \beta) \text{ option} \end{array} \right\}$$
$$\text{Forest} : \left\{ \begin{array}{l} \text{type } t = \beta; \\ \text{val } \text{head} : \beta \rightarrow (\text{int} + \beta) \text{ option} \end{array} \right\}$$

Example: This call now typechecks for `Tree`

```
1 | Node f → match Forest.head f with ...
2 | Some tr' → head tr'
3 (* Forest.head :  $\beta \rightarrow (\text{int} + \beta) \text{ option}$  *)
4 (* tr' :  $\text{int} + \beta$  *)
```

Double Vision: Step 3 — Context Strengthening

Each submodule is re-checked under a **partially instantiated** environment.

Internal view for **Tree**

(**Tree**. $t(\alpha)$ concrete, **Forest**. $t(\beta)$ abstract)

$$\text{Tree} : \left\{ \begin{array}{l} \text{type } t = \text{int} + \beta; \\ \text{val } \text{head} : (\text{int} + \beta) \rightarrow (\text{int} + \beta) \text{ option} \end{array} \right\}$$

$$\text{Forest} : \left\{ \begin{array}{l} \text{type } t = \beta; \\ \text{val } \text{head} : \beta \rightarrow (\text{int} + \beta) \text{ option} \end{array} \right\}$$

Internal view for **Forest**

(**Tree**. $t(\alpha)$ abstract, **Forest**. $t(\beta)$ concrete)

$$\text{Tree} : \left\{ \begin{array}{l} \text{type } t = \alpha; \\ \text{val } \text{head} : \alpha \rightarrow \alpha \text{ option} \end{array} \right\}$$

$$\text{Forest} : \left\{ \begin{array}{l} \text{type } t = \text{list}(\alpha); \\ \text{val } \text{head} : \text{list}(\alpha) \rightarrow \alpha \text{ option} \end{array} \right\}$$

Example: This call now typechecks for **Tree**

```

1 | Node f → match Forest.head f with ...
2 | Some tr' → head tr'
3 (* Forest.head : β → (int + β)option *)
4 (* tr' : int + β *)

```

Double Vision: Step 3 — Context Strengthening

Each submodule is re-checked under a **partially instantiated** environment.

Internal view for Tree

(Tree.t(α) concrete, Forest.t(β) abstract)

$$\text{Tree} : \left\{ \begin{array}{l} \text{type } t = \text{int} + \beta; \\ \text{val } \text{head} : (\text{int} + \beta) \rightarrow (\text{int} + \beta) \text{ option} \end{array} \right\}$$

$$\text{Forest} : \left\{ \begin{array}{l} \text{type } t = \beta; \\ \text{val } \text{head} : \beta \rightarrow (\text{int} + \beta) \text{ option} \end{array} \right\}$$

Example: This call now typechecks for Tree

```

1 | Node f → match Forest.head f with ...
2 | Some tr' → head tr'
3 (* Forest.head : β → (int + β)option *)
4 (* tr' : int + β *)

```

Internal view for Forest

(Tree.t(α) abstract, Forest.t(β) concrete)

$$\text{Tree} : \left\{ \begin{array}{l} \text{type } t = \alpha; \\ \text{val } \text{head} : \alpha \rightarrow \alpha \text{ option} \end{array} \right\}$$

$$\text{Forest} : \left\{ \begin{array}{l} \text{type } t = \text{list}(\alpha); \\ \text{val } \text{head} : \text{list}(\alpha) \rightarrow \alpha \text{ option} \end{array} \right\}$$

Example: Rejected in Forest

```

1 match tr with
2 | Leaf _ → ... (* Forest cannot see
3               Tree's internal representation:
4               Tree.t = Leaf .. | Node .. *)

```

Double Vision: Step 3 — Context Strengthening

Each submodule is re-checked under a **partially instantiated** environment.

Internal view for Tree

(Tree.t(α) concrete, Forest.t(β) abstract)

$$\text{Tree} : \left\{ \begin{array}{l} \text{type } t = \text{int} + \beta; \\ \text{val } \text{head} : (\text{int} + \beta) \rightarrow (\text{int} + \beta) \text{ option} \end{array} \right\}$$

$$\text{Forest} : \left\{ \begin{array}{l} \text{type } t = \beta; \\ \text{val } \text{head} : \beta \rightarrow (\text{int} + \beta) \text{ option} \end{array} \right\}$$

Example: This call now typechecks for Tree

```

1 | Node f → match Forest.head f with ...
2 | Some tr' → head tr'
3 (* Forest.head : β → (int + β)option *)
4 (* tr' : int + β *)

```

Internal view for Forest

(Tree.t(α) abstract, Forest.t(β) concrete)

$$\text{Tree} : \left\{ \begin{array}{l} \text{type } t = \alpha; \\ \text{val } \text{head} : \alpha \rightarrow \alpha \text{ option} \end{array} \right\}$$

$$\text{Forest} : \left\{ \begin{array}{l} \text{type } t = \text{list}(\alpha); \\ \text{val } \text{head} : \text{list}(\alpha) \rightarrow \alpha \text{ option} \end{array} \right\}$$

Example: Rejected in Forest

```

1 match tr with
2 | Leaf _ → ... (* Forest cannot see
3               Tree's internal representation:
4               Tree.t = Leaf .. | Node .. *)

```

Data abstraction across module boundaries is preserved.

Double Vision: Elaboration

$$\beta, \mathcal{C}_{\text{Tree}}^{\text{ext}}[\alpha \mapsto \text{int} + \beta] \vdash M_{\text{Tree}} \rightsquigarrow e_{\text{tree}} \quad \alpha, \mathcal{C}_{\text{Forest}}^{\text{ext}}[\beta \mapsto \text{list}(\alpha)] \vdash M_{\text{Forest}} \rightsquigarrow e_{\text{forest}}$$

Double Vision: Elaboration

$$\beta, \mathcal{C}_{\text{Tree}}^{\text{ext}}[\alpha \mapsto \text{int} + \beta] \vdash M_{\text{Tree}} \rightsquigarrow e_{\text{tree}} \quad \alpha, \mathcal{C}_{\text{Forest}}^{\text{ext}}[\beta \mapsto \text{list}(\alpha)] \vdash M_{\text{Forest}} \rightsquigarrow e_{\text{forest}}$$

With term elaborated under the strengthened contexts, we can only build the fixpoint on signature where both α and β are *concrete*.

$$\begin{cases} \alpha = \text{int} + \beta \\ \beta = \text{list}(\alpha) \end{cases} \Rightarrow \begin{cases} \alpha \mapsto \tau_1 \triangleq \mu\alpha.\text{int} + \text{list}(\alpha) \\ \beta \mapsto \tau_2 \triangleq \text{list}(\mu\alpha.\text{int} + \text{list}(\alpha)) \end{cases}$$

Double Vision: Elaboration

$$\beta, \mathcal{C}_{\text{Tree}}^{\text{ext}}[\alpha \mapsto \text{int} + \beta] \vdash M_{\text{Tree}} \rightsquigarrow e_{\text{tree}} \quad \alpha, \mathcal{C}_{\text{Forest}}^{\text{ext}}[\beta \mapsto \text{list}(\alpha)] \vdash M_{\text{Forest}} \rightsquigarrow e_{\text{forest}}$$

With term elaborated under the strengthened contexts, we can only build the fixpoint on signature where both α and β are *concrete*.

$$\begin{cases} \alpha = \text{int} + \beta \\ \beta = \text{list}(\alpha) \end{cases} \Rightarrow \begin{cases} \alpha \mapsto \tau_1 \triangleq \mu\alpha.\text{int} + \text{list}(\alpha) \\ \beta \mapsto \tau_2 \triangleq \text{list}(\mu\alpha.\text{int} + \text{list}(\alpha)) \end{cases}$$

Elaboration: specialize e_{tree} and e_{forest} with the concrete witnesses of the other module

$$\{\ell_{\text{tree}} = e_{\text{tree}}[\tau_2]\} @ \{\ell_{\text{forest}} = e_{\text{forest}}[\tau_1]\} \dots$$

Double Vision: Elaboration

$$\beta, \mathcal{C}_{\text{Tree}}^{\text{ext}}[\alpha \mapsto \text{int} + \beta] \vdash M_{\text{Tree}} \rightsquigarrow e_{\text{tree}} \quad \alpha, \mathcal{C}_{\text{Forest}}^{\text{ext}}[\beta \mapsto \text{list}(\alpha)] \vdash M_{\text{Forest}} \rightsquigarrow e_{\text{forest}}$$

With term elaborated under the strengthened contexts, we can only build the fixpoint on signature where both α and β are *concrete*.

$$\begin{cases} \alpha = \text{int} + \beta \\ \beta = \text{list}(\alpha) \end{cases} \Rightarrow \begin{cases} \alpha \mapsto \tau_1 \triangleq \mu\alpha.\text{int} + \text{list}(\alpha) \\ \beta \mapsto \tau_2 \triangleq \text{list}(\mu\alpha.\text{int} + \text{list}(\alpha)) \end{cases}$$

Elaboration: specialize e_{tree} and e_{forest} with the concrete witnesses of the other module

$$\text{fix } \lambda(X : \mathcal{C}_{\text{Tree}}^{\text{ext}}[\alpha, \beta \mapsto \tau_1, \tau_2]). \\ \dots \{l_{\text{tree}} = e_{\text{tree}}[\tau_2]\} @ \{l_{\text{forest}} = e_{\text{forest}}[\tau_1]\} \dots$$

Double Vision: Elaboration

$$\beta, \mathcal{C}_{\text{Tree}}^{\text{ext}}[\alpha \mapsto \text{int} + \beta] \vdash M_{\text{Tree}} \rightsquigarrow e_{\text{tree}} \quad \alpha, \mathcal{C}_{\text{Forest}}^{\text{ext}}[\beta \mapsto \text{list}(\alpha)] \vdash M_{\text{Forest}} \rightsquigarrow e_{\text{forest}}$$

With term elaborated under the strengthened contexts, we can only build the fixpoint on signature where both α and β are *concrete*.

$$\begin{cases} \alpha = \text{int} + \beta \\ \beta = \text{list}(\alpha) \end{cases} \Rightarrow \begin{cases} \alpha \mapsto \tau_1 \triangleq \mu\alpha.\text{int} + \text{list}(\alpha) \\ \beta \mapsto \tau_2 \triangleq \text{list}(\mu\alpha.\text{int} + \text{list}(\alpha)) \end{cases}$$

Elaboration: specialize e_{tree} and e_{forest} with the concrete witnesses of the other module

$$\begin{aligned} & \text{pack} \langle \tau_1 \tau_2, \text{fix } \lambda(X : \mathcal{C}_{\text{Tree}}^{\text{ext}}[\alpha, \beta \mapsto \tau_1, \tau_2]). \\ & \quad \dots \{ \ell_{\text{tree}} = e_{\text{tree}}[\tau_2] \} @ \{ \ell_{\text{forest}} = e_{\text{forest}}[\tau_1] \} \dots \\ & \rangle \text{as } \exists^{\nabla} \tau_1, \tau_2 (\alpha, \beta). \mathcal{C}^{\text{ext}} \end{aligned}$$

Double Vision: Elaboration

$$\beta, \mathcal{C}_{\text{Tree}}^{\text{ext}}[\alpha \mapsto \text{int} + \beta] \vdash M_{\text{Tree}} \rightsquigarrow e_{\text{tree}} \quad \alpha, \mathcal{C}_{\text{Forest}}^{\text{ext}}[\beta \mapsto \text{list}(\alpha)] \vdash M_{\text{Forest}} \rightsquigarrow e_{\text{forest}}$$

With term elaborated under the strengthened contexts, we can only build the fixpoint on signature where both α and β are *concrete*.

$$\begin{cases} \alpha = \text{int} + \beta \\ \beta = \text{list}(\alpha) \end{cases} \Rightarrow \begin{cases} \alpha \mapsto \tau_1 \triangleq \mu\alpha.\text{int} + \text{list}(\alpha) \\ \beta \mapsto \tau_2 \triangleq \text{list}(\mu\alpha.\text{int} + \text{list}(\alpha)) \end{cases}$$

Elaboration: specialize e_{tree} and e_{forest} with the concrete witnesses of the other module

$$\begin{aligned} & \text{pack} \langle \tau_1 \tau_2, \text{fix } \lambda(X : \mathcal{C}_{\text{Tree}}^{\text{ext}}[\alpha, \beta \mapsto \tau_1, \tau_2]). \\ & \quad \dots \{ \ell_{\text{tree}} = e_{\text{tree}}[\tau_2] \} @ \{ \ell_{\text{forest}} = e_{\text{forest}}[\tau_1] \} \dots \\ & \rangle \text{as } \exists^{\nabla} \tau_1, \tau_2 (\alpha, \beta). \mathcal{C}^{\text{ext}} \end{aligned}$$

(Specialized μ -types are compatible: $(\text{int} + \beta)[\beta \mapsto \tau_2] = \text{int} + \text{list}(\mu\alpha.\text{int} + \text{list}(\alpha)) = \tau_1$)

Double Vision: a two-pass elaboration strategy

Inspired by Dreyer (2007), we introduce a **Static Pass** to M_μ^ω elaboration:

1. **Static Checking:** Check the module body but **ignore terms**.
 - `let x = e ~> val x : ⊥`
 - Only type definitions are kept.
2. **Equation Discovery:** Collect internal type definitions from the static result.
3. **Context Strengthening:** Refine the recursive assumptions with these discovered identities.
4. **Elaboration:** Re-check the body with terms, equipped with the concrete views.

Double Vision: a two-pass elaboration strategy

Inspired by Dreyer (2007), we introduce a **Static Pass** to M_μ^ω elaboration:

1. **Static Checking:** Check the module body but **ignore terms**.
 - `let x = e ~> val x : ⊥`
 - Only type definitions are kept.
2. **Equation Discovery:** Collect internal type definitions from the static result.
3. **Context Strengthening:** Refine the recursive assumptions with these discovered identities.
4. **Elaboration:** Re-check the body with terms, equipped with the concrete views.

Formal type checking and elaboration rules are defined in the paper.

Double Vision as Source-level Encoding

```
1 let F_Tree (Tree : S_Tree [ Tree.t  $\mapsto$  Leaf .. | Node .. ] )
2     (Forest : S_Forest [ Tree.t  $\mapsto$  Leaf .. | Node .. ]) = M_Tree
3
4 let F_Forest (Tree : S_Tree [ Forest.t  $\mapsto$  Tree.t list ])
5     (Forest : S_Forest [ Forest.t  $\mapsto$  Tree.t list ]) = M_Forest
6
7 module rec P : sig
8   module Tree : S_Tree [ Tree.t  $\mapsto$  .. , Forest.t  $\mapsto$  .. ]
9   module Forest : S_Forest [ Tree.t  $\mapsto$  .. , Forest.t  $\mapsto$  .. ]
10 end = struct
11   module Tree = F_Tree (P.Tree) (P.Forest)
12   module Forest = F_Forest (P.Tree) (P.Forest)
13 end
14
15 let P = (P : sig module Tree : S_Tree module Forest : S_Forest end)
```

Double vision can be understood as a source-level transformation. M_μ^ω 's elaboration suggests this encoding is fundamental.

A Limitation: No Deep Double Vision

We only do strengthening on definitions sealed at the top-level (external signature).

```
1 module type TREE = sig
2   type t
3   val head : t → t option end
4 module type FOREST = sig
5   type t
6   val head : t → Tree.t option end
7 module type TF = sig
8   module Tree : TREE
9   module Forest : FOREST
10 end
```

```
1 module rec TreeForest : TF = struct
2   module Tree : TREE = struct
3     type t = Leaf of int | Node of Forest.t
4     let head = ... (* TreeForest.Tree.t != t *)
5   end
6   module Forest : FOREST = struct
7     type t = Tree.t list
8     let head = ...
9   end
10 end
```

A Limitation: No Deep Double Vision

We only do strengthening on definitions sealed at the top-level (external signature).

```
1 module type TREE = sig
2   type t
3   val head : t → t option end
4 module type FOREST = sig
5   type t
6   val head : t → Tree.t option end
7 module type TF = sig
8   module Tree : TREE
9   module Forest : FOREST
10  end
```

```
1 module rec TreeForest : TF = struct
2   module Tree : TREE = struct
3     type t = Leaf of int | Node of Forest.t
4     let head = ... (* TreeForest.Tree.t != t *)
5   end
6   module Forest : FOREST = struct
7     type t = Tree.t list
8     let head = ...
9   end
10  end
```

`Tree` and `Forest` are sealed inside `TreeForest`, so static pass cannot discover their internal definitions. The above cannot be solved by M_{μ}^{ω} .

A Limitation: No Deep Double Vision

We only do strengthening on definitions sealed at the top-level (external signature).

```

1  module type TREE = sig
2    type t
3    val head : t → t option end
4  module type FOREST = sig
5    type t
6    val head : t → Tree.t option end
7  module type TF = sig
8    module Tree : TREE
9    module Forest : FOREST
10 end

```

```

1  module rec TreeForest : TF = struct
2    module Tree : TREE = struct
3      type t = Leaf of int | Node of Forest.t
4      let head = ... (* TreeForest.Tree.t != t *)
5    end
6    module Forest : FOREST = struct
7      type t = Tree.t list
8      let head = ...
9    end
10 end

```

`Tree` and `Forest` are sealed inside `TreeForest`, so static pass cannot discover their internal definitions. The above cannot be solved by M_{μ}^{ω} .

Workaround : remove the inner sealing `Tree : TREE` and `Forest : FOREST`.

Comparison with Related Work

Feature	Moscow ML Russo (2001)	Traviata Nakata et al. (2006)	RMC Dreyer (2007)	$\lambda_{\text{abs}}^{\text{rec}}$ Im et al. (2011)	M_{μ}^{ω} Ours	OCaml
Generative Functors	✓		✓	✓	✓	✓
Applicative Functors		✓			✓	✓
Higher-Order Functor	✓		✓	✓	✓	✓
External Signature	Required	Inferred	Required	Required	Required	Required
Double Vision	Avoided	Manual	Full	Full	Top-level	Fragile
Cyclic Types ¹		✓		✓	✓	✓
Elaboration Target			RTG		F_{μ}^{ω}	
Type Soundness		✓	✓	✓	✓	

Comparison with Related Work

Feature	Moscow ML Russo (2001)	Traviata Nakata et al. (2006)	RMC Dreyer (2007)	$\lambda_{\text{abs}}^{\text{rec}}$ Im et al. (2011)	M_{μ}^{ω} Ours	OCaml
Generative Functors	✓	✓	✓	✓	✓	✓
Applicative Functors		✓			✓	✓
Higher-Order Functor	✓		✓	✓	✓	✓
External Signature	Required	Inferred	Required	Required	Required	Required
Double Vision	Avoided	Manual	Full	Full	Top-level	Fragile
Cyclic Types ¹		✓		✓	✓	✓
Elaboration Target			RTG		F_{μ}^{ω}	
Type Soundness		✓	✓	✓	✓	

Comparison with Related Work

Feature	Moscow ML Russo (2001)	Traviata Nakata et al. (2006)	RMC Dreyer (2007)	$\lambda_{\text{abs}}^{\text{rec}}$ Im et al. (2011)	M_{μ}^{ω} Ours	OCaml
Generative Functors	✓		✓	✓	✓	✓
Applicative Functors		✓			✓	✓
Higher-Order Functor	✓		✓	✓	✓	✓
External Signature	Required	Inferred	Required	Required	Required	Required
Double Vision	Avoided	Manual	Full	Full	Top-level	Fragile
Cyclic Types ¹		✓		✓	✓	✓
Elaboration Target			RTG		F_{μ}^{ω}	
Type Soundness		✓	✓	✓	✓	

Comparison with Related Work

Feature	Moscow ML Russo (2001)	Traviata Nakata et al. (2006)	RMC Dreyer (2007)	$\lambda_{\text{abs}}^{\text{rec}}$ Im et al. (2011)	M_{μ}^{ω} Ours	OCaml
Generative Functors	✓		✓	✓	✓	✓
Applicative Functors		✓			✓	✓
Higher-Order Functor	✓		✓	✓	✓	✓
External Signature	Required	Inferred	Required	Required	Required	Required
Double Vision	Avoided	Manual	Full	Full	Top-level	Fragile
Cyclic Types ¹		✓		✓	✓	✓
Elaboration Target			RTG		F_{μ}^{ω}	
Type Soundness		✓	✓	✓	✓	

¹We refer to type cycles that arise from recursive module definitions, not recursive datatypes, which can be handled via standard, decidable iso-recursive types.

Comparison with Related Work

Feature	Moscow ML Russo (2001)	Traviata Nakata et al. (2006)	RMC Dreyer (2007)	$\lambda_{\text{abs}}^{\text{rec}}$ Im et al. (2011)	M_{μ}^{ω} Ours	OCaml
Generative Functors	✓		✓	✓	✓	✓
Applicative Functors		✓			✓	✓
Higher-Order Functor	✓		✓	✓	✓	✓
External Signature	Required	Inferred	Required	Required	Required	Required
Double Vision	Avoided	Manual	Full	Full	Top-level	Fragile
Cyclic Types ¹		✓		✓	✓	✓
Elaboration Target			RTG		F_{μ}^{ω}	
Type Soundness		✓	✓	✓	✓	

For practical implementation, Traviata and $\lambda_{\text{abs}}^{\text{rec}}$ require type cycles to be cut by opaque definitions, M_{μ}^{ω} requires them to be elaborated to F_{μ}^{ω} recursive types at base kind.

¹We refer to type cycles that arise from recursive module definitions, not recursive datatypes, which can be handled via standard, decidable iso-recursive types.

Comparison with Related Work

Feature	Moscow ML Russo (2001)	Traviata Nakata et al. (2006)	RMC Dreyer (2007)	$\lambda_{\text{abs}}^{\text{rec}}$ Im et al. (2011)	M_{μ}^{ω} Ours	OCaml
Generative Functors	✓		✓	✓	✓	✓
Applicative Functors		✓			✓	✓
Higher-Order Functor	✓		✓	✓	✓	✓
External Signature	Required	Inferred	Required	Required	Required	Required
Double Vision	Avoided	Manual	Full	Full	Top-level	Fragile
Cyclic Types ¹		✓		✓	✓	✓
Elaboration Target			RTG		F_{μ}^{ω}	
Type Soundness		✓	✓	✓	✓	

Conclusion

- M_{μ}^{ω} provides a **principled foundation** for recursive modules.
- **Unified treatment** for generative/applicative functors
new primitives: fixre & compact
- **Double Vision** is resolved following RMC by Dreyer,
but M_{μ}^{ω} suggests a source-level encoding.
- **Future Work:**
 - Extending to deep Double Vision to support nested sealed submodules (or not).
 - Source level path-based type system for recursive modules à la ZipML.

Conclusion

- M_{μ}^{ω} provides a **principled foundation** for recursive modules.
- **Unified treatment** for generative/applicative functors
new primitives: fixre & compack
- **Double Vision** is resolved following RMC by Dreyer,
but M_{μ}^{ω} suggests a source-level encoding.
- **Future Work:**
 - Extending to deep Double Vision to support nested sealed submodules (or not).
 - Source level path-based type system for recursive modules à la ZipML.

Thank you!

References I

- D. Dreyer. A type system for recursive modules. *SIGPLAN Not.*, 42(9):289–302, oct 2007. ISSN 0362-1340. doi: 10.1145/1291220.1291196. URL <https://doi.org/10.1145/1291220.1291196>.
- H. Im, K. Nakata, J. Garrigue, and S. Park. A syntactic type system for recursive modules. *SIGPLAN Not.*, 46(10):993–1012, oct 2011. ISSN 0362-1340. doi: 10.1145/2076021.2048141. URL <https://doi.org/10.1145/2076021.2048141>.
- K. Nakata and J. Garrigue. Recursive modules for programming. *SIGPLAN Not.*, 41(9):74–86, Sept. 2006. ISSN 0362-1340. doi: 10.1145/1160074.1159813. URL <https://doi.org/10.1145/1160074.1159813>.
- C. V. Russo. Recursive structures for standard ml. *SIGPLAN Not.*, 36(10):50–61, oct 2001. ISSN 0362-1340. doi: 10.1145/507669.507644. URL <https://doi.org/10.1145/507669.507644>.