Fulfilling OCaml modules with transparency

Supplementary material

BLAUDEAU CLEMENT, Inria, France and Université Paris Cité, France
DIDIER RÉMY, Inria, France
GABRIEL RADANNE, Inria, France

This document contains the supplementary material for the *Fulfilling OCaml modules with transparency* [1]. §1 contains a few small additional illustrative examples. This document also provides the complete set of typing rules (§2) and anchoring rules (§3) for M^ω, typing rules for F^ω (§4) and the elaboration rules of M^ω into F^ω (§5). Proofs of the anchoring properties can be found in §3.

CCS Concepts: • **Software and its engineering** → *Functional languages*; *Semantics*; **Modules / packages**; • **Theory of computation** → **Type theory**; **Type structures**.

Additional Key Words and Phrases: existential types, signature avoidance, applicative functors, F-omega, ML

ACM Reference Format:
Blaudeau Clement, Didier Rémy, and Gabriel Radanne. 2024. Fulfilling OCaml modules with transparency: Supplementary material. 1, 1 (April 2024), 17 pages.

Contents

1 EXAMPLES

1.1 Example of concrete ascription

Figure 1 demonstrates a code pattern, present in the OCaml ecosystem, where concrete ascription would be useful. Currently, OCaml fails to share and identify the types `Space3D.SSet.t` and `Set(Reals).t`. This pattern arises when combining a functor (`Make3D`) that reexports its argument (`K`) with an applicative functor called twice (`Set`), once on the argument and once on the reexported argument so that the modules resulting from both applications may interact. Here, the fixed interface `VectorSpace` could not be *functorized* to explicit the dependency with the underlying field, as not all vector spaces are functors over a field. Besides, type-level sharing is not sufficient to obtain the right type equalities when calling the `Set` functor. By contrast, when the granularity of applicativity relies on static equivalence as in Moscow ML, the example would typecheck without requiring concrete ascription: there is a design trade-off between abstraction safety and flexibility.

Authors' addresses: Blaudeau Clement, Inria, Paris, France and Université Paris Cité, Paris, France, clement.blaudeau@inria.fr; Didier Rémy, Inria, Paris, France, didier.remy@inria.fr; Gabriel Radanne, Inria, Lyon, France, gabriel.radanne@inria.fr.

2024. ACM XXXX-XXXX/2024/4-ART
https://doi.org/

```
1  module type Field = ...
2  module type VectorSpace = sig
3    module Scalar : Field
4    ... (** more fields *)
5  end
6  module Set(Y:...) = ...
7  module LinAlgebra(V:VectorSpace) = struct
8    module SSet = Set(V.Scalar)  ...
9  end
```

```
10 module Make3D(K:Field) = (struct
11   module Scalar = K
12   ... (** built from K *)
13 end : sig
14   module Scalar : (= K < Field)  ...
15 end)
16 module Reals = ...
17 module Space3D = LinAlgebra(Make3D(Reals))
18 (** Space3D.SSet.t =? Set(Reals).t *)
```

Fig. 1. An example of code pattern where transparent ascription is necessary. On the left-hand side, `VectorSpace` defines an interface for vector spaces which contains a sub-module `Scalar` for the field of scalar numbers. The functor `LinAlgebra` (line 7) uses a vector space to define linear algebra operations, one of them being sets of scalar numbers. At some other point in the development (line 10), 3D vector spaces are built directly from any field `K` via the functor `Make3D`. Its signature contains a transparent ascription on its parameter `K`. Finally, on line 17, the module `Space3D` implements linear algebra for the vector space $\mathbb{R}^3$. We want the inner sets `Space3D.SSet.t`, and `Set(Reals).t` to be compatible. This requires the aliasing information to be kept between the parameter and the body of the functor `Make3D`.

1.2 Signature avoidance

All examples of this section refer to a signature `S` defined as:

```
1  module type S = sig type t end
```

1.2.1 *Valid OCaml syntax.*

Example 1.1. We give a basic example of signature avoidance similar to [1, Figure 3], but in *valid* OCaml syntax. The anonymous projection is replaced by an anonymous functor call. The key is that the type field of the argument `Y.t` is inaccessible outside of the functor's body and must therefore be avoided.

```
1  module M =
2    (functor (X: S) -> struct
3      type a = X.t * bool
4      type b = X.t * int
5      end)
6    (struct type t = int end : S)

7  let f ((x,_): M.a) = ((x,42): M.b)
```

As the typechecker abstracts both `a` and `b`, it fails at line 7 with

```
1  Error: This pattern matches values of type 'a * 'b
2         but a pattern was expected which matches values of type M.a
```

1.2.2 *Solvable cases.*

We show two examples of solvable cases that are not correctly handled by the current OCaml typechecker (over-abstraction), but would be covered by our anchoring algorithm. For each, we give the code, the inferred signature given by the OCaml typechecker, and the corresponding signature obtained by M^ω typing and anchoring.

Example 1.2. A modified version of Example 1.1 with a solvable case of signature avoidance. Here, abstracting `a` and rewriting the type declaration for `b` as `type b = a * int` would keep all type-sharing:

Source code

```
module M =
 (functor (X: S) -> struct
   type a = X.t
   type b = X.t * int
   end)
 (struct type t = int end : S)
```

Inferred signature

$\texttt{module}\, M : \texttt{sig}_A\ \texttt{type}\, a = A.a\ \texttt{type}\, b = A.b\ \texttt{end}$

Anchored signature

$\texttt{module}\, M : \texttt{sig}_A\ \texttt{type}\, a = A.a\ \texttt{type}\, b = A.a \times \mathit{int}\ \texttt{end}$

Example 1.3. A variant of the previous example at the module level, with module aliases instead of type equalities:

Source code

```
module M =
 (functor (X: S) -> struct
   module X1 = Y
   module X2 = Y
 end)
 (struct type t = int end : S)
```

Inferred signature

$\texttt{module}\, M : \texttt{sig}_A\ \texttt{module}\, X_1 : \texttt{S}\ \texttt{module}\, X_2 : \texttt{S}\ \texttt{end}$

Anchored signature

$\texttt{module}\, M : \texttt{sig}_A\ \texttt{module}\, X_1 : \texttt{S}\ \texttt{module}\, X_2 : (= A.X_1 \leq \texttt{S})\ \texttt{end}$

1.2.3 *Module encoding in F^ω.*

Example 1.4. A simple module M with three type fields, on the left-hand side. The raw encoding (after reduction of administrative let-bindings) on the right-hand side shows how abstract types are shared between components via lifting.

Source code

$\texttt{module}\, M = \texttt{struct}_A\ \texttt{type}\, t = A.t\ \texttt{type}\, u = A.u\ \texttt{type}\, v = A.t \times A.u\ \texttt{end}$

Encoding of e

$$e = \mathsf{lift}^{\blacktriangledown}\ \langle \alpha,\ x_1 = \mathsf{pack}\ \langle (), \{\ell_t = \langle\!\langle () \rangle\!\rangle\}\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\alpha.\ \{\ell_t : \langle\!\langle \alpha \rangle\!\rangle\}$$
$$@\ \mathsf{lift}^{\blacktriangledown}\ \langle \beta,\ x_2 = \mathsf{pack}\ \langle (), \{\ell_u = \langle\!\langle () \rangle\!\rangle\}\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\beta.\ \{\ell_u : \langle\!\langle \beta \rangle\!\rangle\}$$
$$@\ \{\ell_v = \langle\!\langle \alpha \times \beta \rangle\!\rangle\}\rangle\rangle$$

Signature of e

$$\mathcal{C} = \exists \alpha, \beta.\ \{\ell_t : \langle\!\langle \alpha \rangle\!\rangle\ ;\ \ell_u : \langle\!\langle \beta \rangle\!\rangle\ ;\ \ell_v : \langle\!\langle \alpha \times \beta \rangle\!\rangle\}$$

2 M^ω SYSTEM

In this section we give the complete set of typing rules of the M^ω system.

2.1 Subtyping

2.1.1 *Signature subtyping*

$\boxed{\Gamma \vdash \mathcal{C} \leq \mathcal{C}'}$

$$\frac{\overline{\mathcal{D}_0} \subseteq \overline{\mathcal{D}} \qquad \Gamma \vdash \overline{\mathcal{D}_0} \leq \overline{\mathcal{D}}'}{\Gamma \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \leq \mathsf{sig}\ \overline{\mathcal{D}}'\ \mathsf{end}}\ \textsc{M-Sub-Sig-Struct} \qquad \frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} \leq \mathcal{C}'\left[\overline{\alpha}' \mapsto \overline{\tau}\right]}{\Gamma \vdash () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \leq () \to \exists^{\blacktriangledown}\overline{\alpha}'.\mathcal{C}'}\ \textsc{M-Sub-Sig-GenFct}$$

$$\frac{\Gamma, \overline{\alpha}' \vdash \mathcal{C}'_a \leq \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma, \overline{\alpha}' \vdash \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] \leq \mathcal{C}'}{\Gamma \vdash \forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \leq \forall \overline{\alpha}'.\mathcal{C}'_a \to \mathcal{C}'}\ \textsc{M-Sub-Sig-AppFct}$$

2.1.2 *Declaration subtyping* $\boxed{\Gamma \vdash \mathcal{D} \leq \mathcal{D}'}$

M-Sub-Decl-Val
$$\Gamma \vdash (\mathsf{val}\ x : \tau) \leq (\mathsf{val}\ x : \tau)$$

M-Sub-Decl-Type
$$\Gamma \vdash (\mathsf{type}\ t = \tau) \leq (\mathsf{type}\ t = \tau)$$

M-Sub-Decl-Mod
$$\frac{\Gamma \vdash \mathcal{C} \leq \mathcal{C}'}{\Gamma \vdash (\mathsf{module}\ X : \mathcal{C}) \leq (\mathsf{module}\ X : \mathcal{C}')}$$

M-Sub-Decl-ModType
$$\frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} \leq \mathcal{C}' \qquad \Gamma, \overline{\alpha} \vdash \mathcal{C}' \leq \mathcal{C}}{\Gamma \vdash (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}) \leq (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}')}$$

2.2 Typing

2.2.1 *Signature typing* $\boxed{\Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}}$

M-Typ-Sig-ModType
$$\frac{\Gamma \vdash P : \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C} \in \overline{\mathcal{D}}}{\Gamma \vdash P.T : \lambda\overline{\alpha}.\mathcal{C}}$$

M-Typ-Sig-LocalModType
$$\frac{A.(T :\ \mathsf{module\ type}\ \lambda\overline{\alpha}.\mathcal{C}) \in \Gamma}{\Gamma \vdash A.T : \lambda\overline{\alpha}.\mathcal{C}}$$

M-Typ-Sig-GenFct
$$\frac{\Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash () \to \mathsf{S} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}$$

M-Typ-Sig-AppFct
$$\frac{\Gamma \vdash \mathsf{S}_a : \lambda\overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a \vdash \mathsf{S} : \lambda\overline{\beta}.\mathcal{C}}{\Gamma \vdash (Y : \mathsf{S}_a) \to \mathsf{S} : \lambda\overline{\beta'}.\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]}$$

M-Typ-Sig-Str
$$\frac{\Gamma \vdash_A \overline{\mathsf{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathsf{sig}_A\ \overline{\mathsf{D}}\ \mathsf{end} : \lambda\overline{\alpha}.\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}}$$

M-Typ-Sig-Con
$$\frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} \leq \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa}}{\Gamma \vdash (= P \leq \mathsf{S}) : \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]}$$

2.2.2 *Declaration typing* $\boxed{\Gamma \vdash_A \mathsf{D} : \lambda\overline{\alpha}.\mathcal{D}}$

M-Typ-Decl-Val
$$\frac{\Gamma \vdash \mathsf{u} : \tau}{\Gamma \vdash_A (\mathsf{val}\ x : \mathsf{u}) : (\mathsf{val}\ x : \tau)}$$

M-Typ-Decl-Type
$$\frac{\Gamma \vdash \mathsf{u} : \tau}{\Gamma \vdash_A (\mathsf{type}\ t = \mathsf{u}) : (\mathsf{type}\ t = \tau)}$$

M-Typ-Decl-TypeAbs
$$\Gamma \vdash_A A.(\mathsf{type}\ t = t) : \lambda\alpha.(\mathsf{type}\ t = \alpha)$$

M-Typ-Decl-Mod
$$\frac{\Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathsf{module}\ X : \mathsf{S}) : \lambda\overline{\alpha}.(\mathsf{module}\ X : \mathcal{C})}$$

M-Typ-Decl-ModType
$$\frac{\Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathsf{module\ type}\ T = \mathsf{S}) : (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C})}$$

M-Typ-Decl-Empty
$$\Gamma \vdash_A \varnothing : \varnothing$$

M-Typ-Decl-Seq
$$\frac{\Gamma \vdash_A \mathsf{D} : \lambda\overline{\alpha}_1.\mathcal{D} \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D} \vdash_A \overline{\mathsf{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \vdash_A \mathsf{D}, \overline{\mathsf{D}} : \lambda\overline{\alpha}_1\,\overline{\alpha}.\mathcal{D}, \overline{\mathcal{D}}}$$

2.2.3 *Core type checking extension* $\boxed{\Gamma \vdash \mathsf{u} : \tau}$

M-Typ-Type-Path
$$\frac{\Gamma \vdash P : \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{type}\ t = \tau \in \overline{\mathcal{D}}}{\Gamma \vdash P.t : \tau}$$

M-Typ-Type-Local
$$\frac{A.(t :\ \mathsf{type}\ \tau) \in \Gamma}{\Gamma \vdash A.t : \tau}$$

2.2.4 *Module typing*

$\boxed{\Gamma \vdash \mathsf{M} : \exists^{\diamond} \overline{\alpha}.\mathcal{C}}$

$$\textsc{M-Typ-Mod-Var}\ \frac{(Y : \mathcal{C}) \in \Gamma}{\Gamma \vdash Y : \mathcal{C}} \qquad \textsc{M-Typ-Mod-Local}\ \frac{A.(X :\ \mathsf{module}\ \mathcal{C}) \in \Gamma}{\Gamma \vdash A.X : \mathcal{C}}$$

$$\textsc{M-Typ-Mod-Struct}\ \frac{\Gamma \vdash_A \overline{\mathsf{B}} : \exists^{\diamond} \overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{struct}_A\ \overline{\mathsf{B}}\ \mathsf{end} : \exists^{\diamond} \overline{\alpha}.\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}}$$

$$\textsc{M-Typ-Mod-Ascr}\ \frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathsf{S} : \lambda \overline{\alpha}.\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} \leq \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa}}{\Gamma \vdash (P : \mathsf{S}) : \exists^{\triangledown} \overline{\alpha}.\mathcal{C}'} \qquad \textsc{M-Typ-Mod-GenFct}\ \frac{\Gamma \vdash \mathsf{M} : \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}}{\Gamma \vdash () \to \mathsf{M} : () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}}$$

$$\textsc{M-Typ-Mod-AppFct}\ \frac{\Gamma \vdash \mathsf{S}_a : \lambda \overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, (Y : \mathcal{C}_a) \vdash \mathsf{M} : \exists^{\triangledown} \overline{\beta}.\mathcal{C}}{\Gamma \vdash (Y : \mathsf{S}_a) \to \mathsf{M} : \exists^{\triangledown} \overline{\beta'}.\forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]} \qquad \textsc{M-Typ-Mod-AppGen}\ \frac{\Gamma \vdash P : () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}}{\Gamma \vdash P() : \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}}$$

$$\textsc{M-Typ-Mod-AppApp}\ \frac{\Gamma \vdash P : \forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \qquad \Gamma \vdash P' : \mathcal{C}' \qquad \Gamma \vdash \mathcal{C}' \leq \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa}}{\Gamma \vdash P(P') : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]} \qquad \textsc{M-Typ-Mod-Seal}\ \frac{\Gamma \vdash \mathsf{M} : \exists^{\triangledown} \overline{\alpha}.\mathcal{C}}{\Gamma \vdash \mathsf{M} : \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}}$$

$$\textsc{M-Typ-Mod-Proj}\ \frac{\Gamma \vdash \mathsf{M} : \exists^{\diamond} \overline{\alpha}.\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{module}\ X : \mathcal{C} \in \overline{\mathcal{D}} \qquad \overline{\alpha}' = \mathsf{fv}(\mathcal{C}) \cap \overline{\alpha}}{\Gamma \vdash \mathsf{M}.X : \exists^{\diamond} \overline{\alpha}'.\mathcal{C}}$$

2.2.5 *Binding typing*

$\boxed{\Gamma \vdash \mathsf{B} : \exists^{\diamond} \overline{\alpha}.\mathcal{D}}$

$$\textsc{M-Typ-Bind-Let}\ \frac{\Gamma \vdash^{\diamond} \mathsf{e} : \tau}{\Gamma \vdash^{\diamond}_A (\mathtt{let}\, x = \mathsf{e}) : (\mathsf{val}\ x : \tau)} \qquad \textsc{M-Typ-Bind-Type}\ \frac{\Gamma \vdash \mathsf{u} : \tau}{\Gamma \vdash^{\diamond}_A (\mathtt{type}\, t = \mathsf{u}) : (\mathsf{type}\ t = \tau)} \qquad \textsc{M-Typ-Bind-AbsType}\ \Gamma \vdash_A (\mathtt{type}\, t = A.t) : \exists^{\diamond} \alpha.(\mathsf{type}\ t = \alpha)$$

$$\textsc{M-Typ-Bind-Mod}\ \frac{\Gamma \vdash \mathsf{M} : \exists^{\diamond} \overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module}\, X = \mathsf{M}) : (\exists^{\diamond} \overline{\alpha}.\mathsf{module}\ X : \mathcal{C})} \qquad \textsc{M-Typ-Bind-ModType}\ \frac{\Gamma \vdash^{\diamond} \mathsf{S} : \lambda \overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module\ type}\, T = \mathsf{S}) : (\mathsf{module\ type}\ T = \lambda \overline{\alpha}.\mathcal{C})}$$

$$\textsc{M-Typ-Bind-Empty}\ \Gamma \vdash_A \varnothing : \varnothing \qquad \textsc{M-Typ-Bind-Seq}\ \frac{\Gamma \vdash_A \mathsf{B} : \exists^{\diamond} \overline{\alpha}_1.\mathcal{D} \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D} \vdash_A \overline{\mathsf{B}} : \exists^{\diamond} \overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \vdash_A \mathsf{B}, \overline{\mathsf{B}} : \exists^{\diamond} \overline{\alpha}_1, \overline{\alpha}.\mathcal{D}, \overline{\mathcal{D}}}$$

2.2.6 *Core expression typing extension*

$\boxed{\Gamma \vdash^{\diamond} \mathsf{e} : \tau}$

$$\textsc{M-Typ-Type-Path}\ \frac{\Gamma \vdash P : \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{val}\ x : \tau \in \overline{\mathcal{D}}}{\Gamma \vdash^{\diamond} P.x : \tau} \qquad \textsc{M-Typ-Type-Local}\ \frac{A.(x :\ \mathsf{val}\ \tau) \in \Gamma}{\Gamma \vdash^{\diamond} A.x : \tau}$$

2.3 Proof sketch of Theorem 3.1

$$\left.\begin{array}{l}\Gamma \vdash [\![\mathsf{M}_1]\!] : \mathsf{sig\ module}\ \mathit{Val} : \mathcal{C}_1\ \ \mathsf{type}\ \mathit{id} = \tau\ \mathsf{end} \\ \Gamma \vdash [\![\mathsf{M}_2]\!] : \mathsf{sig\ module}\ \mathit{Val} : \mathcal{C}_2\ \ \mathsf{type}\ \mathit{id} = \tau\ \mathsf{end}\end{array}\right\} \implies \exists \mathcal{C}_0, \left\{\begin{array}{l}\Gamma \vdash \mathcal{C}_0 \leq \mathcal{C}_1 \\ \Gamma \vdash \mathcal{C}_0 \leq \mathcal{C}_2\end{array}\right.$$

For this proof, we consider a slightly modified system called $\mathsf{M}^{\omega}_{<:}$ based on F^{ω} extended with bounded quantification. We identify M^{ω} types and M^{ω} signatures and use τ and $\mathcal{C}$ interchangeably in this section. The system $\mathsf{M}^{\omega}_{<:}$ is built from M^{ω} as follows:

(1) We extend the quantifiers $\exists^{\diamond}$, $\forall$, and λ to support bounded quantification for abstract types that serve as identities (the other types being bound by the top bound $\top$).
(2) We modify the typing and subtyping rules accordingly. Omitting the rules that do not feature bounds or simply thread them from the premise to the conclusion, only three typing rules are

affected, as they now feature an additional subtyping condition as their premise (using gray background to emphasize the differences):

$$\textsc{MS-Typ-Sig-Con}\quad \frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathsf{S} : \lambda(\overline{\alpha} \leq \overline{\tau}').\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} \leq \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa} \qquad \Gamma \vdash \overline{\tau} \leq \overline{\tau}'}{\Gamma \vdash (= P \leq \mathsf{S}) : \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]}$$

$$\textsc{MS-Typ-Mod-Ascr}\quad \frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathsf{S} : \lambda(\overline{\alpha} \leq \overline{\tau}').\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} \leq \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa} \qquad \Gamma \vdash \overline{\tau} \leq \overline{\tau}'}{\Gamma \vdash (P : \mathsf{S}) : \exists^{\triangledown}(\overline{\alpha} \leq \overline{\tau}').\mathcal{C}'}$$

$$\textsc{MS-Typ-Mod-AppApp}\quad \frac{\Gamma \vdash P : \forall(\overline{\alpha} \leq \overline{\tau}').\mathcal{C}_a \to \mathcal{C} \qquad \Gamma \vdash P' : \mathcal{C}' \qquad \Gamma \vdash \mathcal{C}' \leq \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa} \qquad \Gamma \vdash \overline{\tau} \leq \overline{\tau}'}{\Gamma \vdash P(P') : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]}$$

(3) We modify the typing rules for introducing abstract types (MS-Typ-Decl-TypeAbs and MS-Typ-Bind-AbsType) by distinguishing identity tags (Rules MS-Typ-Decl-TypeAbsId and MS-Typ-Bind-AbsTypeId) from other abstract types. While abstract types are simply bound by $\top$, identity tags are bound by the signature of the associated value.

$$\textsc{MS-Typ-Decl-TypeAbs}\quad \frac{t \neq id}{\Gamma \vdash_A (\mathsf{type}\, t = A.t) : \lambda(\alpha \leq \top).(\mathsf{type}\ t = \alpha)} \qquad \textsc{MS-Typ-Decl-TypeAbsId}\quad \frac{\Gamma \vdash A.Val : \mathcal{C}}{\Gamma \vdash_A (\mathsf{type}\ id = A.id) : \lambda(\alpha \leq \mathcal{C}).(\mathsf{type}\ id = \alpha)}$$

$$\textsc{MS-Typ-Bind-AbsType}\quad \frac{t \neq id}{\Gamma \vdash_A (\mathsf{type}\, t = A.t) : \exists^{\diamond}(\alpha \leq \top).(\mathsf{type}\ t = \alpha)} \qquad \textsc{MS-Typ-Bind-AbsTypeId}\quad \frac{\Gamma \vdash A.Val : \mathcal{C}}{\Gamma \vdash_A (\mathsf{type}\ id = A.id) : \exists^{\diamond}(\alpha \leq \mathcal{C}).(\mathsf{type}\ id = \alpha)}$$

(4) Subtyping is extended with a new rule MS-Sub-Bound for subtyping bound variables (MS-Sub-Bound-Star is just a particular case of MS-Sub-Bound):

$$\textsc{MS-Sub-Bound}\quad \frac{\varphi \leq \lambda(\overline{\alpha} \leq \overline{\tau}').\mathcal{C} \in \Gamma \qquad \Gamma \vdash \overline{\tau} \leq \overline{\tau}'}{\Gamma \vdash (\varphi\, \overline{\tau}) \leq \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]} \qquad \textsc{MS-Sub-Bound-Star}\quad \frac{\alpha \leq \mathcal{C} \in \Gamma}{\Gamma \vdash \alpha \leq \mathcal{C}}$$

Besides, the two following rules for subtyping between functors are also extended with additional premises to justify the instantiation of abstract types:

$$\textsc{MS-Sub-Sig-GenFct}\quad \frac{\Gamma, \overline{\alpha} \leq \overline{\tau} \vdash \mathcal{C} \leq \mathcal{C}'\left[\overline{\alpha}' \mapsto \overline{\tau_1}\right] \qquad \Gamma, \overline{\alpha} \leq \overline{\tau} \vdash \overline{\tau_1} \leq \overline{\tau}'}{\Gamma \vdash () \to \exists^{\blacktriangledown}(\overline{\alpha} \leq \overline{\tau}).\mathcal{C} \leq () \to \exists^{\blacktriangledown}(\overline{\alpha}' \leq \overline{\tau}').\mathcal{C}'}$$

$$\textsc{MS-Sub-Sig-AppFct}\quad \frac{\Gamma, \overline{\alpha}' \leq \overline{\tau}' \vdash \mathcal{C}'_a \leq \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau_1}] \qquad \Gamma, \overline{\alpha}' \leq \overline{\tau}' \vdash \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau_1}] \leq \mathcal{C}' \qquad \Gamma, \overline{\alpha}' \leq \overline{\tau}' \vdash \overline{\tau_1} \leq \overline{\tau}}{\Gamma \vdash \forall(\overline{\alpha} \leq \overline{\tau}).\mathcal{C}_a \to \mathcal{C} \leq \forall(\overline{\alpha}' \leq \overline{\tau}').\mathcal{C}'_a \to \mathcal{C}'}$$

Subtyping in $\mathsf{M}^{\omega}_{<:}$ remains transitive.

The proof then proceeds in two steps:

(1) We first show that the $\mathsf{M}^{\omega}_{<:}$ maintains an *identity-tag well-formedness invariant* in typing derivations: If $\Gamma \vdash \mathsf{sig\ type}\ id = \tau\ \mathsf{module}\ Val : \mathcal{C}\ \mathsf{end} : \mathsf{wf}_{id}$ then $\Gamma \vdash \tau \leq \mathcal{C}$ (**1**). To do so, we reinforced well-formedness for identity tags by changing the rule to

$$\frac{\Gamma \vdash \mathcal{C} : \mathsf{wf} \qquad \Gamma \vdash \tau \leq \mathcal{C}}{\Gamma \vdash \mathsf{sig\ type}\ id = \tau\ \mathsf{module}\ Val : \mathcal{C}\ \mathsf{end} : \mathsf{wf}}$$

This defines a stronger well-formedness judgment $\Gamma \vdash \mathcal{C} : \mathsf{wf}_{id}$. We show that $\mathsf{M}^{\omega}_{<:}$ typing judgments for modules and signatures always produce signatures that preserves identity-tag

well-formed. That is, $\vdash \Gamma : \mathsf{wf}_{id}$ and either $\Gamma \vdash \mathtt{S} : \lambda(\overline{\alpha} \leq \overline{\mathcal{C}}).\mathcal{C}'$ or $\Gamma \vdash \mathtt{M} : \exists^{\diamond}(\overline{\alpha} \leq \overline{\mathcal{C}}).\mathcal{C}'$ implies $\Gamma, \overline{\alpha} \leq \overline{\mathcal{C}} \vdash \mathcal{C}' : \mathsf{wf}_{id}$.
Therefore, we may restrict $\mathsf{M}^{\omega}_{<:}$ derivations to use the identity-tag well-formedness invariant as long as we start with a identity well-formed environment. By inversion of well-formedness, this ensures the invariant (1).

(2) Using the previous invariant, we then show that typability in M^{ω} implies typability in $\mathsf{M}^{\omega}_{<:}$.

The proof of the first step proceeds by a simple induction over the typing derivation: identities are either introduced fresh, in which case the bound is equal to the signature of the corresponding *Val*-field, or obtained via subtyping, in which case we use transitivity of subtyping.

The rest of this section is dedicated to the proof of the second step. The only differences between the original and the enhanced typing systems are the addition of subtyping bounds and subtyping relations between the bounds. The core of the proof is to show that these are actually not restrictive, which follows from two key facts: (1) subtyping is only done with a right-hand side signature that *comes from a source signature*, i.e., that is the result of typing a source signature; and (2) such signatures always contain the bounds of their identity tags in at least one positive occurrence.

Properties of typed source signatures. M^{ω} signatures obtained via typing of a source signature (referred to as TSS in the following) are a strict subset of identity-well-formed signatures. Bounds allow us to store the signature $\mathcal{C}$ of the original module that introduced an identity tag α, and the signature $\mathcal{C}'$ of every module that shares the identity α is a supertype of the original signature $\mathcal{C}$. In a TSS, the bound of an identity tag α is *exactly* the signature $\mathcal{C}$ of the first module occurrence with identity tag α, and therefore, the bound always appears *explicitly* in the signature. By contrast, in a signature obtained via module typing, the access to the original module of signature $\mathcal{C}$ may have been lost, typically by projection, leaving only modules whose signatures are supertypes of $\mathcal{C}$.

First-order example. Before diving into the proof by induction, we consider the typing of a basic source signature S:

$$\Gamma \vdash \mathtt{S} : \lambda(\alpha \leq \mathcal{C}_{\alpha}).\mathcal{C}$$

There must be a subterm of $\mathcal{C}$ at a positive occurrence that is equal to:

$$\mathsf{sig\ module}\ \mathit{Val} : \mathcal{C}_{\alpha}\ \mathsf{type}\ \mathit{id} = \alpha\ \mathsf{end}$$

When subtyping this signature with another one in the enhanced system,

$$\Gamma \vdash \lambda(\alpha \leq \mathcal{C}_{\alpha}).\mathcal{C} \leq \lambda(\beta \leq \mathcal{C}_{\beta}).\mathcal{C}'$$

there is a new subtyping check between the bounds:

$$\Gamma \vdash \mathcal{C}_{\beta} \leq \mathcal{C}_{\alpha} \qquad (\mathbf{1})$$

Whenever the subtyping between $\mathcal{C}'$ and $\mathcal{C}$ succeeded in the original system, $\mathcal{C}'$ features a subterm of the form

$$\mathsf{sig\ module}\ \mathit{Val} : \mathcal{C}_0\ \mathsf{type}\ \mathit{id} = \beta\ \mathsf{end}$$

By subtyping, we have $\Gamma \vdash \mathcal{C}_0 \leq \mathcal{C}_{\alpha}$. Thanks to the invariant of the first part of the proof, we know that $\Gamma \vdash \mathcal{C}_{\beta} \leq \mathcal{C}_0$. Transitivity of subtyping ensures (1). Hence subtyping also succeeds in the enhanced system.

Proof by induction. We prove by induction over the typing derivation that typing and subtyping in M^{ω} implies typing and subtyping in the enhanced system. Cases for unchanged rules, or rules that just thread the bounds from the premise to the conclusion are immediate. The only interesting cases are the three typing rules and two subtyping rules shown above have an additional premise, which we prove is actually implied by the other premises.

Typing M-Typ-Sig-Con. The M^{ω} derivation ends with the following rule.

$$\text{M-Typ-Sig-Con}\quad \frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} \leq \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa}}{\Gamma \vdash (= P \leq \mathsf{S}) : \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]}$$

For simplification of presentation that $\overline{\alpha}$ is a single variable α.

We show that we can rebuild a $\mathsf{M}^{\omega}_{<:}$ derivation:

$$\text{MS-Typ-Sig-Con}\quad \frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathsf{S} : \lambda(\alpha \leq \tau').\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} \leq \mathcal{C}'[\alpha \mapsto \tau]\ (\mathbf{2}) \qquad \Gamma \vdash \tau : \kappa \qquad \Gamma \vdash \tau \leq \tau'\ (\mathbf{3})}{\Gamma \vdash (= P \leq \mathsf{S}) : \mathcal{C}'[\alpha \mapsto \tau]}$$

All the premises but (3), hence including (2), follow by induction hypothesis. The remaining goal is to show that the additional condition (3) actually follows from (2).

For this purpose, we define $[\mathcal{C}]^+$, the *positive* declarations of a signature $\mathcal{C}$, as the flattened list of declarations in strict positive positions inside $\mathcal{C}$ (without entering inside generative functors or module types), taken in the usual binding order, as follows:

$$\begin{aligned}
[\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}]^+ &= \overline{[\mathcal{D}]^+} && \text{(Structural signature)}\\
[\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}]^+ &= \forall\overline{\alpha}.[\mathcal{C}]^+ && \text{(Applicative functor)}\\
[() \to _]^+ &= \varnothing && \text{(Generative functor)}\\
[\mathsf{module}\ X : \mathcal{C}]^+ &= (\mathsf{module}\ X : \mathcal{C}), [\mathcal{C}]^+ && \text{(Submodule declaration)}\\
[\mathcal{D}]^+ &= \mathcal{D} && \text{(Other declarations)}
\end{aligned}$$

Declarations inside applicative functors are universally quantified. A key observation is that a signature in TSS form always contain the bound of its identity type among its positive declarations:

$$\begin{aligned}
\Gamma \vdash \mathsf{S} : \lambda(\alpha \leq \tau).\mathcal{C} &\implies (\mathsf{module}\ \mathit{Val} : \tau) \in [\mathcal{C}]^+ \wedge \mathsf{type}\ \mathit{id} = \alpha \in [\mathcal{C}]^+\\
\Gamma \vdash \mathsf{S} : \lambda(\alpha \leq \lambda\overline{\beta}.\tau).\mathcal{C} &\implies \forall\overline{\beta}.(\mathsf{module}\ \mathit{Val} : \tau) \in [\mathcal{C}]^+ \wedge \forall\overline{\beta}.\mathsf{type}\ \mathit{id} = (\alpha\overline{\beta}) \in [\mathcal{C}]^+
\end{aligned}$$

Crucially, all the instantiations we consider feature a TSS $\mathcal{C}'$ on their right-hand side. By construction, subtyping between two signatures $\mathcal{C}$ and $\mathcal{C}'$ implies subtyping between declarations at any positive occurrence in $\mathcal{C}$ and its corresponding declaration in $\mathcal{C}'$. Ignoring applicative functors at first, this gives:

$$\Gamma \vdash \mathcal{C} \leq \mathcal{C}' \implies \forall \mathcal{D}' \in [\mathcal{C}']^+.\ \exists \mathcal{D} \in [\mathcal{C}]^+.\ \Gamma \vdash \mathcal{D} \leq \mathcal{D}'$$

Therefore, there exists a declaration in the positive part of $\mathcal{C}$ that is a subtype of the explicit bound of α, which appears in $[\mathcal{C}']^+$. That is,

$$\exists \mathcal{C}_0.\ \Gamma \vdash \mathsf{module}\ \mathit{Val} : \mathcal{C}_0 \leq \mathsf{module}\ \mathit{Val} : \tau'$$

This implies $\Gamma \vdash \mathcal{C}_0 \leq \tau'$. Using the invariant of the first step of the proof, we get $\Gamma \vdash \tau \leq \mathcal{C}_0$, which implies (3) by transitivity of subtyping. This argument applies to the three typing rules. It extends to higher-order declarations with universal quantification only adding universally quantified variables in the typing context Γ.

Typing rules MS-Typ-Mod-Ascr and MS-Typ-Mod-AppApp.

The former is exactly the same as the previous case. For the latter, the only change if that the bounded quantification $\lambda(\alpha \leq \tau').\mathcal{C}'$ is being replaced by $\forall(\alpha \leq \tau').\mathcal{C}'_a \to \mathcal{C}'$.

Subtyping Rule MS-Sub-Sig-AppFct. The same argument applies, just with an extended context. Restricting again to only one type variable for readability, we need to rebuild a derivation in $\mathsf{M}^{\omega}_{<:}$

that ends with:

$$\frac{\text{MS-Sub-Sig-AppFct}\quad \Gamma, \alpha' \leq \tau' \vdash \mathcal{C}'_a \leq \mathcal{C}_a[\alpha \mapsto \tau_1]\ (\mathbf{4}) \qquad \ldots \qquad \Gamma, \alpha' \leq \tau' \vdash \tau_1 \leq \tau\ (\mathbf{5})}{\Gamma \vdash \forall(\alpha \leq \tau).\mathcal{C}_a \to \mathcal{C} \leq \forall(\alpha' \leq \tau').\mathcal{C}'_a \to \mathcal{C}'}$$

where all the premises but (5) follow by induction hypothesis.

As $\mathcal{C}_a$ is a TSS, we have

$$\text{sig type } id = \alpha \text{ module } Val : \tau \text{ end } \in [\mathcal{C}_a]^+$$

Therefore, since α is not free in τ,

$$\text{sig type } id = \tau_1 \text{ module } Val : \tau \text{ end } \in [\mathcal{C}_a[\alpha \mapsto \tau_1]]^+$$

Correspondingly, we must have in $\mathcal{C}'_a$, for some signature σ:

$$\text{sig type } id = \tau_0 \text{ module } Val : \sigma \text{ end } \in [\mathcal{C}'_a]^+\ (\mathbf{6})$$

Subtyping component by component, we have, since subtyping in non-variant on type fields in general and identity type fields in particular:

$$\Gamma, \alpha' \leq \tau' \vdash \sigma \leq \tau \quad \wedge \quad \tau_1 = \tau_0$$

The identity well-formedness invariant of (6) implies:

$$\Gamma, \alpha' \leq \tau' \vdash \tau_0 \leq \sigma$$

By transitivity of subtyping we get $\Gamma, \alpha' \leq \tau' \vdash \tau_1 \leq \tau$, i.e.,. (5), as expected.

MS-Sub-Sig-GenFct. We rebuild a derivation of the form

$$\frac{\text{MS-Sub-Sig-GenFct}\quad \Gamma, \overline{\alpha} \leq \overline{\tau} \vdash \mathcal{C} \leq \mathcal{C}'[\overline{\alpha}' \mapsto \overline{\tau_1}]\ (\mathbf{7}) \qquad \Gamma, \overline{\alpha} \leq \overline{\tau} \vdash \overline{\tau_1} \leq \overline{\tau}'\ (\mathbf{8})}{\Gamma \vdash () \to \exists^{\blacktriangledown}(\overline{\alpha} \leq \overline{\tau}).\mathcal{C} \leq () \to \exists^{\blacktriangledown}(\overline{\alpha}' \leq \overline{\tau}').\mathcal{C}'}$$

The proof is similar to the previous case where (7) follows induction and (8) follows from (7) and the fact that $\mathcal{C}'$ is in TSS-form.

3 ANCHORING

In the section, we detail of the anchoring anchoring algorithm. This assumes that we have instrumented the typing rules as described in [1, §4.2]. In particular,

- Typing contexts Γ contain $\cdot$ marks that split it into a sequence of Δ's;
- We used multi-applications $\varphi\langle\overline{\alpha_1}\rangle \ldots \langle\overline{\alpha_n}\rangle$ instead of a sequence of single applications $\varphi(\overline{\alpha_1} \ldots \overline{\alpha_n})$ to remember the successive solemnization steps.
- The tagging of some signatures $\mathcal{C}^\dagger$ to prevent their type declarations to be used as anchoring points.

More precisely, the tagging $\mathcal{C}^\dagger$ of signatures (and declarations) is defined as follows:

$$\begin{array}{rcl} (\text{sig } \overline{\mathcal{D}} \text{ end})^\dagger & = & \text{sig } \overline{\mathcal{D}^\dagger} \text{ end} \\ (\forall\overline{\alpha}.\mathcal{C}_1 \to \mathcal{C}_2)^\dagger & = & \forall\overline{\alpha}.\mathcal{C}_1 \to \mathcal{C}_2 \\ (() \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C})^\dagger & = & () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}^\dagger \end{array}$$

$$\begin{array}{rcl} (\text{val } x : \tau)^\dagger & = & \text{val } x : \tau \\ (\text{type } t = \alpha)^\dagger & = & \text{type } t = \alpha \\ (\text{type } t = \varphi\langle\overline{\alpha_1}\rangle \ldots \langle\overline{\alpha_n}\rangle)^\dagger & = & \text{type } t = \varphi\langle\overline{\alpha_1}\rangle \ldots \langle\overline{\alpha_n}\rangle^\dagger \\ (\text{module } X : \mathcal{C})^\dagger & = & \text{module } X : \mathcal{C}^\dagger \\ (\text{module type } T = \mathcal{C})^\dagger & = & \text{module type } T = \mathcal{C} \end{array}$$

That is, it recursively propagates through type declarations and signatures to mark type declarations `type` $t = \tau$ as `type` $t = \tau^\dagger$, which will prevent them from being used as anchoring points, as the premise of Rule A-Decl-Anchor (see §3.3) requires t to be (equal to) a type of the form $\varphi\langle\overline{\alpha_1}\rangle \ldots \langle\overline{\alpha_n}\rangle$,

which can't be a $\tau_0^\dagger$. We ignore first order types when marking, as we chose to not restrict their anchoring points.

Rule A-Decl-Anchor uses a predicate $\mathrm{args}(\Delta)$ to retrieve the universal variables $\langle\overline{\alpha}_1\rangle \ldots \langle\overline{\alpha}_n\rangle$ from Δ, i.e., the sub-sequence of Δ compose of all $\langle\overline{\alpha}_i\rangle$'s that immediately precede a functor parameter in Δ.

Finally, we change the typing rule for functor application to add a mark to the resulting signature:

$$\textsc{M-Typ-Mod-AppApp}\quad \frac{\Gamma \vdash P : \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \qquad \Gamma \vdash P' : \mathcal{C}' \qquad \Gamma \vdash \mathcal{C}' \leq \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa}}{\Gamma \vdash P(P') : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]^\dagger}$$

Remark. Type variables in Γ are always treated abstractly, whether they originated from universal or existential type variables. We do not distinguish them syntactically in Γ to avoid heavy notations. However, universal variables are those appearing in sequences $\langle\overline{\alpha}\rangle$ that immediately precede a functor parameter Y in Γ. An intuition is that existentials are meant to be anchored while universal *cannot* be anchored. Notice that $\mathrm{args}(\Delta)$ is the sequence of *universally* quantified variables, hence, ignoring existentially quantified variables. Rule A-Sig-FctApp is particular, as it contains two premises that treat $\overline{\alpha}$ differently, that is, universally in the functor's body $\mathcal{C}$, but existentially while typing the signature $\mathcal{C}_a$ where they need to be anchored.

3.1 Anchoring of environments

$\boxed{\Gamma \hookrightarrow \theta_\Gamma}$

$$\textsc{A-Env-Decl}\quad \frac{\Gamma\,;\,\theta_\Gamma \vdash \mathcal{D} \overset{A}{\hookrightarrow} \mathsf{D} : \theta}{\Gamma, A.\mathcal{D} \hookrightarrow \theta_\Gamma \uplus \theta} \qquad \textsc{A-Env-Arg}\quad \frac{\Gamma \cdot \overline{\alpha}\,;\,\theta_\Gamma \vdash \mathcal{C}_a \overset{Y}{\hookrightarrow} \mathsf{S}_a : \theta_a \qquad \mathrm{dom}(\theta) = \overline{\alpha}}{\Gamma, \overline{\alpha}, (Y : \mathcal{C}) \hookrightarrow \theta_\Gamma \uplus \theta_a} \qquad \textsc{A-Env-Abs}\quad \frac{\Gamma \hookrightarrow \theta_\Gamma}{\Gamma, \overline{\alpha} \hookrightarrow \theta_\Gamma}$$

3.2 Signature anchoring

$\boxed{\Gamma\,;\,\theta_\Gamma \vdash \mathcal{C} \overset{P}{\hookrightarrow} \mathsf{S} : \theta}$

$$\textsc{A-Sig-StrPath}\quad \frac{\Gamma\,;\,\theta_\Gamma \vdash \overline{\mathcal{D}} \overset{A}{\hookrightarrow} \overline{\mathsf{D}} : \theta \qquad A \notin \Gamma}{\Gamma\,;\,\theta_\Gamma \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \overset{P}{\hookrightarrow} \mathtt{sig}_A\ \overline{\mathsf{D}}\ \mathsf{end} : \theta[A \mapsto P]}$$

$$\textsc{A-Sig-StrNone}\quad \frac{\Gamma\,;\,\theta_\Gamma \vdash \overline{\mathcal{D}} \overset{A}{\hookrightarrow} \overline{\mathsf{D}} : \theta \qquad A \notin \Gamma \qquad \mathrm{dom}(\theta) = \overline{\alpha}}{\Gamma\,;\,\theta_\Gamma \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \hookrightarrow \mathtt{sig}_A\ \overline{\mathsf{D}}\ \mathsf{end} : (\overline{\alpha} \mapsto \_)}$$

$$\textsc{A-Sig-FctGen}\quad \frac{\Gamma \cdot \overline{\alpha}\,;\,\theta_\Gamma \vdash \mathcal{C} \hookrightarrow \mathsf{S} : (\overline{\alpha} \mapsto \_)}{\Gamma\,;\,\theta_\Gamma \vdash () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \xhookrightarrow{A.Val} () \to \mathsf{S} : \varnothing}$$

$$\textsc{A-Sig-FctApp}\quad \frac{\Gamma \cdot \overline{\alpha}\,;\,\theta_\Gamma \vdash \mathcal{C}_a \overset{Y}{\hookrightarrow} \mathsf{S}_a : \theta_a \qquad \mathrm{dom}(\theta_a) = \overline{\alpha} \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a\,;\,\theta_\Gamma \uplus \theta_a \vdash \mathcal{C} \xhookrightarrow{A.Val(Y)} \mathsf{S} : \theta}{\Gamma\,;\,\theta_\Gamma \vdash \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \xhookrightarrow{A.Val} (Y : \mathsf{S}_a) \to \mathsf{S} : \lambda Y.\theta}$$

3.3 Declaration anchoring

$\boxed{\Gamma\,;\,\theta_\Gamma \vdash \mathcal{D} \overset{A}{\hookrightarrow} \mathtt{D} : \theta}$

A-Decl-Val
$$\frac{}{\Gamma\,;\,\theta_\Gamma \vdash \mathtt{val}\ x : \tau \overset{A}{\hookrightarrow} (\mathtt{val}\ x : \mathsf{u}) : \varnothing}$$

A-Decl-Anchor
$$\frac{\varphi \notin \mathrm{dom}(\theta_\Gamma) \qquad \varphi \in \Delta \qquad \mathrm{args}(\Delta) = \overline{\alpha_1}\,;\,\ldots\,\overline{\alpha_n}}{\Gamma \cdot \Delta\,;\,\theta_\Gamma \vdash \mathtt{type}\ t = \varphi\langle\overline{\alpha_1}\rangle \ldots \langle\overline{\alpha_n}\rangle \overset{A}{\hookrightarrow} \mathtt{type}\ t = A.t : (\varphi \mapsto A.t)}$$

A-Decl-Type
$$\frac{\Gamma\,;\,\theta_\Gamma \vdash \tau \hookrightarrow \mathsf{u}}{\Gamma\,;\,\theta_\Gamma \vdash \mathtt{type}\ t = \tau \overset{A}{\hookrightarrow} \mathtt{type}\ t = \mathsf{u} : \varnothing}$$

A-Decl-Mod
$$\frac{\Gamma\,;\,\theta_\Gamma \vdash \mathcal{C} \overset{A.X}{\longrightarrow} \mathtt{S} : \theta}{\Gamma\,;\,\theta_\Gamma \vdash \mathtt{module}\ X : \mathcal{C} \overset{A}{\hookrightarrow} \mathtt{module}\ X : \mathtt{S} : A.\theta}$$

A-Decl-Empty
$$\Gamma\,;\,\theta_\Gamma \vdash \varnothing \overset{A}{\hookrightarrow} \varnothing : \varnothing$$

A-Decl-ModType
$$\frac{\Gamma, \overline{\alpha}\,;\,\theta_\Gamma \vdash \mathcal{C} \hookrightarrow \mathtt{S} : (\overline{\alpha} \mapsto \_)}{\Gamma\,;\,\theta_\Gamma \vdash (\mathtt{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}) \overset{A}{\hookrightarrow} \mathtt{module\ type}\ T = \mathtt{S} : \varnothing}$$

A-Decl-Seq
$$\frac{\Gamma\,;\,\theta_\Gamma \vdash \mathcal{D} \overset{A}{\hookrightarrow} \mathtt{D} : \theta_1 \qquad \Gamma, A.\mathcal{D}\,;\,\theta \uplus \theta_1 \vdash \overline{\mathcal{D}} \overset{A}{\hookrightarrow} \overline{\mathtt{D}} : \theta_2}{\Gamma\,;\,\theta_\Gamma \vdash \mathcal{D}, \overline{\mathcal{D}} \overset{A}{\hookrightarrow} \mathtt{D}, \overline{\mathtt{D}} : \theta_1 \uplus \theta_2}$$

3.4 Anchoring of abstract types

$\boxed{\Gamma\,;\,\theta_\Gamma \vdash \tau \hookrightarrow \mathsf{u}}$

A-Type-Application
$$\frac{\begin{array}{c}\tau = \varphi\langle\tau_1 \ldots\rangle \ldots \langle\tau_n \ldots\rangle \qquad \theta_\Gamma(\varphi) = \lambda Y_k. \ldots \lambda Y_n.\ P.t \\ \forall i \in [\![k, n]\!]\,.\ \Gamma\,;\,\theta_\Gamma \vdash \tau_i \hookrightarrow P_i.id \qquad \mathsf{u} = \theta_\Gamma(\varphi)(P_k) \ldots (P_n) \qquad \Gamma \vdash \mathsf{u} : \tau\end{array}}{\Gamma\,;\,\theta_\Gamma \vdash \tau \hookrightarrow \mathsf{u}}$$

3.5 Properties of anchoring

3.5.1 Proof of [1, Theorem 4.2]. We show by induction, when type equality in M^ω is taken up to $\beta\eta$-equivalence:

$$\Gamma \cdot \Delta\,;\,\theta_\Gamma \vdash \mathcal{C} \hookrightarrow \mathtt{S} : \theta \ \wedge\ \mathrm{dom}(\theta) = \overline{\alpha} \ \wedge\ \Gamma \hookrightarrow \theta_\Gamma \implies \Gamma \cdot \Delta \vdash \mathtt{S} : \lambda\overline{\beta}.\mathcal{C}' \ \wedge\ \mathcal{C}'[\overline{\beta} \mapsto \overline{\alpha}(\mathrm{args}(\Delta))] = \mathcal{C}$$

$$\Gamma \cdot \Delta\,;\,\theta_\Gamma \vdash \overline{\mathcal{D}} \overset{A}{\hookrightarrow} \overline{\mathtt{D}} : \theta \ \wedge\ \mathrm{dom}(\theta) = \overline{\alpha} \ \wedge\ \Gamma \hookrightarrow \theta_\Gamma \implies \Gamma \cdot \Delta \vdash \overline{\mathtt{D}} : \lambda\overline{\beta}.\overline{\mathcal{D}}' \ \wedge\ \overline{\mathcal{D}}'[\overline{\beta} \mapsto \overline{\alpha}(\mathrm{args}(\Delta))] = \overline{\mathcal{D}}$$

The proof is by structural induction on the anchoring derivation.

- A-Sig-FctGen: The conclusion of the rule is the signature anchoring judgment
$$\Gamma \cdot \Delta\,;\,\theta_\Gamma \vdash () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \overset{A.Val}{\longrightarrow} () \to \mathtt{S} : \varnothing$$
Since the domain of the local map is empty, we just have to show $\Gamma \cdot \Delta \vdash () \to \mathtt{S} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}$ (**1**). The premise of the rule is $\Gamma \cdot \Delta \cdot \overline{\alpha}\,;\,\theta_\Gamma \vdash \mathcal{C} \hookrightarrow \mathtt{S} : (\overline{\alpha} \mapsto \_)$. By induction hypothesis, we have $\Gamma \cdot \Delta \vdash \mathtt{S} : \lambda\overline{\beta}.\mathcal{C}'$ (**2**) and $\mathcal{C}'[\overline{\beta} \mapsto \overline{\alpha}] = \mathcal{C}$, since $\mathrm{args}(\overline{\alpha})$ is empty, that is $\lambda\overline{\beta}.\mathcal{C}' = \lambda\overline{\alpha}.\mathcal{C}$. Then (1) follows by M-Sig-GenFct applied to (2).
- A-Sig-FctApp: The rule is
$$\frac{\Gamma \cdot \Delta \cdot \overline{\alpha}\,;\,\theta_\Gamma \vdash \mathcal{C}_a \overset{Y}{\hookrightarrow} \mathtt{S}_a : \theta_a\ (\mathbf{1}) \qquad \Gamma \cdot \Delta, \overline{\alpha}, Y : \mathcal{C}_a\,;\,\theta_\Gamma \uplus \theta_a \vdash \mathcal{C} \overset{A.Val(Y)}{\longrightarrow} \mathtt{S} : \theta\ (\mathbf{2})}{\Gamma \cdot \Delta\,;\,\theta_\Gamma \vdash \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \overset{A.Val}{\longrightarrow} (Y : \mathtt{S}_a) \to \mathtt{S} : \lambda Y.\theta}$$
with $\mathrm{dom}(\theta_a) = \overline{\alpha}$. By IH applied to (1), we have $\Gamma \cdot \Delta \vdash \mathtt{S}_a : \lambda\overline{\alpha}.\mathcal{C}_a$ (**3**) since $\mathrm{args}(\overline{\alpha}, \overline{\alpha})$ is empty. Let $\overline{\gamma}$ be $\mathrm{dom}(\theta)$. By IH applied to (2), we have $\Gamma \cdot \Delta, \overline{\alpha}, Y : \mathcal{C}_a \vdash \mathtt{S} : \lambda\overline{\beta}.\mathcal{C}'$ (**4**) with

$\mathcal{C}'[\overline{\beta} \mapsto \overline{\gamma}(\text{args}(\Delta), \overline{\alpha})] = \mathcal{C}$ (**5**), since $\text{args}(\Delta, \overline{\alpha})$ is equal to $\text{args}(\Delta), \overline{\alpha}$. By rule M-Typ-Sig-AppFct applied to (3) and (4), we have:

$$\Gamma \cdot \Delta \vdash (Y : \mathtt{S}_a) \to \mathtt{S} : \lambda\overline{\beta'}.\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}'[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}]$$

It remains to check that $\left(\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}'[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}]\right)[\overline{\beta'} \mapsto \overline{\gamma}(\text{args}(\Delta))]$ is equal to $\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}$, which follows by composition of substitutions from (5), since β' does not occur free in $\mathcal{C}_a$.

- A-Sig-StrPath and A-Sig-StrNone are immediate by induction.
- For A-Decl-Val, A-Decl-Type, we may easily show that $\Gamma\,;\theta_\Gamma \vdash \tau \hookrightarrow \mathtt{u}$ implies $\Gamma \vdash \mathtt{u} : \tau$.
  In both cases, the conclusion is of the form $\Gamma;\theta_\Gamma \vdash \mathtt{D} \xhookrightarrow{A} \mathcal{D} : \varnothing$, it suffices to show $\Gamma \cdot \Delta \vdash \mathtt{D} : \mathcal{D}$ (**1**).
  In the case of A-Decl-Val, the conclusion is $\Gamma\,;\theta_\Gamma \vdash \mathtt{val}\ x : \tau \xhookrightarrow{A} (\mathtt{val}\ x : \mathtt{u}) : \varnothing$ and the premise is $\Gamma \vdash \mathtt{u} : \tau$, which implies $\Gamma \vdash \mathtt{u} : \tau$. Then, (1) follows by M-Typ-Decl-Val.
  The case of A-Decl-Type is similar.
- A-Decl-Mod, A-Decl-ModType are immediate by induction.
- A-Decl-Anchor: the conclusion of the rule is:

$$\Gamma \cdot \Delta\,;\theta_\Gamma \vdash \mathtt{type}\ t = \varphi\langle\overline{\alpha_1}\rangle \ldots \langle\overline{\alpha_n}\rangle \xhookrightarrow{A} \mathtt{type}\ t = A.t : (\varphi \mapsto A.t)$$

  By Rule M-Typ-Decl-TypeAbs, we have $\Gamma \cdot \Delta \vdash (\mathtt{type}\ t = A.t) : \lambda\alpha.\text{type } t = \alpha$. From the premises, we know that $\varphi \in \Delta$ and $\text{args}(\Delta) = \overline{\alpha_1}\,; \ldots\ \overline{\alpha_n}$. Hence, $(\text{type } t = \alpha)[\varphi \mapsto \varphi\,\overline{\alpha_1}\,; \ldots\ \overline{\alpha_n}]$ is equal to $\text{type } t = \varphi\langle\overline{\alpha_1}\rangle \ldots \langle\overline{\alpha_n}\rangle$, which is exactly the declaration we started with.
- A-Decl-Seq: Writing the domain of $\theta_1 \uplus \theta_2$ as $\text{dom}(\theta_1 \uplus \theta_2) = (\overline{\alpha}_1, \overline{\alpha}_2)$, we have by induction hypothesis:

$$\begin{aligned} \Gamma \cdot \Delta \vdash \mathtt{D}_1 : \lambda\overline{\beta}_1.\mathcal{D}'_1 \quad &\wedge \quad \mathcal{D}'_1[\overline{\beta}_1 \mapsto \overline{\alpha}_1(\text{args}(\Delta))] = \mathcal{D}_1 \\ \Gamma \cdot \Delta, A.\mathcal{D} \vdash \overline{\mathtt{D}_2} : \lambda\overline{\beta}_2.\overline{\mathcal{D}'_2} \quad &\wedge \quad \overline{\mathcal{D}'_2}[\overline{\beta}_2 \mapsto \overline{\alpha}_2(\text{args}(\Delta, A.\mathcal{D}))] = \overline{\mathcal{D}_2} \end{aligned}$$

  First, we introduce $\overline{\mathcal{D}''_2} = \overline{\mathcal{D}'_2}[\overline{\alpha}_1(\text{args}(\Delta)) \mapsto \overline{\beta}_1]$. Then, we show that

$$\Gamma \cdot \Delta \vdash \mathtt{D}_1, \overline{\mathtt{D}_2} : \lambda\overline{\beta}_1\overline{\beta}_2.\mathcal{D}'_1, \overline{\mathcal{D}''_2}$$

  The first premise is immediate by induction hypothesis. For the second one, we show by induction that :

$$\Gamma \cdot \Delta, A.\mathcal{D} \vdash \overline{\mathtt{D}_2} : \lambda\overline{\beta}_2.\overline{\mathcal{D}'_2} \implies \Gamma \cdot \Delta, \overline{\beta}_1, A.\mathcal{D}' \vdash \overline{\mathtt{D}_2} : \lambda\overline{\beta}_2.\overline{\mathcal{D}''_2}$$

  Secondly, we show that the substitution is valid. By definition, we have

$$\text{args}(\Delta, A.\mathcal{D}) = \text{args}(\Delta)$$

  As the declaration $\mathcal{D}'_1$ is well-formed in an environment that does not contain the $\overline{\beta}_2$, substituting for them does not change anything. Therefore, we have:

$$\begin{aligned} &\mathcal{D}'_1, \overline{\mathcal{D}''_2}[\overline{\beta}_1, \overline{\beta}_2 \mapsto \overline{\alpha}_1(\text{args}(\Delta)), \overline{\alpha}_2(\text{args}(\Delta))] \\ &= \mathcal{D}'_1[\overline{\beta}_1 \mapsto \overline{\alpha}_1(\text{args}(\Delta))]\,, (\overline{\mathcal{D}''_2}[\overline{\beta}_1 \mapsto \overline{\alpha}_1(\text{args}(\Delta))])[\overline{\beta}_2 \mapsto \overline{\alpha}_2(\text{args}(\Delta))] \\ &= \mathcal{D}_1, \overline{\mathcal{D}'_2}[\overline{\beta}_2 \mapsto \overline{\alpha}_2(\text{args}(\Delta))] \\ &= \mathcal{D}_1, \overline{\mathcal{D}'_2}[\overline{\beta}_2 \mapsto \overline{\alpha}_2(\text{args}(\Delta, A.\mathcal{D}))] \\ &= \mathcal{D}_1, \overline{\mathcal{D}_2} \end{aligned}$$

3.5.2 Proof of [1, Theorem 4.1]. We first prove that typed signatures are anchorable, then that the result is equivalent to the signature we started with. We start with the following lemma:

Lemma 3.1 (Skolemization lemma). *Skolemizing a signature does not change its local anchoring:*

$$\Gamma, \overline{\alpha} \cdot \overline{\beta}, \Delta \,;\, \theta_\Gamma \vdash \mathcal{C} \overset{P}{\hookrightarrow} \mathsf{S} : \theta \implies \Gamma \cdot \overline{\beta'}, \overline{\alpha}, \Delta \,;\, \theta_\Gamma \vdash \mathcal{C}[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}] \overset{P}{\hookrightarrow} \mathsf{S} : \theta$$

Proof. The key observation is that, from the source syntax, accessing *local* types does take into account the depth of enclosing applicative functors. The two key rules are:

- A-Decl-Anchor: when anchoring a type, the number of (list of) universally quantified parameters n does not appear in the anchoring (nor in the resulting declaration).
- A-Type-Application: similarly, the first type arguments τ_1 to τ_k are ignored. If the anchoring point $\theta_\Gamma(\varphi)$ is parameterized by only $n - k$ arguments, adding another argument in front does not change the anchoring.

□

For the first half, we show only that typed signatures are anchorable:

$$\Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C} \wedge \Gamma \hookrightarrow \theta_\Gamma \implies \exists \mathsf{S}'.\ \Gamma \cdot \overline{\alpha} \,;\, \theta_\Gamma \vdash \mathcal{C} \hookrightarrow \mathsf{S}' : \overline{\alpha} \mapsto \_$$

We proceed by induction on the typing derivation of signatures and declarations:

- M-Typ-Sig-ModType and M-Typ-Sig-ModType are base-cases of the induction. We easily show that the anchoring of the environment $\Gamma \hookrightarrow \theta_\Gamma$ implies that stored module types are anchorable.
- M-Typ-Sig-GenFct and M-Typ-Sig-Str are immediate.
- M-Typ-Sig-AppFct: the induction hypothesis gives us that the signature of the core of the functor is anchorable before the substitution of $\overline{\beta}$ for $\overline{\beta'}(\overline{\alpha})$. Here, we use the skolemization lemma.
- M-Typ-Sig-Con: we first note that all type variables $\overline{\alpha}$ appear in $\mathcal{C}'$. Then, as subtyping between types is only defined by equality, all the types expressions $\overline{\tau}$ appear in $\mathcal{C}$. As $\mathcal{C}$ is anchorable, so are all its type components, making $\mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]$ anchorable.

The second part is an immediate consequence of Theorem 4.2.

4 SYSTEM F$^\omega$

In this section we give the definition of our version of F$^\omega$ extended with record types and predicative kind polymorphism.

4.1 Environment and kind checking

$\boxed{\vdash \Gamma}$ and $\boxed{\Gamma \vdash \star}$

$$\vdash \cdot \qquad \frac{\vdash \Gamma \qquad \varkappa \notin \Gamma}{\vdash \Gamma, \varkappa} \qquad \frac{\Gamma \vdash \kappa \qquad \alpha \notin \Gamma}{\vdash \Gamma, \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : \star \qquad x \notin \Gamma}{\vdash \Gamma, x : \tau}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \star} \qquad \frac{\vdash \Gamma \qquad \varkappa \in \Gamma}{\Gamma \vdash \varkappa} \qquad \frac{\Gamma \vdash \kappa \qquad \Gamma \vdash \kappa'}{\Gamma \vdash \kappa \to \kappa'} \qquad \frac{\Gamma, \varkappa \vdash \kappa}{\Gamma \vdash \forall \varkappa.\kappa}$$

4.2 Type checking

$\boxed{\Gamma \vdash \mathsf{u} : \kappa}$

$$\frac{\Gamma \vdash \tau_1 : \star \qquad \Gamma \vdash \tau_2 : \star}{\Gamma \vdash \tau_1 \to \tau_2 : \star} \qquad \frac{\overline{\Gamma \vdash \tau : \star} \qquad \vdash \Gamma}{\Gamma \vdash \{\overline{\ell_l : \tau}\} : \star} \qquad \frac{\vdash \Gamma \qquad \alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma, \alpha : \kappa \vdash \tau : \star}{\Gamma \vdash \forall(\alpha : \kappa).\tau : \star}$$

$$\frac{\Gamma, \varkappa \vdash \tau : \star}{\Gamma \vdash \forall \varkappa.\tau : \star} \qquad \frac{\Gamma, \alpha : \kappa \vdash \tau : \star}{\Gamma \vdash \exists^{\blacktriangledown}(\alpha : \kappa).\tau : \star} \qquad \frac{\Gamma, \alpha : \kappa \vdash \tau : \kappa'}{\Gamma \vdash \Lambda(\alpha : \kappa).\tau : \kappa \to \kappa'}$$

$$\frac{\Gamma \vdash \tau_1 : \kappa' \to \kappa \qquad \Gamma \vdash \tau_2 : \kappa'}{\Gamma \vdash \tau_1\, \tau_2 : \kappa} \qquad \frac{\Gamma, \varkappa \vdash \tau : \kappa}{\Gamma \vdash \Lambda \varkappa.\tau : \forall \varkappa.\kappa} \qquad \frac{\Gamma \vdash \tau : \forall \varkappa.\kappa \qquad \Gamma \vdash \varsigma}{\Gamma \vdash \tau\, \varsigma : \kappa[\varkappa \mapsto \varsigma]}$$

4.3 Term typing

$\boxed{\Gamma \vdash e : \mathsf{u}}$

F-Var
$$\frac{\vdash \Gamma \qquad x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

F-Abs
$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda(x : \tau).e : \tau \to \tau'}$$

F-App
$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\, e_2 : \tau}$$

F-Record
$$\frac{\overline{\Gamma \vdash e : \tau} \qquad \#(\overline{\ell})}{\Gamma \vdash \{\overline{\ell = e}\} : \{\overline{\ell : \tau}\}}$$

F-Proj
$$\frac{\Gamma \vdash e : \{\ell : \tau, \overline{\ell_1 : \tau_1}\}}{\Gamma \vdash e.\ell : \tau}$$

F-Append
$$\frac{\Gamma \vdash e_1 : \{\overline{\ell_1 : \tau_1}\} \qquad \Gamma \vdash e_2 : \{\overline{\ell_2 : \tau_2}\} \qquad \overline{\ell_1} \# \overline{\ell_2}}{\Gamma \vdash e_1 \mathbin{@} e_2 : \{\overline{\ell_1 : \tau_1}.\overline{\ell_2 : \tau_2}\}}$$

F-Tapp
$$\frac{\Gamma \vdash e : \forall(\alpha : \kappa).\sigma \qquad \Gamma \vdash \tau : \kappa}{\Gamma \vdash e\, \tau : \sigma[\tau \mapsto \alpha]}$$

F-Kapp
$$\frac{\Gamma \vdash e : \forall \varkappa.\tau \qquad \Gamma \vdash \varsigma}{\Gamma \vdash e\, \varsigma : \tau[\varkappa \mapsto \varsigma]}$$

F-Tabs
$$\frac{\Gamma, \alpha : \kappa \vdash e : \tau}{\Gamma \vdash \lambda(\alpha : \kappa).e : \forall(\alpha : \kappa).\tau}$$

F-Kabs
$$\frac{\Gamma, \varkappa \vdash e : \tau}{\Gamma \vdash \Lambda \varkappa.e : \forall \varkappa.\tau}$$

F-Pack
$$\frac{\Gamma \vdash \exists^{\blacktriangledown}(\alpha : \kappa).\sigma : \star \qquad \Gamma \vdash \tau : \kappa \qquad \Gamma \vdash e : \sigma[\tau \mapsto \alpha]}{\Gamma \vdash \mathsf{pack}\, \langle \tau, e \rangle \text{ as } \exists^{\blacktriangledown}(\alpha : \kappa).\sigma : \exists^{\blacktriangledown}(\alpha : \kappa).\sigma}$$

F-Unpack
$$\frac{\Gamma \vdash e_1 : \exists^{\blacktriangledown}(\alpha : \kappa).\tau \qquad \Gamma, \alpha : \kappa, x : \tau \vdash e_2 : \sigma \qquad \Gamma \vdash \sigma : \star}{\Gamma \vdash \mathsf{unpack}\, \langle \alpha, x \rangle = e_1 \text{ in } e_2 : \sigma}$$

4.4 Encoding of transparent existentials

For reference, we remind the encoding of transparent existential types in Figure 2.

4.5 lift* operator

The operator lift^* is defined as $\mathsf{lift}_q^{\forall^p \triangleright}$ where p and q represent the size of $\bar{\alpha}$ and $\bar{\beta}$[1], where $\mathsf{lift}_q^{\forall^p \triangleright}$ is itself inductively defined as follows:

$$\mathsf{lift}_{q+1}^{\forall^p \triangleright} e \triangleq \mathsf{repack}^{\triangledown}\, \langle \alpha, x \rangle = \mathsf{lift}^{\forall^p}(\mathsf{lift}^{\triangleright} e) \text{ in } \mathsf{lift}_q^{\forall^p \triangleright} x \qquad \mathsf{lift}^{\forall^{p+1}} e \triangleq \mathsf{lift}^{\forall}(\Lambda\alpha.\mathsf{lift}^{\forall^p}(e\,\alpha))$$

$$\mathsf{lift}_0^{\forall^p \triangleright} e \triangleq e \qquad \mathsf{lift}^{\forall^0} e \triangleq e$$

[1] p and q are left implicit in $\mathsf{lift}^* e$ as they can be determined from the type of the argument e

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

Fig. 2. Implementation of transparent existentials as a library in F^{ω} with (predicative) kind polymorphism

4.6 Derived typing rules for transparent existentials

$\boxed{\Gamma \vdash e : \mathsf{u}}$

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

F-Seal
$$\frac{\Gamma \vdash e : \exists^{\nabla\tau}(\alpha:\kappa).\,\sigma}{\Gamma \vdash \mathsf{seal}\ e : \exists^{\blacktriangledown}(\alpha:\kappa).\,\tau'}$$

F-Hidden
$$\frac{\Gamma \vdash e_1 : \exists^{\nabla\tau}(\alpha:\kappa).\,\sigma \qquad \Gamma, \alpha:\kappa, x:\tau \vdash e_2 : \sigma'}{\Gamma \vdash \mathsf{repack}^{\nabla}\,\langle\alpha,x\rangle = e_1\ \mathsf{in}\ e_2 : \exists^{\blacktriangledown}(\alpha:\kappa).\,\sigma'}$$

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

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

5 ELABORATION RULES

5.1 Subtyping

5.1.1 *Signature subtyping*

$\boxed{\Gamma \vdash \mathcal{C} \prec \mathcal{C}' \rightsquigarrow f}$

E-Sub-Sig-Struct
$$\frac{\overline{\mathcal{D}_0} \subseteq \overline{\mathcal{D}} \qquad \Gamma \vdash \overline{\mathcal{D}_0} \prec \overline{\mathcal{D}'} \rightsquigarrow \overline{f} \qquad \overline{I'} = \mathsf{dom}(\overline{\mathcal{D}'})}{\Gamma \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \prec \mathsf{sig}\ \overline{\mathcal{D}'}\ \mathsf{end} \rightsquigarrow \lambda x.\ \{\overline{\ell_{I'} = f\ (x.\ell_{I'})}\}}$$

E-Sub-Sig-GenFct
$$\frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} \prec \mathcal{C}'[\overline{\alpha}' \mapsto \overline{\tau}] \rightsquigarrow f}{\Gamma \vdash () \to \exists^{\triangledown}\overline{\alpha}.\mathcal{C} \prec () \to \exists^{\triangledown}\overline{\alpha}'.\mathcal{C}' \rightsquigarrow \lambda x.\lambda\_.\,\mathsf{unpack}\ \langle\overline{\alpha}, y\rangle = x\ ()\ \mathsf{in}\ \mathsf{pack}\ \langle\overline{\tau}, f\ y\rangle\ \mathsf{as}\ \exists^{\triangledown}\overline{\alpha}'.\mathcal{C}'}$$

E-Sub-Sig-AppFct
$$\frac{\Gamma, \overline{\alpha}' \vdash \mathcal{C}'_a \prec \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow f \qquad \Gamma, \overline{\alpha}' \vdash \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] \prec \mathcal{C}' \rightsquigarrow g}{\Gamma \vdash \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \prec \forall\overline{\alpha}'.\mathcal{C}'_a \to \mathcal{C}' \rightsquigarrow \lambda x : \_.\Lambda\overline{\alpha}'.\lambda y : \mathcal{C}'_a.g\ (x\ \overline{\tau}\ (f\ y))}$$

5.1.2 *Declaration subtyping* $\boxed{\Gamma \vdash \mathcal{D} \prec \mathcal{D}' \rightsquigarrow f}$

E-Sub-Decl-Val
$$\Gamma \vdash (\mathsf{val}\ x : \tau) \prec (\mathsf{val}\ x : \tau) \rightsquigarrow \lambda x.x$$

E-Sub-Decl-Type
$$\Gamma \vdash (\mathsf{type}\ t = \tau) \prec (\mathsf{type}\ t = \tau) \rightsquigarrow \lambda x.x$$

E-Sub-Decl-Mod
$$\frac{\Gamma \vdash \mathcal{C} \prec \mathcal{C}' \rightsquigarrow f}{\Gamma \vdash (\mathsf{module}\ X : \mathcal{C}) \prec (\mathsf{module}\ X : \mathcal{C}') \rightsquigarrow f}$$

E-Sub-Decl-ModType
$$\frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} \prec \mathcal{C}' \rightsquigarrow f \qquad \Gamma, \overline{\alpha} \vdash \mathcal{C}' \prec \mathcal{C} \rightsquigarrow g}{\Gamma \vdash (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}) \prec (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}') \rightsquigarrow \lambda x.x}$$

5.2 Typing

5.2.1 *Module typing* $\boxed{\Gamma \vdash \mathtt{M} : \exists^{\triangledown}\bar{\vartheta}.\mathcal{C} \rightsquigarrow e}$

E-Typ-Mod-Arg
$$\frac{(Y : \mathcal{C}) \in \Gamma}{\Gamma \vdash Y : \mathcal{C} \rightsquigarrow Y}$$

E-Typ-Mod-Var
$$\frac{(A.X :\ \mathsf{module}\ \mathcal{C}) \in \Gamma}{\Gamma \vdash A.X : \mathcal{C} \rightsquigarrow A_X}$$

E-Typ-Mod-AppFct
$$\frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a \vdash \mathtt{M} : \exists^{\triangledown\overline{\tau}}(\overline{\beta}).\mathcal{C} \rightsquigarrow e}{\Gamma \vdash (Y : \mathtt{S}) \to \mathtt{M} : \exists^{\triangledown\overline{\lambda\overline{\alpha}.\tau}}(\overline{\beta'}).\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right] \rightsquigarrow \mathsf{lift}^*(\Lambda\overline{\alpha}.\lambda(Y : \mathcal{C}_a).e)}$$

E-Typ-Mod-AppApp
$$\frac{\Gamma \vdash P : \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \rightsquigarrow e \qquad \Gamma \vdash P' : \mathcal{C}' \rightsquigarrow e' \qquad \Gamma \vdash \mathcal{C}' \prec \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow f}{\Gamma \vdash P(P') : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow e\ \overline{\tau}\ (f\ e')}$$

E-Typ-Mod-GenFct
$$\frac{\Gamma \vdash \mathtt{M} : \exists^{\triangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash () \to \mathtt{M} : () \to \exists^{\triangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow \lambda(\_ : ()).e}$$

E-Typ-Mod-GenApp
$$\frac{\Gamma \vdash P : () \to \exists^{\triangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash P() : \exists^{\triangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow e\ ()}$$

E-Typ-Mod-Ascr
$$\frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C} \qquad \Gamma \vdash P : \mathcal{C}' \rightsquigarrow e \qquad \Gamma \vdash \mathcal{C}' \prec \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow f \qquad \Gamma \vdash \overline{\tau} : \overline{\zeta}}{\Gamma \vdash (P : \mathtt{S}) : \exists^{\triangledown\overline{\tau}}(\overline{\alpha}).\mathcal{C} \rightsquigarrow \mathsf{pack}\ f\ e\ \mathsf{as}\ \exists^{\triangledown\overline{\tau}}(\overline{\alpha}).\mathcal{C}}$$

E-Typ-Mod-Seal
$$\frac{\Gamma \vdash \mathsf{M} : \exists^{\nabla\overline{\tau}}(\overline{\alpha}).\mathcal{C} \rightsquigarrow e}{\Gamma \vdash \mathsf{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow \mathsf{seal}^{|\overline{\alpha}|}\ e}$$

E-Typ-Mod-Struct
$$\frac{\Gamma \vdash_A \overline{\mathsf{B}} : \exists^{\bar{\vartheta}}\bar{\alpha}.\overline{\mathcal{D}} \rightsquigarrow e \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{struct}_A\ \overline{\mathsf{B}}\ \mathsf{end} : \exists^{\bar{\vartheta}}\bar{\alpha}.\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \rightsquigarrow e}$$

E-Typ-Mod-Proj
$$\frac{\Gamma \vdash \mathsf{M} : \exists^{\bar{\vartheta}}\bar{\alpha}.\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \rightsquigarrow e \qquad \mathsf{module}\ X : \mathcal{C} \in \overline{\mathcal{D}} \qquad \overline{\alpha}' = \mathsf{fv}(\mathcal{C}) \cap \overline{\alpha}}{\Gamma \vdash \mathsf{M}.X : \exists^{\bar{\vartheta}}\bar{\alpha}.\mathcal{C} \rightsquigarrow \mathsf{clean}^{\diamond}\langle\overline{\alpha}, \overline{\alpha}'\rangle\ (\mathsf{repack}^{\diamond}\ \langle\overline{\alpha}, x\rangle = e\ \mathsf{in}\ x.\ell_X)}$$

Rule E-Typ-Mod-Proj, which performs garbage collection, uses a helper function $\mathsf{clean}^{\diamond}\langle\overline{\alpha}, \overline{\beta}\rangle\ e$ defined as::

$$\begin{aligned}
\mathsf{clean}^{\diamond}\langle\gamma\overline{\alpha}, \gamma\overline{\beta}\rangle\ e &\triangleq \mathsf{repack}^{\diamond}\ \langle\alpha, x\rangle = e\ \mathsf{in}\ \mathsf{clean}^{\diamond}\langle\overline{\alpha}, \overline{\beta}\rangle\ x \\
\mathsf{clean}^{\diamond}\langle\gamma\overline{\alpha}, \gamma'\overline{\beta}\rangle\ e &\triangleq \mathsf{unpack}^{\diamond}\ \langle\alpha, x\rangle = e\ \mathsf{in}\ \mathsf{clean}^{\diamond}\langle\overline{\alpha}, \gamma\overline{\beta}\rangle\ x \\
\mathsf{clean}^{\diamond}\langle\varnothing, \varnothing\rangle\ e &\triangleq e
\end{aligned}$$

5.2.2 *Binding typing*

$\boxed{\Gamma \vdash_A \mathsf{B} : \mathcal{D} \rightsquigarrow e}$

E-Typ-Bind-Let
$$\frac{\Gamma \vdash \mathsf{e} : \tau \rightsquigarrow e}{\Gamma \vdash_A (\mathtt{let}\ x = \mathsf{e}) : (\mathsf{val}\ x : \tau) \rightsquigarrow \{\ell_x = e\}}$$

E-Typ-Bind-Type
$$\frac{\Gamma \vdash \mathsf{u} : \tau}{\Gamma \vdash_A (\mathsf{type}\ t = \mathsf{u}) : (\mathsf{type}\ t = \tau) \rightsquigarrow \{\ell_t = \langle\langle\tau\rangle\rangle\}}$$

E-Typ-Bind-ModType
$$\frac{\Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module\ type}\ T = \mathsf{S}) : (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}) \rightsquigarrow \{\ell_T = \langle\langle\lambda\overline{\alpha}.\mathcal{C}\rangle\rangle\}}$$

E-Typ-Bind-Empty
$$\Gamma \vdash_A \varnothing : \varnothing \rightsquigarrow \{\}$$

E-Typ-Bind-Mod
$$\frac{\Gamma \vdash_A \mathsf{M} : \exists^{\bar{\vartheta}}\bar{\alpha}.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash_A (\mathtt{module}\ X = \mathsf{M}) : (\exists^{\bar{\vartheta}}\bar{\alpha}.\mathsf{module}\ X : \mathcal{C}) \rightsquigarrow \mathsf{repack}^{\diamond}\ \langle\overline{\alpha}, x\rangle = e\ \mathsf{in}\ \{\ell_X = x\}}$$

E-Typ-Bind-Seq
$$\frac{\Gamma \vdash_A \mathsf{B} : \exists^{\bar{\vartheta}_1}\overline{\alpha}_1.\mathcal{D} \rightsquigarrow e_1 \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D} \vdash_A \overline{\mathsf{B}} : \exists^{\blacktriangledown}\bar{\vartheta}_2.\overline{\mathcal{D}} \rightsquigarrow e_2}{\Gamma \vdash_A \mathsf{B}, \overline{\mathsf{B}} : \exists^{\bar{\vartheta}_1\bar{\vartheta}_2}\overline{\alpha}_1\overline{\alpha}_2.(\mathcal{D}, \overline{\mathcal{D}}) \rightsquigarrow \mathsf{lift}^{\diamond}\langle\overline{\alpha}_1, x_1 = e_1\ @\ (\mathsf{let}\ A_{I_1} = x_1.\ell_{I_1}\ \mathsf{in}\ e_2)\rangle}$$

REFERENCES

[1] C. Blaudeau, D. Rémy, and G. Radanne. Fulfilling OCaml modules with transparency. *Proc. ACM Program. Lang.*, 8 (OOPSLA1), 4 2024. doi: 10.1145/3649818.