Fulfilling OCaml Modules with Transparency

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

ML modules come as an additional layer on top of a core language to offer large-scale notions of composition and abstraction. They largely contributed to the success of OCaml and SML. While modules are easy to write for common cases, their advanced use may become tricky. Additionally, despite a long line of works, their meta-theory remains difficult to comprehend, with involved soundness proofs. In fact, the module layer of OCaml does not currently have a formal specification and its implementation has some surprising behaviors.

Building on previous translations from ML modules to F^ω, we propose a type system, called M^ω, that covers a large subset of OCaml modules, including both applicative and generative functors, and extended with transparent ascription. This system produces signatures in an OCaml-like syntax extended with F^ω quantifiers. We provide a reverse translation from M^ω signatures to path-based source signatures along with a characterization of *signature avoidance* cases, making M^ω signatures well suited to serve as a new internal representation for a typechecker. The soundness of the type system is shown by elaboration in F^ω. We improve over previous encodings of sealing *within applicative functors*, by the introduction of *transparent existential types*, a weaker form of existential types that can be lifted out of universal and arrow types. This shines a new light on the form of abstraction provided by applicative functors and brings their treatment much closer to those of generative functors.

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. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 101 (April 2024), 29 pages. https://doi.org/10.1145/3649818

1 INTRODUCTION

Modularity is a key concept to build and maintain complex systems. Large code bases are broken down into smaller components, called *modules*, both to give structure to the whole system and to build standardized and reusable units. Complexity is made manageable by channeling interactions through reduced interfaces. To that regard, language-level mechanisms are essential for keeping implementation details and internal invariants hidden from interface, while allowing modules to be combined in subtle ways. A wide variety of modularity techniques appear in different programming languages: from simple functions to libraries, compilation units, objects, type-classes, packages, etc. In languages of the ML family, modularity is provided by a *module system*, which forms a *separate* language layer built on top of the core language. To quote Rossberg [21], "*ML is two languages in one*". The interactions between modules are controlled statically by a strict type system, making

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.

ACM 2475-1421/2024/4-ART101
https://doi.org/10.1145/3649818

modularity work in practice with little run-time overhead. A module is described by its interface, called a *signature*, which serves both as a lightweight specification and as an API.

The OCaml module system is extensively used and particularly rich in features: it provides applicative and generative functors, module aliases, recursive modules, first-class modules, etc. Most sizable OCaml projects use modules to access libraries or define parametric instances of data structures; several successful projects have made heavy use of modules, as in MirageOS [16] where modules and functors are assembled on demand using a DSL [20].

However, despite the successes and the interest of the community regarding ML modules, giving them a formal type-theoretic definition and proving their properties (type soundness, abstraction safety), turned out to be technically involved. In the case of OCaml, the foundational works of Leroy [13, 14], have not been extended to include the numerous new features. The current situation is a module system that is widely used but still unspecified. Adding, modifying, or even fixing a feature requires a deep knowledge of the technical internals of the typechecker. For substantial extensions which could have unforeseen breaking changes, such as transparent ascription or *modular implicits* [29], lacking a specification is a show-stopper.

Combining ideas from many years of research, a successful and elegant approach to model ML-modules is their translation into F^ω, the higher-order polymorphic lambda calculus. Extending the work of Russo [25], who provided a type system interpreting signatures as F^ω types, a milestone was achieved by Rossberg et al. [23], who also gave an elaboration of module expressions as F^ω-terms, proving soundness of the system.

We build on the insights of the work of Rossberg et al. [23], which we adapt and improve for an OCaml-like language extended with *transparent ascription*. To separate the concerns of specification and soundness, we present a standalone type system, called M^ω, that produces signatures in an OCaml-like syntax extended with F^ω quantifiers *à la* Russo [25]—but are actually syntactic sugar over F^ω types. The M^ω system is central to our work: it provides an up-to-date specification of OCaml modules, may serve as a new internal representation of signatures for the typechecker, and can be used to reason about the design and issues of module systems. To ensure type soundness of M^ω, we give an elaboration *à la* Rossberg et al. [23]. Yet, we remove some artifacts and complexity of the treatment of aliasing and, more importantly, of the encoding of applicative functors. This is achieved by introducing *transparent existential types* which enable skolemization and bring the treatment of generative and applicative functors much closer to one another.

Our contributions are:

- The introduction of *transparent* existential types in F^ω, a weak form of existential types that allows their lifting through arrow types and universal quantifiers. Using transparent existentials, we provide a simpler encoding of applicative functors, significantly reducing the difference with the one of generative functors.
- A specification of a large subset of OCaml modules in M^ω, including both applicative and generative functors, and extended with transparent ascription, using ML-style signature syntax but explicit F^ω-style quantifiers.
- An *anchoring* algorithm that translates M^ω signatures back into the path-based source signatures with a principled approach to the signature avoidance problem.
- A source-to-source encoding of *aliasing*—a key to *abstraction safety*, that relies solely on type abstraction, hence removing the need for a primitive treatment by the type system.

Plan. In §2, we start with an overview of the key features, strengths, and weaknesses of the OCaml module system. In §3, we present M^ω, a type system for OCaml modules relying on a richer signature language based on F^ω. In §4, we discuss the signature avoidance problem and introduce *anchoring*—the backward translation from M^ω signatures to source signatures. In §5, we present

```
1  module Complex = struct
2    type t = int * int
3    let zero = (0, 0)
4    let one = (1, 0)
5    let add u v = ...
6  end
7  module type Ring = sig
8    type t
9    val zero : t val one : t
10   val add : t -> t -> t
11 end
12 module CRing = (Complex : Ring)
13 module Polynomials = functor (R : Ring) ->
14   struct
15     type t = R.t list
16     let zero = [] let one = [R.one]
17     let add = ...
18   end
19 module CX = Polynomials(Complex)
20 module PolynomialsXY (R : Ring) =
21     Polynomials(Polynomials(R))
```

Fig. 1. Basic modularity.

the elaboration of modules into F^{ω} terms as an extension of the M^{ω} type system. To this end, we introduce transparent existential types and show how they simplify the treatment of applicative functors. Finally, we discuss related works (§6.1), omitted features (§6.2), and future works (§6.1).

2 A MODERN MODULE SYSTEM

This section is an introduction to the design space of ML-modules focusing on the concepts, problems, and solutions that should help understand the rest of the paper. The ideas are not new and have been discussed in more details in the literature, as in the introduction of [8] for instance. We start with a quick overview of the basic features: structures, signatures, sealing, and functors. We then focus on the difference between generative and applicative functors. The design of applicative functors is linked with the property of *abstraction safety*, which leads to a notion of module-level aliases and *module identity*. To improve the current situation of module aliases, we propose and discuss an extension to *concrete ascription*. Finally, we explain the *signature avoidance* problem.

2.1 Basic ML Modularity

An introductory example is given in Figure 1. Modules are created by gathering term and type definitions in a *structure*, which can be named, as illustrated by module `Complex`. Definitions inside a structure are called *bindings* which can be type declarations (line 2), values (line 3), submodules, or *module types*, also called *signatures* (line 7). Definitions inside a signature are called *declarations*. Type declarations can be left abstract, as the one at line 8.

Module types are used to control interactions between modules in two ways. First, the *outside view* of a module can be restricted to protect internal invariants by an explicit *ascription* to a given module type (line 12). Ascriptions can be used to *hide* fields (making them inaccessible from the outside) or *abstract* type components, which hides the underlying implementation while keeping the name visible. Here, `CRing.t` is an available type, but its implementation as a pair of integers is hidden: one cannot coerce a pair of integers into a `CRing.t` value. Second, module types can be used to restrict how a given module depends on other modules. This is achieved by turning the module into a *functor*. Here, `Polynomials` is a functor that can take any implementation `R` satisfying the `Ring` interface and that returns an implementation of the ring of polynomials over `R`. The *body* of the functor is *polymorphic* with respect to the abstract type fields of its argument, and thus, does not depend on their actual implementations. Functors can then be called and composed: `Polynomials` can be applied to modules satisfying `Ring` such as `Complex` and `CRing` (lines 19), but also the output of `Polynomials` itself (line 21). This check is *structural*, as a module doesn't need to nominally mention the `Ring` signature, it is sufficient to have the appropriate fields. Finally, modules can be packed inside other modules as *sub-modules*, functors can be *higher order*, and ascriptions can be used at any point, allowing functor applications to also produce abstract types.

```
module Tokens () = (struct
    type t = int
    let x = ref 0   ...
  end : sig type t ... end)
module PublicTokens = Tokens()
module PrivateTokens = Tokens()
(** PublicTokens.t =/= PrivateTokens.t *)
```

(a) A generative functor — OCaml functors are made generative by having `()` as their last parameter. Here, each application of the `Tokens` functor produces a module with its own internal state that generates fresh tokens independently.

```
module OrderedSet (E:Ordered) = (struct
    type t = E.t list
    let empty : t = []   ...
  end : sig type t ... end)
module S1 = OrderedSet(Integers)
module S2 = OrderedSet(Integers)
(** S1.t === S2.t *)
```

(b) An applicative functor — Functors are applicative by default in OCaml. Here, `OrderedSets(E)` is a module implementing (ordered) sets of elements of type `E.t`. Applicative functors can be used in paths directly, leading to `S1.t = OrderedSets(Integer).t`.

Fig. 2. Examples of generative and applicative functors.

2.2 Applicative and Generative Functors

Both modules and functors can be used to either *structure* the code base or to build *reusable components*. In the latter case, several instances of a given module might be available in the context when combining different pieces of code. This is typically the case for modules providing common data-structures such as lists, hash-tables, sets, etc. When such a module is the result of a functor application, a question arises: should every instance of the same application produce incompatible abstract types, i.e., types not considered equal by the typechecker? This question leads to the distinction between *applicative* and *generative* functors, which have different semantics and correspond to different use cases. Both are supported by OCaml and illustrated in Figure 2. If two instances have *equal* abstract types, there are effectively *compatible* and the functions and values from each module can be used together. We say that two instances are *incompatible* when they have different abstract types.

Applying a *generative functor* twice *generates* two incompatible modules, with incompatible abstract types. The body of such functor might be stateful, emits effects, or dynamically choose the implementations of its abstract types (using *first-class modules*). Generativity can also be used by programmers as a strong abstraction barrier to force incompatibility between otherwise pure and compatible data-structures that represent different objects in the program. OCaml *syntactically* distinguishes generative functors from applicative ones by requiring the last argument to be a special unit argument "`()`" (we expand on the reasons behind this choice in §3.1).

Conversely, applying an *applicative functor* twice with *the same argument* produces compatible modules, with the same abstract types. The body of such a functor must be pure[1] and have a static implementation of its abstract types. Applicativity acts as a weaker abstraction barrier, making several instances of the same structure compatible. This is especially useful to provide generic functionalities (such as hash-maps[2], sets, lists, etc.) that may appear in several places and yet be compatible. Applicative functors are the default in OCaml.

2.3 Abstraction Safety and Granularity of Applicativity

A key design point is the *granularity* of applicative functors: under what criterion should two applications of a functor produce compatible modules? We say that two modules are *similar* when applying the same functor to both yields compatible modules. An option, used in Moscow ML, is to consider modules to be similar when they have the same type fields (same names and same definitions). This criterion is called *static equivalence* [7, 24, 27]. It is *type-safe*, as the actual

[1] In OCaml, it is left to the user's responsibility to mark impure functors as generative, the typechecker does not track effects, only preventing unpacking of first-class modules and calling generative functors inside the body of applicative ones.
[2] `Hashtbl.Make` is actually pure, as it does not produce a new hash table itself, even though it contains impure functions.

implementation of the abstract types produced by the functor can only depend statically[3] on its parameters, thus only on its *statically known* type fields.

However, the static equivalence criterion can make two functor applications compatible while they actually have different internal invariants. In the example on Figure 2, the lists used to represent sets are ordered with respect to the comparison function of `E`. Therefore, a typechecker implementing the static equivalence criterion would allow a user to mix sets ordered with different ordering functions, which would produce wrong results—but not crash.

Yet, developers often expect a stronger property called *abstraction safety*: abstract types should protect arbitrary local invariants that may also depend on values. In the example on Figure 2, applications of `OrderedSets` should produce compatible abstract types only when both the type `E.t` and the values, in particular `E.compare`, are the same.

To preserve abstraction safety, modules should be deemed similar only when both their type *and* value fields are equal. Unfortunately, the equality (or equivalence) of values is undecidable in general. Besides, tracking even an approximation of the equality of value fields would be too fine-grained and cumbersome, as modules may have numerous value fields. To enforce abstraction safety while remaining practical, OCaml follows a *coarse-grain approach*: tracking equalities only at the module level. This was originally introduced as a *syntactic criterion* by Leroy [13]: two functor applications produce the same abstract types when they are syntactically identical.

2.4 Aliasing and Ascription

The *syntactic criterion* is however limited: when *aliasing* a module, as in the expression `module X' = X`, `X'` and `X` are not considered similar—they are syntactically different. To allow for a better tracking of module equalities, *module aliases*[4] [9] were added to OCaml: the signature language was extended with the *alias signature* construct[5] $(= P)$ to express that a module is a *statically known alias* of the module at the path P. Keeping as much aliasing information as possible allows for more type equalities when using applicative functors.

Unfortunately, the aliasing feature of OCaml is quite restricted, as it does not compose with functors. As an example, let us consider the code snippet to the right. The module `X'` cannot be given the expected alias signature `(= X)`, which contradicts the substitution-based intuition. If the type system were to maintain module aliases through functor calls, it would impose strong constraints on the compilation of structures and functors that would drastically affect the performance trade-offs of modules. Indeed, in all ML-module systems, and OCaml in particular, structures are accessed using static dispatch, i.e., with statically known offsets to be fast. As a counterpart, the dynamic and static view must coincide and subtyping, which changes the static view, is not code free. That is, explicit coercions are inserted at functor calls: typically, the functor `F` does not receive `X` of signature `T` as argument but a copy with fewer, reordered fields, as described by the signature `S` of the parameter `Y`. Therefore, the functor parameter `Y` cannot be given the alias type `(=X)` which should at least be compatible with the type `T`[6]. Hence, the OCaml type

```
1 module X : T = (* ... *)
2 module F (Y:S) = Y (* reexport *)
3 module X' = F(X)
```

[3] In the absence of first-class modules, which are forbidden in applicative functor for that very reason.

[4] OCaml actually offers two distinct notions of aliases, which are respectively *present* or *absent* at runtime. Here and in the rest of the paper, we only consider *present* aliases, even though they are not the default behavior, as they are the ones that pose a theoretical challenge. Historically, the main motivation behind module aliases was rather the fine-grained control of compilation units and name-spaces with absent aliases than the interaction with applicative functors and present aliases.

[5] OCaml actually only has alias *declarations* `module X = P` inside structural signatures. We present here alias *signatures* for the sake of simplicity, which correspond to adding an enclosing structure with an alias declaration.

[6] Keeping aliases through functor calls would amount to give the functor a type that subsumes the bounded polymorphic type `∀ (A ≤ S). A -> A`, which, for records/structures, requires code-free subtyping.

system does not follow the naive substitution semantics for aliases inside functors. Instead, it uses a set of syntactically-based restrictions to prevent aliases in functor signatures. However, those restrictions are not stable under substitution—and can currently be bypassed in some edge-cases[7].

Interestingly, *transparent ascription*, originally introduced as a module expression written `(M : S)` in SML, helps lifting this restriction. It restricts the outside view of a module `M` to the fields present in the signature `S` while preserving all type equalities. However, this feature does not increase expressiveness in SML as a similar result could be obtained via a *usual* (opaque) ascription of SML with a signature where all type equalities have been made explicit. A proposal for OCaml[8] is to add transparent ascription as an extension not only of the module language, but of the *signature* language, writing `(= P < S)` for the signature of a module that is an alias of `P` but restricted to the fields of `S`, which we call a *concrete signature*. A module with such a signature has the *identity* of `P` and the content `S`. Concrete signatures provide a generalization of aliasing, storing both the aliasing information *and* the actual signature (hence, the memory representation). The transparent ascription *expression à la* SML `(M : S)`, is then just syntactic sugar for an opaque ascription with a concrete signature (see §3.1). Thanks to concrete signatures, aliasing information can be preserved through the implicit ascription at functor calls. As OCaml features applicative functors (unlike SML), this would increase the expressiveness of the signature language. Besides, concrete signatures are compatible with static dispatch and copying at function calls allows deletion and reordering of fields while keeping type equalities. Concrete signatures `(= P < S)` are a special case of the more general module sharing mechanism of *F-ing* [23], which could be obtained as `(like (P : S))`.[9]

Finally, module identity is essential for *modular implicits* [29], a proposal for adding inference of module expressions from a pre-declared set of modules and functors. In order to ensure coherence, one must guarantee that an inferred module is unique, up to some notion of equivalence. As concrete signatures enable more sharing of identities in signatures of inferred modules, aliasing becomes a good static approximation of that equivalence.

2.5 A Key Weakness: the Signature Avoidance Problem

The *signature avoidance* problem is a key issue of ML module systems. It originates from a mismatch between the expressiveness of the module and signature languages: the *reachable space* of possible module expressions is larger than the *describable space* of signatures: some modules simply cannot be described by a signature. This mismatch is caused by the interaction of three mechanisms. First, type abstraction creates new types that are only compatible with themselves (and their aliases). Then, sharing abstract types between modules, which is essential for module interactions, produces inter-module dependencies. Finally, hiding type or module components (either by a projection or by implicit subtyping at a functor application) can *break* such dependencies by removing type aliases from scope while they are still being referenced. For instance, an abstract type `t` can be hidden while a value of type `t list` is still in scope. An example of such pattern is given in Figure 3. Sometimes, no possible signature exists for a module; other times there are several incomparable ones. Specifically with applicative functors, when higher-order abstract types are out of scope, there are often *only* incomparable solutions.

Strategies for solving signature avoidance. When a type declaration refers to an out-of-scope type, there are three main strategies to correct the signature: (1) removing the dependency by making the

[7] See the following issues: OCaml #7818, OCaml #2051, OCaml #10435, OCaml #10612 and OCaml #11441.
[8] OCaml #10612. Transparent ascription is written `(P :> S)` in OCaml #10612, the opposite of the SML convention.
[9] This sharing mechanism relies on a general notion of *semantic* paths (which stand for any module expression that does not introduce new abstract types) whereas OCaml uses more restrictive *syntactic* paths. We argue that for the latter, concrete signatures are an interesting expressiveness trade-off.

```
module type S = sig type t end
module M = (struct
  module X:S = struct type t = int end
  module Y = struct
    type a = X.t * bool
    type b = X.t * int
  end
end).Y
let f ((x,_): M.a): M.b = (x, 3)
```

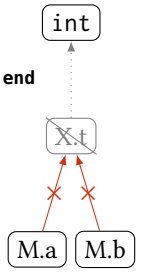

Fig. 3. Example of a signature avoidance situation and the associated type-dependencies tree. The module `M` is built by projecting *only* the submodule `Y`, which exposes unsolvable dependencies with a type (`X.t`) that became unreachable. The function `f` is therefore not well-typed. (Here, we use a general projection not present in current OCaml, but can be easily reproduced with an anonymous functor call, as done in [2, ??].)

type declaration abstract, (2) rewriting the type equalities using in-scope aliases, or (3) extending the signature syntax to account for the existence of out-of-scope types. The first strategy (1) can lead to loss of type equalities, but is easy to implement—it is the one currently in use in the OCaml typechecker. Cases where the second strategy (2) succeeds constitute the *solvable* cases of signature avoidance. The OCaml typechecker has some heuristics for rewriting type-equalities, but they are incomplete, lacking a notion of *equivalence class*. This results in unpredictable, hard to understand signature avoidance errors that should, in principle, be solvable. Sometimes, no *in-scope* alias is available and signature avoidance cannot be solved without an extended syntax: those are the *general* cases of signature avoidance. We advocate for the third approach (3), embodied by M^ω and presented in §3—at least for the internal representation of the typechecker—which allows us to separate the type system from the issue of dealing with the signature avoidance problem. However, there are associated challenges. If the extended language is only used as an internal representation, then a reverse translation is needed for printing the result to the user and for error messages. This reverse translation has to deal with signature avoidance cases. If instead, the extended language is made accessible to the user, the decidability of type-checking is not guaranteed in the presence of higher-order abstract types; besides, it is still unclear whether it would be practical.

Signature avoidance in practice. OCaml users usually get around this limitation by explicitly naming modules before using them, which adds *always-accessible* type definitions. The module syntax of OCaml actually encourages this approach by limiting the places where inlined, anonymous modules can be used. In particular, projection on an anonymous module (as done in Figure 3) is forbidden. However, explicit naming can be cumbersome and limits the usability of module-based programming patterns such as modular implicits.

3 THE QUANTIFIER-BASED M^ω APPROACH

In this section, we present the type system M^ω that covers the set of features informally explained in the previous section without suffering from the signature avoidance problem. M^ω distinguishes between source signatures written by the user and M^ω signatures used for typechecking. M^ω signatures use explicit binders (existential, universal, lambda) as in F^ω (and *F-ing*) to express type abstraction and polymorphism, including applicativity and generativity. We start with the grammar of source expressions (§3.1) and an overview of M^ω (§3.2). Then, we present the three main typing judgments with a *type-only* granularity of applicativity (§3.3, §3.4, and §3.5). To model the OCaml style applicativity, we show how *module identity* and *aliasing* can be piggybacked on the type abstraction mechanism by a simple source-to-source transformation (§3.6).

3.1 The Source Language

The source grammar, given on Figure 4, is built on top of a core language of expressions e and types u which are mostly left abstract. We only consider value identifiers x and type identifiers t, extended with *qualified values* $Q.x$ and *qualified types* $Q.t$: these are the only way for the core level

Path and Prefix

$P ::= Q.X$ (Access)
$\mid Y$ (Functor parameter)
$\mid P(P)$ (Applicative application)
$Q ::= A \mid P$ (Prefix)

Module Expression

$\mathtt{M} ::= P$ (Path)
$\mid \mathtt{M}.X$ (Projection)
$\mid (P : \mathtt{S})$ (Ascription)
$\mid P()$ (Generative application)
$\mid () \to \mathtt{M}$ (Generative functor)
$\mid (Y : \mathtt{S}) \to \mathtt{M}$ (Applicative functor)
$\mid \mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end}$ (Structure)

Binding

$\mathtt{B} ::= \mathtt{let}\, x = \mathtt{e}$ (Value)
$\mid \mathtt{type}\, t = \mathtt{u}$ (Type)
$\mid \mathtt{module}\, X = \mathtt{M}$ (Module)
$\mid \mathtt{module\ type}\, T = \mathtt{S}$ (Module type)

Source signature

$\mathtt{S} ::= Q.T$ (Module type)
$\mid () \to \mathtt{S}$ (Generative functor)
$\mid (Y : \mathtt{S}_a) \to \mathtt{S}$ (Applicative functor)
$\mid \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end}$ (Structural signature)
$\mid (= P \leq \mathtt{S})$ (Concrete signature)

Source declaration

$\mathtt{D} ::= \mathtt{val}\, x : \mathtt{u}$ (Value)
$\mid \mathtt{type}\, t = \mathtt{u}$ (Type)
$\mid \mathtt{module}\, X : \mathtt{S}$ (Module)
$\mid \mathtt{module\ type}\, T = \mathtt{S}$ (Module type)

Identifier

$I ::= x \mid t \mid X \mid Y \mid T$ (Any identifier)

Core language

$\mathtt{e} ::= Q.x$ (Qualified value)
$\mid \ldots$ (Other expression)
$\mathtt{u} ::= Q.t$ (Qualified type)
$\mid \ldots$ (Other type)

Fig. 4. Syntax of the source language.

$\mathtt{M}^\omega$ Types

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

Environment

$\Gamma ::= \varnothing$ (Empty)
$\mid \Gamma, \alpha$ (Abstract type)
$\mid \Gamma, (Y : \mathcal{C})$ (Functor parameter)
$\mid \Gamma, (A.\mathcal{D})$ (Declaration)

Opacity

$\diamond ::= \triangledown$ (Transparent) $\mid \blacktriangledown$ (Opaque)

$\mathtt{M}^\omega$ signature

$\mathcal{C} ::= \mathtt{sig}\ \overline{\mathcal{D}}\ \mathtt{end}$ (Structural signature)
$\mid \forall\overline{\alpha}.\mathcal{C} \to \mathcal{C}$ (Applicative functor)
$\mid () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}$ (Generative functor)

$\mathtt{M}^\omega$ declaration

$\mathcal{D} ::= \mathtt{val}\ x : \tau$ (Values)
$\mid \mathtt{type}\ t = \tau$ (Types)
$\mid \mathtt{module}\ X : \mathcal{C}$ (Modules)
$\mid \mathtt{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}$ (Module types)

Fig. 5. Syntax of $\mathtt{M}^\omega$ signatures.

to access the module level. The abstract syntax of module expressions and signatures is rather standard and mostly follows the current OCaml syntax. There are a few minor technical choices:

- Module-related meta-variables use *typewriter uppercase letters*, M, S, etc., while lowercase letters are used for expressions and types of the core language. Lists are written with an overhead bar. Identifiers I and paths P use a standard font.
- In order to simplify the treatment of scoping and shadowing, we introduce *self-references*, ranged over by letter A, in both structures and signatures. They are used to refer to the current object; their binding occurrence appears as a subscript to the structure or signature they belong to ($\mathtt{struct}_A$ ... end), so that self-references can freely be renamed. They are not present in OCaml and should be thought of as being added by a first pass before typing. We explain how they help treat shadowing in §3.2.
- In order to have a uniform treatment for access to local and non-local variables, we use *prefixes*, written with the letter Q, to range over either a path P or a self reference A.

- *Abstract types* are specified as types pointing to themselves, e.g., $\texttt{type}\, t = A.t$ where A is the self-reference of the current structure (and often grayed out for readability).
- Functor parameters Y are α-convertible. The other identifiers (X, T, x and t) are not, as they play the role of both internal and external names.

Projectibility. Choosing (1) whether projection is allowed on any module expression or only on a restricted subset, and (2) how the core language can refer to values and types of modules is an important design choice in ML systems, coined *projectibility* by Dreyer et al. [7]. Contrary to *F-ing*, but following Leroy [13] and Russo [25], we chose to use a *syntactic* notion of path.

- We allow projection on any module expression, but we restrict functor applications and ascriptions to paths. Doing so, the only expression that can "cause" signature avoidance is the projection. OCaml does the opposite, mainly to prevent code patterns prone to triggering signature avoidance. Our choice is more general, as we can define a *let* construct for modules using the following syntactic sugar:

$$\mathsf{let}\ X = \mathtt{M}\ \mathsf{in}\ \mathtt{M}' \triangleq (\mathtt{struct}_A\ \ \mathtt{module}\, X = \mathtt{M}\ \ \mathtt{module}\, \mathsf{Res} = \mathtt{M}'\ \mathsf{end}).\mathsf{Res}$$

  Using this construct, we easily get functor application and ascription on arbitrary module expressions. The reverse encoding of projection as an anonymous functor call requires an explicit signature annotation on the argument and thus cannot be seen as syntactic sugar.
- A qualified access inside a generative functor application, which would be of the form $G().t$, is syntactically ill-formed, as paths do not contain the unit argument $()$. By contrast, a qualified access inside an applicative functor application $F(X).t$ is permitted.
- A qualified access inside a module type, which would be of the form $Q.T.t$, is syntactically ill-formed, as paths do not contain module type identifiers T.
- We only provide opaque ascription in module expressions, as concrete ascription is given by the following syntactic sugar: $(P < \mathtt{S}) \triangleq (P : (= P \leq \mathtt{S}))$

As both path and module expressions feature a projection dot, the grammar is slightly ambiguous. However, this is not a problem as we see paths as a subset of module expressions. In particular, we only consider the projection dot of module expressions in the typing rules.

N-ary functors. As in OCaml, our grammar features unary applicative functors and nullary generative functors. A unary generative functor can be obtained as an applicative functor returning a generative one. Indeed, we could add the usual currying notation sugar:

$$(Y : \mathtt{S})\ () \to \mathtt{M} \triangleq (Y : \mathtt{S}) \to (() \to \mathtt{M})$$

While n-ary applicative functors are straightforward, one might wonder if n-ary generative functors require a unit argument between every parameter. Actually, the `()` acts as a *generative barrier* and can be placed to control the sharing between partial applications .

3.2 M^ω Overview

In Figure 5, we introduce the syntax for M^ω-signatures $\mathcal{C}$, a more expressive signature language. By convention, we use curvy capitals ($\mathcal{C}, \mathcal{D}, \ldots$) for M^ω-objects, which also use M^ω-types τ instead of source types u, obtained by replacing qualified types $Q.t$ by abstract types α or applied abstract types $\alpha(\overline{\tau})$ (or concrete types τ) where α range over a new collection of abstract type variables.

M^ω signatures $\mathcal{C}$ use explicit quantifiers: universal quantification for functor parameters, and existential for the body of generative functors. We annotate existential types with an opacity flag $\diamond$ to indicate generativity (using the opaque flag $\blacktriangledown$) or applicativity (using the transparent[10] flag

[10]This notion of transparency is *unrelated* to the transparent *vs.* concrete notion of ascription.

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

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

$$\textsc{M-Typ-Sig-Str}\ \frac{\Gamma \vdash_A \overline{\mathsf{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathsf{sig}_A\ \overline{\mathsf{D}}\ \mathsf{end} : \lambda\overline{\alpha}.\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}} \qquad \textsc{M-Typ-Sig-Con}\ \frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} \le \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]}{\Gamma \vdash (= P \le \mathsf{S}) : \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]}$$

Fig. 6. Signature typing rules.

∇). Transparent existentials do not appear directly in the grammar of Figure 5 but in the typing judgment for module expressions, which uses existentially quantified signatures of either form $\exists^{\nabla}\overline{\alpha}.\mathcal{C}$ or $\exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}$. Module types $\lambda\overline{\alpha}.\mathcal{C}$ are parametric[11] in each type variable α, which may later become universally quantified, existentially quantified, or replaced by a concrete instance. We consider M^{ω} types up to $\alpha\beta\eta$-equivalence.

Typing environments contain three types of bindings: an abstract type variable α, a functor argument $Y : \mathcal{C}$, or a declaration $A.\mathcal{D}$. OCaml allows some form of shadowing, which, for the sake of simplicity, we reject by requiring typing environments to have distinct bindings. However, in addition, bindings made inside a submodule can also locally shadow a definition made in an enclosing structure. We authorize this form of shadowing, as bindings made inside different structures would have a different self-reference prefix (and are therefore considered as two different bindings). Technically, this is achieved by using a *well-formedness* predicate over environments $\mathsf{wf}(\Gamma)$ that, in addition to recursively checking well-formedness of all bindings in Γ, ensures that variables α and Y are bound at most once and that two bindings of the form $A.\mathcal{D}_1$ and $A.\mathcal{D}_2$ of the same self-reference variable A may only occur when $\mathcal{D}_1$ and $\mathcal{D}_2$ define disjoint identifiers.[12] As a simplifying convention for the rest of this paper, we consider well-formedness of the environment as a precondition to all rules.

Besides the changes in the source language, and the annotation of existential quantifiers with modes, the M^{ω} typing judgments are very close to the ones of *F-ing* [23], but stripped of the elaborated terms. We refer to it and to [2] for more details. There are three main judgments: typechecking of signatures (and declarations), subtyping, and typechecking of module expressions (and bindings), which we present in this order.

A "standard" typing system. Overall, the type system should be understood as a combination of standard features—but using ML-modules nomenclature—together with specific mechanisms to deal with abstraction. Structures are just records, with bindings being record-field expressions and declarations being record-field types. Ascriptions are explicit coercions, and functors are functions. However, in addition, some constructs introduce abstract type variables with an *implicit scope*. Hence, the key technical point of M^{ω} is how the type sharing is represented with *explicit* quantifiers.

3.3 Typechecking of Signatures

The key concepts of M^{ω} can be illustrated with the typechecking of signatures $\Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}$, which translates a source signature S into its M^{ω} counterpart $\lambda\overline{\alpha}.\mathcal{C}$, making the set of abstract type $\overline{\alpha}$ explicit. The set of rules is given in Figure 6 and discussed below.

[11] Here, we follow Russo [25] rather than [23] and use a lambda quantifier for signatures.
[12] This amounts to see $A.\mathcal{D}$ as binding A_I where I is the identifier defined by $\mathcal{D}$.

Declarations. Typechecking of signatures uses a helper judgment $\Gamma \vdash_A \mathtt{D} : \lambda\overline{\alpha}.\mathcal{D}$ for typechecking declarations for which we only give the key rules, referring to [2, **??**] for the full set of rules. The syntactic enforcement of the position of quantifiers in this judgment helps understand the lifting of abstract types, a key concept that is pervasive throughout the declaration typing rules. An abstract type is introduced by an abstract type declarations:

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

However, since type t must also be accessible in the following declarations, the λ-binder for α must be *lifted* to enclose the whole region where t, hence α, is accessible. This lifting is performed when merging a list of declarations in Rule M-Typ-Decl-Seq where binders are merged together in front of the list of declarations.

$$\text{M-Typ-Decl-Seq}\ \frac{\Gamma \vdash_A \mathtt{D}_0 : \lambda\overline{\alpha}_0.\mathcal{D}_0 \qquad \Gamma, \overline{\alpha}_0, A.\mathcal{D}_0 \vdash_A \overline{\mathtt{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \vdash_A \mathtt{D}_0, \overline{\mathtt{D}} : \lambda\overline{\alpha}_0\,\overline{\alpha}.\mathcal{D}_0, \overline{\mathcal{D}}} \qquad \text{M-Typ-Decl-Mod}\ \frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module}\, X : \mathtt{S}) : \lambda\overline{\alpha}.(\mathtt{module}\, X : \mathcal{C})}$$

Lifting also occurs when typing a submodule declaration (Rule M-Typ-Decl-Mod) where the set of quantified types introduced by the submodule are lifted to the declaration itself.

Signatures. Typing rules for signatures can be found on Figure 6. Module type definitions are inlined (rules M-Typ-Sig-ModType and M-Typ-Sig-LocalModType), which explains why M^ω signatures do not have a counterpart for module types $Q.T$ in source signatures. In Rule M-Typ-Sig-Con for a concrete signature ($= P \leq \mathtt{S}$), the signature $\mathtt{S}$ is first elaborated into an M^ω-signature $\lambda\overline{\alpha}.\mathcal{C}'$ that is checked against the M^ω-signature $\mathcal{C}$ of path P. The result signature is $\mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]$ after applying the matching substitution $[\overline{\alpha} \mapsto \overline{\tau}]$ to $\mathcal{C}'$. Notably, no new abstract type is introduced.

Functors and scopes. Rule M-Typ-Sig-GenFct for generative functors shows how the scope of abstract types $\overline{\alpha}$ introduced in their bodies is restricted the top-level of the functor body, leaving an opaque existential signature $\exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}$ for the functor codomain. Hence, every instantiation of the functor will generate *new* (incompatible) abstract types $\overline{\alpha}$, as expected for generative functors. By contrast, applicative functors should not be assigned a signature of the form $\forall\overline{\alpha}.\mathcal{C} \to \exists^{\blacktriangledown}\overline{\beta}.\mathcal{C}'$ where all applications would produce new types, nor $\lambda\beta.\forall\overline{\alpha}.\mathcal{C} \to \mathcal{C}'$ where all applications would share the same types regardless of their argument. The solution, introduced by Biswas [1] and reused in *F-ing*, is to use a higher-order abstract type β' applied to the universally quantified variables $\overline{\alpha}$ to capture the fact that the type of the codomain is *some type function* of the arguments. This gives a signature of the form $\lambda\overline{\beta'}.\forall\overline{\alpha}.\mathcal{C} \to \mathcal{C}[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}]$ as can be seen in Rule M-Typ-Sig-AppFct.

3.4 Subtyping

The subtyping judgment $\Gamma \vdash \mathcal{C} \leq \mathcal{C}'$ (and its helper judgment $\Gamma \vdash \mathcal{D} \leq \mathcal{D}'$) checks that a signature $\mathcal{C}$ is *more restrictive* than a signature $\mathcal{C}'$, meaning that the former has more fields and introduces fewer abstract types. It combines structural subtyping of records (with field deletion and reordering), covariant subtyping of functions, and instantiation of quantifiers. We only highlight two key rules below, referring to [2, **??**] for the full set of rules:

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

Rule M-Sub-Sig-Struct compares two structural signatures. Rule M-Sub-Sig-GenFct for generative functors amounts to check subtyping between existential types $\Gamma \vdash \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \leq \exists^{\blacktriangledown}\overline{\alpha'}.\mathcal{C}'$, which in turn amounts to finding an instantiation $[\overline{\alpha'} \mapsto \overline{\tau}]$ of the abstract types so that $\mathcal{C}$ is a subtype of $\mathcal{C}'[\overline{\alpha'} \mapsto \overline{\tau}]$. While this is the standard way of specifying subtyping for existential types, it is

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

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

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

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

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

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

Fig. 7. Module (and path) typing rules.

algorithmically challenging in the presence of higher-order abstract types, and could potentially lead to undecidability of subtyping. This problem has already been identified by Rossberg et al. [23], but shown to be decidable for certain pairs of signatures $(\mathcal{C}, \mathcal{C}')$ satisfying a syntactic condition,[13] which happens to be true for signatures encountered during subtyping. This results from the fact that subtyping is always checked against signatures $\mathcal{C}'$ that are the elaboration of source signatures.

3.5 Typechecking of Module Expressions

Typechecking of expressions $\Gamma \vdash \mathtt{M} : \exists^{\diamond} \overline{\alpha}.\mathcal{C}$ infers the M^{ω}-signature $\exists^{\diamond} \overline{\alpha}.\mathcal{C}$ of a source module $\mathtt{M}$. The signature features an existential quantification annotated with an opacity flag $\diamond$. In particular, the opacity is the same for all abstract variables (which are all transparent or all opaque). In fact, the judgment should be read $\Gamma \vdash^{\diamond} \mathtt{M} : \exists^{\diamond} \overline{\alpha}.\mathcal{C}$ where the opacity flag on the judgment is a typing mode, *applicative* or *generative*, respectively, which implies that the existentials, if any, should all be transparent or all be opaque, respectively. However, to lighten the notation, we omit the mode except when it is generative and there is no existential type to enforce it. Thus, when we write $\Gamma \vdash \mathtt{M} : \exists^{\blacktriangledown} \varnothing.\mathcal{C}$ or $\Gamma \vdash \mathtt{M} : \exists^{\triangledown} \varnothing.\mathcal{C}$ when $\overline{\alpha}$ is empty, we actually mean $\Gamma \vdash^{\blacktriangledown} \mathtt{M} : \mathcal{C}$ and $\Gamma \vdash^{\triangledown} \mathtt{M} : \mathcal{C}$. The same convention applies to typing rules for bindings M-Typ-Decl-* which can be found in [2, **??**]. Typing rules for expressions M-Typ-Mod-* are given on Figure 7.

Skolemization. The need for two modes of typing comes from the treatment of applicative functors, specifically Rule M-Typ-Mod-AppFct. In order to share the abstract types $\overline{\beta}$ produced by the body of the functor, we lift them out of the universal quantification (and out of the right-hand side of the arrow) by making them higher-order. This is known as *skolemization* and has been introduced by Russo [25] in the context of modules. However, this is only sound when the abstract types have a statically known witness,[14] which we enforce by requiring transparent existentials for the body of the functor.

Propagation of modes. Signatures with transparent existentials are inferred by default and are *required* for the body of applicative functors. Module expressions that are inherently generative, such as calling a generative functor or computing impure core expressions (or unpacking a first-class module), can only be typed with opaque existential signatures in generative mode. This discipline is enforced by forcing the body of a functor to be typed transparently when it is applicative

[13] [23] defines the syntactic notions of *valid* and *explicit* signatures. They enforce that during typechecking, subtyping only happens between valid signatures on the right-hand side and explicit signatures on the left, for which it is decidable.
[14] Technical reasons will be given in §5.4.

(Rule M-Typ-Mod-AppFct) and opaquely when it is generative (Rule M-Typ-Mod-GenFct). Rule M-Typ-Bind-Seq forces all components of a structural signature to have the same opacity:

$$\text{M-Typ-Bind-Seq}\quad \frac{\Gamma \vdash_A \mathsf{B}_0 : \exists^{\diamond}\overline{\alpha}_0.\mathcal{D} \qquad \Gamma, \overline{\alpha}_0, A.\mathcal{D} \vdash_A \overline{\mathsf{B}} : \exists^{\diamond}\overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \vdash_A \mathsf{B}_0, \overline{\mathsf{B}} : \exists^{\diamond}\overline{\alpha}_0, \overline{\alpha}.\mathcal{D}, \overline{\mathcal{D}}} \qquad \text{M-Typ-Bind-Let}\quad \frac{\Gamma \vdash^{\diamond} \mathsf{e} : \tau}{\Gamma \vdash^{\diamond}_A (\mathtt{let}\ x = \mathsf{e}) : (\mathtt{val}\ x : \tau)}$$

We also rely on a core-language expression typing judgment[15] $\Gamma \vdash^{\diamond} \mathsf{e} : \tau$ equipped with a mode that tracks the presence of effects[16]. When typing a value field, the mode is propagated via an empty existential (Rule M-Typ-Bind-Let). Signatures can also be *downgraded* from transparent to opaque via subsumption (Rule M-Typ-Mod-Seal). All other rules are agnostic of the typing mode. With the convention that M-Typ-Mod-Seal is only used when the generative mode is required for the premise of another rule, i.e., applicative signatures are inferred by default, the system is syntax directed.

Introduction of abstract types. Rule M-Typ-Mod-Ascr for signature ascription ($P : \mathsf{S}$) has some resemblance with Rule M-Typ-Sig-Con for typechecking of concrete signatures ($= P \leq \mathsf{S}$): in both cases, we check that the M^{ω} signature of P is a subtype of the M^{ω}-signature $\lambda\overline{\alpha}.\mathcal{C}'$ of S. By contrast, however, we here drop the matching substitution in the result signature $\exists^{\nabla}\overline{\alpha}.\mathcal{C}'$ and instead introduce the abstract types $\overline{\alpha}$, transparently. In particular, when S is concrete, i.e., $\overline{\alpha}$ is empty, no abstract type is actually introduced. That is, concrete ascription ($P <: \mathsf{S}$), which is syntactic sugar for ($P : (= P \leq \mathsf{S})$), i.e., the opaque ascription of P to the concrete signature ($= P \leq \mathsf{S}$), behaves as expected, filtering out components of P as prescribed by S but without creating new abstract types. Note that applications of an applicative functor (Rule M-Typ-Mod-AppApp) do not introduce new abstract types *per se*, but *applications* of *already existing* higher-order abstract types—which is the key to the sharing between different applications of the same (or an equivalent) functor to the same (or equivalent) arguments.

Projection and signature avoidance. In the source signature syntax, dependencies between modules are hard to track, as modules can use arbitrary paths to access other modules. Signatures can thus have non-obvious internal dependencies. In a type system that works directly on source signatures, as is OCaml, typechecking the projection of a submodule $\mathsf{M}.X$ is delicate: the dependencies of the source signature of X might become dangling after the other components of the signature of M have been lost. By contrast, M^{ω} signatures do not have internal dependencies; they are non-dependent records, as all paths present in concrete type definitions have been inlined and binders for abstract types have been lifted.

In principle, the projection rule M-Typ-Mod-Proj could return the signature $\exists^{\diamond}\overline{\alpha}.\mathcal{C}$, leaving all variables in scope after projection. However, it performs some form of garbage collection by just keeping the subset $\overline{\alpha}'$ of abstract types $\overline{\alpha}$ that appear free in the submodule signature $\mathcal{C}$ so as to avoid keeping useless, unreachable abstract types.

3.6 Identity, Aliasing, and Type Abstraction

So far, our system handles applicativity with a granularity of *type-only* applicativity, as promoted by [25] and *F-ing*. To obtain *abstraction safety*, Rossberg et al. [23] introduced *semantic paths*: marking value and module fields with *phantom* abstract types and using the type sharing mechanism to track value (or module) sharing. Then, type-only applicativity can be transformed into either (1) fine-grained applicativity by marking all values, or (2) coarse-grained applicativity (*à la OCaml*) by marking only modules.

[15] This judgment is trivially extended by rules for accessing qualified variables and types, see [2, ??].
[16] Not present in current OCaml, where it is the user's responsibility to use the generative functors in such cases.

Tagging

$$\mathtt{Tag}\,\{\mathtt{M}\} \triangleq \mathtt{struct}_A\ \mathtt{module}\ \mathit{Val} = \mathtt{M}\ \mathtt{type}\ \mathit{id} = A.\mathit{id}\ \mathtt{end}$$
$$\mathtt{Tag}\,\{\mathtt{S}\} \triangleq \mathtt{sig}_A\ \mathtt{module}\ \mathit{Val} : \mathtt{S}\ \mathtt{type}\ \mathit{id} = A.\mathit{id}\ \mathtt{end}$$

Paths

$$[\![A.X]\!] \triangleq A.X \qquad [\![P.X]\!] \triangleq [\![P]\!].\mathit{Val}.X$$
$$[\![Y]\!] \triangleq Y \qquad [\![P(P')]\!] \triangleq [\![P]\!].\mathit{Val}([\![P']\!])$$

Module expressions

$$[\![\mathtt{M}.X]\!] \triangleq [\![\mathtt{M}]\!].\mathit{Val}.X \qquad [\![P()]\!] \triangleq [\![P]\!].\mathit{Val}()$$
$$[\![(P : \mathtt{S})]\!] \triangleq ([\![P]\!] : [\![\mathtt{S}]\!])$$
$$[\![() \to \mathtt{M}]\!] \triangleq \mathtt{Tag}\,\{() \to [\![\mathtt{M}]\!]\}$$
$$[\![(Y : \mathtt{S}) \to \mathtt{M}]\!] \triangleq \mathtt{Tag}\,\{(Y : [\![\mathtt{S}]\!]) \to [\![\mathtt{M}]\!]\}$$
$$[\![\mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end}]\!] \triangleq \mathtt{Tag}\,\{\mathtt{struct}_A\ [\![\overline{\mathtt{B}}]\!]\ \mathtt{end}\}$$

Signatures

$$[\![A.T]\!] \triangleq A.T \qquad [\![P.T]\!] \triangleq [\![P]\!].\mathit{Val}.T$$
$$[\![(= P \le \mathtt{S})]\!] \triangleq (= [\![P]\!] \le [\![\mathtt{S}]\!])$$
$$[\![() \to \mathtt{S}]\!] \triangleq \mathtt{Tag}\,\{() \to [\![\mathtt{S}]\!]\}$$
$$[\![(Y : \mathtt{S}_a) \to \mathtt{S}]\!] \triangleq \mathtt{Tag}\,\{(Y : [\![\mathtt{S}_a]\!]) \to [\![\mathtt{S}]\!]\}$$
$$[\![\mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end}]\!] \triangleq \mathtt{Tag}\,\{\mathtt{sig}_A\ [\![\overline{\mathtt{D}}]\!]\ \mathtt{end}\}$$

Fig. 8. Source-to-source transformation introducing identity tags for structures and functors using two reserved identifiers `id` and `Val`. Bindings and declarations are transformed by immediate map over submodules and submodule-types.

However, as phantom abstract types act *exactly* as regular abstract types, we can split the introduction of those types from the typing. We propose a simple, compositional *source-to-source* transformation that explicitly introduces a special abstract type field *id*, called an *identity tag* in Figure 8. We call *tagged* expressions those resulting from the transformation, so as to distinguish them from *raw* (untagged) expressions. Structures and functors are wrapped inside a two-field structure with its identity tag and the actual value. New (abstract) identity tags are introduced when typing structures and functors, or via an ascription. Conversely, identity tags are shared when aliasing a module.

Controlling the applicativity granularity by a source-to-source transformation allows for a simpler set of typing rules. Besides, it leaves open the choice to apply the transformation so as to obtain OCaml coarse granularity (and abstraction safety), or just stay with the default static equivalence. (Of course, we may recover derived rules by inlining the transformation, which may be useful, for instance in an optimized implementation, to avoid an intermediate pass.)

Identity tags ensure that two module expressions that share the same identity tag *originate* from a common ancestor with a better signature, as stated by the following theorem:

Theorem 3.1 (Identity tags).

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

Proof sketch. The proof uses bounded polymorphism to add a supertype bound to every identity tag, namely the signature of the original module where the identity has been introduced. We may show that typing in M^ω implies typing in a refined system with bounds, which in turn ensures that the type of a *Val* field is always a supertype of the bound of its identity tag. Details can be found in [2, **??**]. □

4 REBUILDING SOURCE SIGNATURES

In order to also enable the use M^ω just as an internal representation of the typechecker, we need to output signatures in the OCaml syntax, whether typechecking succeeds (to print module interfaces) or fails (to print error messages). Hence, we provide a reverse translation from M^ω signatures back into the source OCaml syntax, called *anchoring*. This translation is necessarily incomplete as some inferred signatures cannot be expressed in the (less expressive) source syntax. However, using M^ω signatures, we can precisely describe three different sources of incompleteness, i.e., signature

avoidance, from which we extract three *guidelines*. Violation of one of these guidelines constitute a specific kind of (anchoring) typechecking errors. We argue that these guidelines lead to more understandable signature avoidance error messages.

Anchoring relies on identity tags, and therefore assumes that we have tagged source programs as described in §3.6 prior to type checking. That is, anchoring translates *tagged* signatures back into source (hence untagged) signatures.

4.1 The Expressiveness Gaps of the Source Syntax

4.1.1 Abstract Type Fields. A first key insight is the difference in the source syntax between the declaration of a *concrete* type ($\texttt{type}\, t = \mathsf{u}$) and that of an *abstract* type ($\texttt{type}\, t = A.t$). An abstract type declaration $\texttt{type}\, t = A.t$ in covariant position effectively *creates* a new abstract type (introducing an existential quantifier in M^ω) and *adds* a type field t to the signature, while a concrete type definition $\texttt{type}\, t = \mathsf{u}$ in covariant position only introduces structural information—adding a field t to refer to the existing type u. By contrast, M^ω signatures *separate* the introduction of new abstract types from the introduction of fields by using explicit quantifiers. In particular, they may mention an abstract type without having a type declaration to refer to it.

Guideline 1. Source signatures can only express situations where the first occurrence of any abstract type α is in a type declaration type $t = \alpha$, called the *anchoring point* for the type α.

4.1.2 Module Identities. Source signatures can only express identity sharing via *concrete* signatures ($= P \leq \mathsf{S}$), thus only when all modules sharing the (same) identity of P have a signature that is a subtype of (the signature of) the module at P. This imposes a subtyping order on the modules sharing the same identity.

Guideline 2. Source signatures can only express identity sharing via concrete signatures. All modules sharing the same identity must have signatures that are supertypes of the first occurrence.

4.1.3 Higher-order Abstract Types. In a source signature, an abstract type t inside an applicative functor F is only reachable via a path with a functor application, as $F(X).t$. This type is therefore restricted to a certain *domain* that corresponds to the parameter signature S of F. If we want to share the type $F(X).t$ with another functor F', the domain S' of F' has to be a subtype of the domain S of F. By contrast, M^ω signatures can express sharing of a higher-order abstract type between functors with arbitrary domains. As an example, let us consider the following M^ω signature, resulting from a projection where the functor that introduced φ became unreachable while two uses of φ remain:

$$\begin{array}{l}\exists\, \varphi\, .\mathsf{module}\ M : \mathsf{sig}\ \ \mathsf{module}\ F\ \ : \forall \overline{\alpha}.\mathcal{C}\ \ \rightarrow \mathsf{sig\ type}\ t = \varphi(\overline{\alpha})\ \mathsf{end} \\ \qquad\qquad\qquad\qquad\quad\ \ \mathsf{module}\ F' : \forall \overline{\alpha}.\mathcal{C}' \rightarrow \mathsf{sig\ type}\ t = \varphi(\overline{\alpha})\ \mathsf{end\ end}\end{array}$$

The source syntax can express the sharing between F and F' only if the domain of the anchoring point (inside F) covers the use inside F', which requires that the domain S' of F' be a subtype of the domain S of F:

$$\begin{array}{l}\texttt{module}\ M : \texttt{sig}\ \ \texttt{module}\ F\ \ : (Y : \mathsf{S}\ ) \rightarrow \texttt{sig}_A\ \ \texttt{type}\, t = A.t\ \texttt{end} \\ \qquad\qquad\qquad\quad\ \ \texttt{module}\ F' : (Y : \mathsf{S}') \rightarrow \texttt{sig}_A\ \ \texttt{type}\, t = F(Y).t\ \texttt{end end}\end{array}$$

Guideline 3. A source signature S can express sharing of an abstract type between several applicative functors only when the first of them (say, F) has an abstract type declaration $\texttt{type}\, t = A.t$ and all following ones have a concrete type declaration of the form $\texttt{type}\, t = F(Y).t$ (and, consequently, a domain included in the domain of F).

Decidability. Guideline 3 for anchoring higher-order abstract types is sound, but too permissive. Indeed, the problem of finding an arbitrary combination of applications of modules in scope to obtain a given type field reduces to a higher-order unification problem, which is undecidable. Hence,

we propose to restrict anchoring further by considering a type declaration type $t = \varphi(\overline{\alpha})$ to be a *suitable* anchoring point only when it occurs inside an applicative functor that is parametric in *exactly* $\overline{\alpha}$. For instance, both anchoring points are suitable in the example above.

Disabling functor applications out of thin air. The decidability heuristic still allows functor applications out of thin air. That is, it may *invent* paths with *new* functor applications that never appeared in the source, just for referring to abstract types that have lost their original path. This would be quite surprising, if not misleading, as it suggests a computation that will never happen.

Following the example above, a type expression of the form $F(X).t$ that is elaborated to $\varphi(\overline{\alpha})$ may have to be anchored in a context where F became unreachable. Should we allow $\varphi(\overline{\alpha})$ to be anchored to a new application $F'(X).t$ where F' might be a totally different functor that just happens to copy the right type field ? To strike the balance between expressiveness and usability, we argue that this should be accepted only when F' is an alias of F—and rejected otherwise.

To distinguish between both cases, we slightly instrument the typing rules[17] by tracking functor applications: when a type expression inside a declaration type $t = \varphi(\overline{\alpha})$ is obtained via a functor application, we mark it as *unsuitable* for anchoring while aliasing a functor keeps its signature unmarked, letting its type declarations available for anchoring.

4.2 The Anchoring Process

For pedagogical purposes, the anchoring algorithm is split in two steps: we first translate M^{ω} signatures into tagged source signatures, before removing tags to obtain source signatures. Both algorithms are presented as relations, although they are deterministic. We first explain some instrumentation added to the typing judgment, including the marking of functors. Then, we highlight the key rules of both steps (the full sets of rules can be found in [2, **??**]).

Instrumenting the typing judgment. First, we extend the grammar of environments with *barriers*: we write $\Gamma \cdot \Gamma'$ for an environment that behaves as Γ, Γ' but with a barrier between Γ and Γ' and let Δ range over environments without barriers. Hence, by writing $\Gamma \cdot \Delta$, we mean that Δ is the part of the environment right after the rightmost mark. This is used to indicate scopes (adding a barrier) and prevent anchoring of types that have been introduced in a larger scope. Marks are introduced in the context by typing rules that open scopes[18]. We also instrument Rule M-Typ-Mod-AppFct to mark skolemization steps, writing $\beta'\langle\overline{\alpha}\rangle$ instead of $\beta'(\overline{\alpha})$ but to mean the same, so that anchoring may pattern-match on list of lists of arguments rather than on a flat list.

Marking higher-order abstract types. Finally, we modify the typing rule for functor application M-Typ-Mod-AppApp to mark higher-order abstract types. To do so, we first use a syntactic mark $\tau^{\dagger}$ on types, which can be seen as the introduction of a postfixed constant $\dagger$ that behaves as $\lambda\alpha.\alpha$. That is, $\tau^{\dagger}$ syntactically differ from τ but really means τ. We write $\mathcal{C}^{\dagger}$ for the signature $\mathcal{C}$ where all type declarations type $t = \tau$ of the structure and substructures have been rewritten into type $t = \tau^{\dagger}$—but the marking does not go inside the body of functors nor inside module types. Therefore, we only change the resulting signature of the rule M-Typ-Mod-AppApp to a marked signature $\mathcal{C}'^{\dagger}[\overline{\alpha} \mapsto \overline{\tau}]$. Marks are kept syntactically but are ignored by typing and subtyping rules.[19]

4.2.1 From M^{ω} Signatures to Tagged Source Signatures. The algorithm proceeds by visiting the M^{ω} signature in left-to-right depth-first order. Along the way, it removes all universal and existential

[17] This information cannot be reconstructed just from types.

[18] More precisely, (1) when entering a generative functor (M-Typ-Mod-GenFct and M-Typ-Sig-GenFct), (2) when entering a module type and (3) when typing the argument of an applicative functor (M-Typ-Mod-AppApp and M-Typ-Sig-AppFct)

[19] Alternatively, we could see marks as the identity type function $\lambda\alpha.\alpha$ and eliminate them by β-equivalence.

quantifiers from the M^ω signature and replaces occurrences of the corresponding abstract types by either a self-reference (at its first occurrence becoming its anchoring point) or a path referring to its anchoring point. An *anchoring map* θ from M^ω types to type expressions with qualified types is built and updated during the visit. The main judgment $\Gamma\,;\,\theta_\Gamma \vdash \mathcal{C} \overset{P}{\hookrightarrow} \mathsf{S} : \theta$ is the anchoring of signatures: given a path P, it translates the M^ω tagged signature $\mathcal{C}$ into a tagged source signature S and produces a (possibly empty) local anchoring map θ of the abstract types anchored in S, prefixed by P. This judgment is also defined for declarations and types. The key rules are those that extend, update, or use the anchoring map to reconstruct source type expressions. They are highlighted below. We refer the reader to [2, **??**] for the full set of rules.

A new anchoring point is introduced when reaching a type declaration of the form type $t = \alpha$ where α is not anchored yet, i.e., not in the domain of θ_Γ. A simplified rule for first-order types is:

$$\frac{\alpha \notin \mathsf{dom}(\theta_\Gamma) \qquad \alpha \in \Delta \qquad \mathsf{args}(\Delta) = \varnothing}{\Gamma \cdot \Delta\,;\,\theta_\Gamma \vdash \mathsf{type}\ t = \alpha \overset{A}{\hookrightarrow} \mathsf{type}\ t = A.t : (\alpha \mapsto A.t)}$$

We ensure that the type α was introduced after the left-most barrier, by requiring $\alpha \in \Delta$, and check that the type declaration has not been made inside an applicative functor by requiring that the environment Δ contains no universally quantified types.[20] We then return the singleton map $(\alpha \mapsto A.t)$. The general version of the rule considers a declaration for a possibly higher-order *unmarked* type expression φ:

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

In particular, this rule only applies when φ is both unmarked and applied to exactly the sequence $\mathsf{args}(\Delta)$ of abstract types in Δ (which necessarily follow φ). Anchoring fails if one of the conditions does not hold. The process could be made more (or less) permissible by tweaking this rule.

The anchoring map θ is *updated* in the two places where access paths to types must be changed as we exit scopes: (1) in Rule A-Sig-StrPath, locally anchored abstract types are made available through the path P and (2) in Rule A-Sig-FctApp, paths to anchored types of θ are point-wise abstracted over the functor parameter Y in the returned map $\lambda Y.\theta$.

A-Sig-StrPath

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

A-Sig-FctApp

$$\frac{\Gamma \cdot \overline{\alpha}\,;\,\theta_\Gamma \vdash \mathcal{C}_a \hookrightarrow \mathsf{S}_a : \theta_a \qquad \mathsf{dom}(\theta) = \overline{\alpha} \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a\,;\,\theta_\Gamma \uplus \theta_a \vdash \mathcal{C} \xrightarrow{A.Val(Y)} \mathsf{S} : \theta}{\Gamma\,;\,\theta_\Gamma \vdash \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \xrightarrow{A.Val} (Y : \mathsf{S}_a) \to \mathsf{S} : \lambda Y.\theta}$$

By contrast, the anchoring map of the body of a generative functor is thrown away, as generative functors cannot appear in paths, and a barrier is added in the premise, as the body cannot capture types defined outside of the functor.

Finally, the anchoring map is used for anchoring an M^ω-types τ into a source one:

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

This rule is designed to allow anchoring of a type τ that is abstract over a certain number of parameters (here, $n - k + 1$) even if τ is actually applied to more parameters (here, n). This comes from the fact that source signatures do not display the depth of enclosing applicative functors. More details are given in [2, **??**]. The resulting type u is the path (resulting from the mathematical

[20] We extract the list of (list of) universally quantified types from the environment using an operator $\mathsf{args}(\Delta)$, which returns a list of lists of variables. Those type variables are easily identified as they immediately precede functor parameters in Δ.

application) $\theta_\Gamma(\varphi)(P_k)\ldots(P_n)$. However, u must be re-typechecked to ensure that paths occurring in τ only contain valid functor applications[21] and that it returns the same type as the input type τ.

4.2.2 Untagging. The first step of anchoring returns a tagged source signature S. It remains to remove the tags, i.e., to return a signature S′ with the `id` and `Val` fields stripped of S, recursively, but expressing the same sharing using concrete signatures. This is defined as a judgment $\Gamma \vdash \mathsf{S} \hookrightarrow \mathsf{S}'$. The two interesting rules are for untagging structural signatures:

$$\textsc{U-Sig-Fresh}\ \frac{\Gamma \vdash \mathsf{S} \hookrightarrow \mathsf{S}'}{\Gamma \vdash \mathsf{Tag}[\mathsf{S}] \hookrightarrow \mathsf{S}'} \qquad \textsc{U-Sig-Con}\ \frac{\Gamma \vdash \mathsf{S} \hookrightarrow \mathsf{S}' \qquad \Gamma \vdash \mathsf{S}' : \mathcal{C}' \qquad \Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathcal{C} \leq \mathcal{C}'}{\Gamma \vdash \mathtt{sig}_A\ \mathtt{type}\ id = P.id\ \mathtt{module}\ Val : \mathsf{S}\ \mathtt{end} \hookrightarrow (= P \leq \mathsf{S}')}$$

When the identity type declaration is an *abstract* type declaration Tag[S] (Rule U-Sig-Fresh), i.e., of the form $\mathtt{sig}_A\ \mathtt{type}\ id = P.id\ \mathtt{module}\ Val : \mathsf{S}\ \mathtt{end}$, the identity of the module is *fresh*, hence the anchored signature of the value is returned directly. Otherwise (Rule U-Sig-Con), the identity type declaration is concrete, i.e., of the form $P.id$; that is, the signature of a module that shares its identity with the module P. We retrieve the M^ω-signature $\mathcal{C}$ of the module P and check that it is a subtype of the M^ω-signature $\mathcal{C}'$ of the untagging S′ of S, so as to ensure that the concrete signature $(= P \leq \mathsf{S}')$ to be returned is valid. The other rules, omitted here, only remove the access to *Val*-fields and inductively call untagging.

4.3 Properties of Anchoring

Anchoring and typechecking of signatures are conceptually inverse of each other, with a few caveats. First, typechecking is not injective: several source signatures can express the same type sharing information. We quotient source signatures by the equivalence induced by M^ω typing:

$$\Gamma \vdash \mathsf{S} \approx \mathsf{S}' \quad \triangleq \quad \Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C} \ \wedge\ \Gamma \vdash \mathsf{S}' : \lambda\overline{\alpha}.\mathcal{C}$$

Second, M^ω signatures can express the fact that they are *inside an applicative functor*, as their abstract types are applied to universally quantified type variables. This does not appear in source signatures, requiring a correspondence up to *skolemization*.

Theorem 4.1 (Anchoring of typed signatures). *Typed source signatures are anchorable[22]:*

$$\Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C} \ \wedge\ \Gamma \hookrightarrow \theta_\Gamma \quad \Longrightarrow \quad \Gamma \cdot \overline{\alpha}\,;\theta_\Gamma \vdash \mathcal{C} \hookrightarrow \mathsf{S}' : (\overline{\alpha} \mapsto \_) \ \wedge\ \Gamma \vdash \mathsf{S} \approx \mathsf{S}'$$

Theorem 4.2 (Anchoring correctness). *Typing back the anchoring gives the original signature, up to re-skolemization of current universally quantified types.*

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

Both results are proved by induction. Finally, untagging is inverse of tagging, as expected. See [2, **??**] for more details.

Theorem 4.3 (Untagging). $\Gamma \vdash \mathsf{S} \hookrightarrow \mathsf{S}' \implies \Gamma \vdash \mathsf{S} \approx [\![\mathsf{S}']\!]$

[21] Indeed, it can happen that a module X is lost, while a transparent ascription `X'` is kept. The types resulting from a functor application `F(X).t` may not be anchorable as `F(X').t` if `X'` lacks certain fields.
[22] We use the judgment $\Gamma \hookrightarrow \theta_\Gamma$ as a *wellformedness* condition to relate Γ and θ_Γ: it is defined as a fold of anchoring over Γ. See the [2, **??**] for more details.

4.4 Discussion

M^ω signatures are more expressive than source signature, but they may also keep *too much information*, revealing the history of the module operations. This may lead to an inferred signature that is not anchorable, while intuitively providing the same type-sharing information as a simpler, anchorable signature. This typically happens when a type variable has become "unreachable", only appearing in sub-expressions. We have identified two such patterns.

Loss of a type argument. The signature[23] $\exists\, \varphi, \alpha.\ \mathcal{C}[\varphi, (\varphi\ \alpha)]$, could be obtained by exporting a functor (providing φ) along with a type obtained by applying this functor to an argument that has latter been hidden. The application $\varphi\ \alpha$ keeps trace that the type was obtained by applying φ to α. However, since the argument α is not accessible, this information became useless. By subtyping, we could safely give the module the simpler signature $\exists\, \varphi, \beta.\ \mathcal{C}[\varphi, \beta]$ cutting the (original) link to the functor. Anchoring the left-hand side will fail since α cannot be anchored, while anchoring the right-hand might succeed. We do not currently allow this simplification during anchoring, since both signatures are not isomorphic in F^ω.

Loss of a type operator. Similarly, the application of a functor may be exported while the functor itself became unreachable. For instance, with two applications of the same functor, we may have a signature of the form $\exists\, \varphi, \alpha, \beta.\ \mathcal{C}[\alpha, \beta, (\varphi\ \alpha), (\varphi\ \beta)]$, which is a subtype of, but not isomorphic to $\exists \alpha, \beta, \alpha', \beta'.\ \mathcal{C}[\alpha, \beta, \alpha', \beta']$.[24]

5 THE FOUNDATIONS: F^ω ELABORATION

The M^ω system is designed to offer a standard, standalone, and expressive approach to the typing of OCaml modules, while hiding the complexity and artifacts of its encoding in F^ω. Yet, the elaboration of module expressions and signatures of M^ω in F^ω, which we now present, served as a basis for the design of M^ω and still shines a new light on its internal mechanisms. It is also used as a proof of type soundness. This elaboration is largely based on the work of Rossberg et al. [23], but differs in a key manner for the treatment of abstract types defined inside applicative functors. A main contribution is the introduction of *transparent* existential types, an intermediate between the standard existential types, called *opaque* existential types, and the absence of abstraction. They bring the treatment of applicative and generative functors closer, and significantly simplify the elaboration.

5.1 F^ω with Kind Polymorphism

We use a variant of explicitly typed F^ω with primitive records (including record concatenation), existential types, and predicative kind polymorphism. While primitive records and existential types are standard, kind polymorphism is less common. Predicativity of kind polymorphism is not needed for type soundness. However, it ensures coherence (of types used as a logic), that is, it prevents typing terms with the empty type $\forall(\alpha : \star).\ \alpha$, whose evaluation would not terminate. For that purpose, kinds are split into two categories: large and small. Polymorphic kinds, which are large, can only be instantiated by small kinds, which in turn do not contain polymorphic kinds. In our setting, kind polymorphism is not essential, as it is only used to internalize the encoding of transparent existential types as F^ω-terms. Alternatively, we could have assumed a family of transparent existential type operators indexed by small kinds, so as to never use large kinds, moving part of the encoding at the meta-level.

[23] Here, we use the notation $\mathcal{C}[\alpha, \dots]$ to indicate that α appears freely in $\mathcal{C}$ and the notation $\mathcal{C}[(\varphi\ \alpha), \dots]$ to indicate that α appears only in the subexpression $(\varphi\ \alpha)$. In particular, $\mathcal{C}[\varphi, (\varphi\ \alpha)]$ means that α only appears as an argument of φ in $\mathcal{C}$.
[24] Actually, if the functor were called only once, the signature would be of the form $\exists\, \varphi, \alpha.\ \mathcal{C}[\alpha, (\varphi\ \alpha)]$, which in this simpler case is actually isomorphic to $\exists \alpha, \alpha'.\ \mathcal{C}[\alpha, \alpha']$.

$$\varsigma := \star \mid \varkappa \mid \varsigma \to \varsigma \qquad \text{(small kinds)}$$
$$\kappa := \varsigma \mid \forall \varkappa.\kappa \mid \kappa \to \kappa \qquad \text{(large kinds)}$$
$$\tau := \alpha \mid \tau \to \tau \mid \{\overline{\ell : \tau}\} \mid \forall(\alpha : \kappa).\tau \mid \exists^{\blacktriangledown}(\alpha : \kappa).\tau \mid \lambda(\alpha : \kappa).\tau \mid \tau\,\tau \mid \forall \varkappa.\tau \mid \Lambda\varkappa.\tau \mid \tau\,\varsigma \mid () \qquad \text{(types)}$$
$$e := x \mid \lambda(x : \tau).e \mid e\,e \mid \Lambda(\alpha : \kappa).e \mid e\,\tau \mid \Lambda\varkappa.e \mid e\,\varsigma \mid e \mathbin{@} e \mid \{\overline{\ell = e}\} \mid e.\ell$$
$$\mid \mathsf{pack}\,\langle \tau, e\rangle \,\mathsf{as}\, \exists^{\blacktriangledown}(\alpha : \kappa).\tau \mid \mathsf{unpack}\,\langle \alpha, x\rangle = e \,\mathsf{in}\, e \mid () \qquad \text{(terms)}$$
$$\Gamma := \cdot \mid \Gamma, \varkappa \mid \Gamma, \alpha : \kappa \mid \Gamma, x : \tau \qquad \text{(environments)}$$

Fig. 9. Syntax of F^ω

The syntax of F^ω is given in Figure 9. Typing rules are standard and available in [2, **??**]. Type equivalence, defined by $\beta\eta$-conversion and reordering of record fields, is also standard and omitted. We use letters τ and e to range over types and expressions to distinguish them from types u and expressions e of the core language, even though these should actually be seen as a subset of τ and e. We consider F^ω to be explicitly typed and explicitly kinded. As a convention, we use a wildcard "_" when a type annotation is unambiguously determined by an immediate subexpression and may be omitted. This is just a syntactic convenience to avoid redundant type information and improve readability, but the underlying terms should always be understood as explicitly-typed F^ω terms. We write $\varkappa$ for kind variables, α and β for type variables of any kind, and φ and ψ for type variables known to be of higher-order kinds. Application of expressions $e\,\varsigma$ and types $\tau\,\varsigma$ to kinds are restricted to small kinds ς. In expressions and type expressions, we actually write kinds κ (and kind abstraction $\Lambda\varkappa.$) in pale color so that they are nonintrusive, and we often leave them implicit. We actually always do so in the elaboration typing rules below for conciseness.

For convenience, we use n-ary notations for homogeneous sequences of type-binders. We introduce let-binding $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ as syntactic sugar for $(\lambda(x : \_).e_2)\,e_1$; we define n-ary pack and unpack operators defined as follows:

$$\begin{array}{ll} \mathsf{pack}\,\langle \tau\bar{\tau}, e\rangle \,\mathsf{as}\, \exists^{\blacktriangledown}\alpha\bar{\alpha}.\sigma & \triangleq \mathsf{pack}\langle \tau, \mathsf{pack}\,\langle \bar{\tau}, e\rangle \,\mathsf{as}\, \exists^{\blacktriangledown}\bar{\alpha}.\sigma[\alpha \mapsto \tau]\rangle \,\mathsf{as}\, \exists^{\blacktriangledown}\alpha\bar{\alpha}.\sigma \\ \mathsf{unpack}\,\langle \alpha\bar{\alpha}, x\rangle = e_1 \,\mathsf{in}\, e_2 & \triangleq \mathsf{unpack}\,\langle \alpha, x\rangle = e_1 \,\mathsf{in}\, \mathsf{unpack}\,\langle \bar{\alpha}, x\rangle = x \,\mathsf{in}\, e_2 \\ \mathsf{pack}\,\langle \varnothing, e\rangle \,\mathsf{as}\, \sigma \triangleq e & \qquad\qquad \mathsf{unpack}\,\langle \varnothing, x\rangle = e_1 \,\mathsf{in}\, e_2 \triangleq \mathsf{let}\ x = e \,\mathsf{in}\, e_2 \end{array}$$

5.2 Encoding of Signatures

M^ω signatures are actually F^ω types with some syntactic sugar. In F^ω, we see Y and A_I as term variables, similar to x's. We assume a collection ℓ_I of record labels indexed by identifiers I of the source language. Structural signatures $\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}$ are just syntactic sugar for record types $\{\overline{\mathcal{D}}\}$. A small trick is needed to represent type fields, which have no computational content, but cannot be erased during elaboration as they carry additional typing constraints. We reuse the solution of *F-ing*, encoding them as identity functions with type annotations. For this, we introduce the following syntactic sugar for the term representing a type field (on the left). We overload the notation to also mean its type (on the right).

$$\langle\!\langle \tau : \kappa \rangle\!\rangle \triangleq \Lambda(\varphi : \kappa \to \star).\lambda(x : \varphi\,\tau).x \quad \text{(Term)} \qquad \langle\!\langle \tau : \kappa \rangle\!\rangle \triangleq \forall(\varphi : \kappa \to \star).\varphi\,\tau \to \varphi\,\tau \quad \text{(Type)}$$

The type τ is used as argument of a higher-kinded type operator φ to uniformly handle the encoding of types of any kind. The key (and only useful) property is that two types (of the same kind) are equal if and only if their encodings are equal. Finally, declarations are syntactic sugar for record entries (distinguished by the category of the identifier):

$$\begin{array}{llll} \mathsf{val}\ x : \tau & \triangleq & \ell_x : \tau & \qquad \mathsf{module}\ X : \mathcal{C} \quad \triangleq \quad \ell_X : \mathcal{C} \\ \mathsf{type}\ t = \tau & \triangleq & \ell_t : \langle\!\langle \tau \rangle\!\rangle & \qquad \mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C} \quad \triangleq \quad \ell_T : \langle\!\langle \lambda\overline{\alpha}.\mathcal{C} \rangle\!\rangle \end{array}$$

5.3 Sharing Existential Types by Repacking

The encoding of module expressions as F^ω terms is slightly more involved than for signatures. Although structures and functors are simply encoded as records and functions, a difficulty arises from the need to *lift* existential types to extend their scope, as explained in §3.3.

Let us first consider the easier generative case. The only construct for handling a term with an abstract type is *unpack*, which allows using the term in a *subexpression*, hence with a limited scope, but not to make an abstract type accessible to the *rest of the program*. Yet, abstract type declarations inside modules have an *open* scope and are visible in the rest of the program. At a technical level, the difficulty comes from the representation of structures. To model them, one needs ordered records (also known as telescopes), where each component can introduce new abstract types accessible to the rest of the record, while standard F^ω only provides non-dependent records.

This observation was at the core of the design of *open existential types* [18] and of *recursive type generativity* [6]. Here, in order to stay in plain F^ω, we reuse and adapt the trick of *F-ing*: structures are built field by field with a special *repacking* pattern: abstract types are unpacked, shared, but abstractly, with the rest of the structure, and then repacked. This allows the terms to mimic the existential lifting done in the types.

To capture this lifting of existentials out of records, we first introduce a combined syntactic form $\mathsf{repack}^\blacktriangledown\,\langle\bar\alpha, x\rangle = e_1 \text{ in } e_2$, which allows the abstract types of e_1 to appear in the type of e_2[25]:

$$\mathsf{repack}^\blacktriangledown\,\langle\bar\alpha, x\rangle = e_1 \text{ in } e_2 \;\triangleq\; \mathsf{unpack}\,\langle\bar\alpha, x\rangle = e_1 \text{ in } \mathsf{pack}\,\langle\bar\alpha, e_2\rangle \text{ as } \exists^\blacktriangledown\bar\alpha.\_$$

Then, we use it to define a new construct to concatenate two records e_1 and e_2 with disjoint domains, but where e_2 might access the first record, via the bound name x_1, and reuse its abstract types, via the bound variables $\overline{\alpha}$:

$$\mathsf{lift}^\blacktriangledown\,\langle\bar\alpha, x_1 = e_1 \mathbin{@} e_2\rangle \;\triangleq\; \mathsf{repack}^\blacktriangledown\,\langle\bar\alpha, x_1\rangle = e_1 \text{ in } \mathsf{repack}\,\left\langle\bar\beta, x_2\right\rangle = e_2 \text{ in } x_1 \mathbin{@} x_2$$

It is better understood by the following derived typing rule and its use in the example of [2, ??].

$$\frac{\Gamma \vdash e_1 : \exists^\blacktriangledown\bar\alpha.\ \{\overline{\ell_1 : \tau_1}\} \qquad \Gamma, \bar\alpha, x_1 : \{\overline{\ell_1 : \tau_1}\} \vdash e_2 : \exists^\blacktriangledown\bar\beta.\ \{\overline{\ell_2 : \tau_2}\} \qquad \overline{\ell_1} \# \overline{\ell_2}}{\Gamma \vdash \mathsf{lift}\langle\bar\alpha, x_1 = e_1 \mathbin{@} e_2\rangle : \exists^\blacktriangledown\bar\alpha, \bar\beta.\ \{\overline{\ell_1 : \tau_1}.\overline{\ell_2 : \tau_2}\}}$$

5.4 Transparent Existential Types and Their Lifting Through Function Types

The repacking pattern allows lifting existential types outside of record types. Unfortunately, this is insufficient for the applicative case, which uses skolemization to further lift abstract types out of the functor body to the front of the functor. This lifting of existential types though universal quantifiers by skolemization and through arrow types, as done in M^ω, is not definable in F^ω.

One solution is to avoid skolemization by *a-priori abstraction* over all possible type and term variables, i.e., the whole typing context. Doing so, existential types are always introduced at the front and need not be skolemized. This is the solution followed by the authors of *F-ing* and by Shan [26]. While this suffices to prove soundness, the encoding is impractical for manual use of the pattern—as it requires frequently abstracting over the whole environment—and therefore does not provide a good intuition of what modules really are. The encoding could be slightly improved by abstracting over fewer variables, without really solving the problem of a-priori abstraction.

We instead retain skolemization, following the intuition of the M^ω system, but we tweak the definition of existential types to make their lifting though universal types definable. Namely, we introduce *transparent existential types*, written $\exists^{\nabla\tau}(\alpha : \kappa).\,\sigma$ to described types that behave as usual existentials $\exists^\blacktriangledown(\alpha : \kappa).\,\sigma$ but remembering the witness type τ for the abstract type α.

[25] We leave the type implicit since the type of repacking is fully determined by the combination of $\bar\alpha$ and the type of e_2

We create a transparent existential type with the expression pack e as $\exists^{\nabla\tau}(\alpha:\kappa).\sigma$, which behaves much as pack $\langle\tau, e\rangle$ as $\exists^{\blacktriangledown}(\alpha:\kappa).\sigma$, except that the witness type τ remains visible in the result type. A transparent existential type is thus weaker than a usual abstract type, as we still see the witness type. It is still abstract, as α cannot be turned back into its witness type τ and has to be treated abstractly. Two transparent existential types with different witnesses are incompatible. This could be seen as a weakness of transparent existentials, but it is actually a key to their lifting through arrow types.

Transparent existential types do not replace usual existential types, which we here call *opaque* existential types, but come in addition to them. Indeed, an expression of a transparent existential type can be further abstracted to become opaque, using the expression seal e, which behaves as the identity but turns the expression e of type $\exists^{\nabla\tau}(\alpha:\kappa).\sigma$ into one of type $\exists^{\blacktriangledown}(\alpha:\kappa).\sigma$.

Transparent existential types may also be used abstractly, with the expression $\mathsf{repack}^{\nabla}\,\langle\alpha, x\rangle = e_1$ in e_2, which is the analog of the expression $\mathsf{repack}^{\blacktriangledown}\,\langle\alpha, x\rangle = e_1$ in e_2 but when e_1 is a transparent existential type $\exists^{\nabla\tau}(\alpha:\kappa).\sigma_1$. In both cases, e_2 is typed in a context extended with the abstract types $\overline{\alpha}$ and a variable x of type σ_1. Crucially, e_2 cannot see the witnesses $\overline{\tau}$. However, the abstract type variables $\overline{\alpha}$ may still appear in the type σ_2 of the expression e_2, and therefore it is made transparent again in the result type of $\mathsf{repack}^{\nabla}\,\langle\alpha, x\rangle = e_1$ in e_2, which is $\exists^{\nabla\tau}(\alpha:\kappa).\sigma_2$. We do not need a primitive transparent version $\mathsf{unpack}^{\nabla}\,\langle\alpha, x\rangle = e_1$ in e_2, since it can be defined as syntactic sugar for unpack $\langle\alpha, x\rangle$ = seal e_1 in e_2.

So far, one may wonder what is the advantage of transparent existentials by comparison with opaque existentials. We provide two key additional constructs for lifting transparent existentials across arrow types and universal types—the only reason to have introduced them in the first place. The lifting across an arrow type, written $\mathsf{lift}^{\rightarrow}e$, turns an expression of type $\sigma_1 \to \exists^{\nabla\tau}(\alpha:\kappa).\sigma_2$ into one of type $\exists^{\nabla\tau}(\alpha:\kappa).(\sigma_1 \to \sigma_2)$ as long as α is fresh for σ_1. Since we can observe the witness τ, we can ensure that the choice of the witness does not depend on the value (of type σ_1), allowing us to lift it outside of the function. While this operation seems easy, it crucially depends on existential types begin transparent—this transformation would be unsound with opaque existentials. For instance, let us consider the following expression: $\lambda x.$ if x then pack $\langle\mathsf{int}, 42\rangle$ as $\exists^{\blacktriangledown}\alpha.\alpha$ else pack $\langle\mathsf{float}, 0.5\rangle$ as $\exists^{\blacktriangledown}\alpha.\alpha$ It has type $\mathtt{bool} \to \exists^{\blacktriangledown}\alpha.\alpha$, but it would be unsound to consider it at the type $\exists^{\blacktriangledown}\alpha.\mathtt{bool} \to \alpha$.

Similarly, lifting across a universal type variable β of kind κ', written $\mathsf{lift}^{\forall}e$, turns an expression of type $\Lambda(\beta:\kappa').\exists^{\nabla\tau}(\alpha:\kappa).\sigma$ into one of type $\exists^{\nabla\lambda(\beta:\kappa').\tau}(\alpha':\kappa' \to \kappa).\forall(\beta:\kappa').\sigma[\alpha \mapsto \alpha'\,\beta]$, provided β is fresh for τ, using skolemization of both the existential variable α and its witness type τ. To summarize, we have extended the syntax of F^{ω} as follows:

$$\begin{array}{rcl} \tau & ::= & \ldots \mid \exists^{\nabla\tau}(\alpha:\kappa).\sigma \\ e & ::= & \ldots \mid \text{pack } e \text{ as } \exists^{\nabla\tau}(\alpha:\kappa).\sigma \mid \text{seal } e \mid \mathsf{repack}^{\nabla}\,\langle\alpha, x\rangle = e_1 \text{ in } e_2 \mid \mathsf{lift}^{\rightarrow}e \mid \mathsf{lift}^{\forall}e \end{array}$$

Their typing rules are given in [2, **??**]. These constructs have no additional computational content, namely $\mathsf{repack}^{\nabla}\,\langle\alpha, x\rangle = \tau$ in σ behaves as a let-binding, while the other constructs behave as e. We add syntactic sugar for n-ary versions of transparent packing and repacking, as we did for opaque existentials. We write seal^n for n applications of seal.

We can define a lifting operation $\mathsf{lift}^{\nabla}\langle\bar{\alpha}, x_1 = e_1 \mathbin{@} e_2\rangle$ for dependent record concatenation as the counterpart of the opaque version, by replacing opaque repacking by transparent repacking. Finally, we also define a new operation $\mathsf{lift}^{*}e$ that uses a combination of the primitive $\mathsf{lift}^{\rightarrow}$ and $\mathsf{lift}^{\forall}$ to turn an expression e of type $\forall\bar{\alpha}.\sigma_1 \to \exists^{\nabla\overline{\tau}}(\bar{\beta}).\sigma_2$ into one of type $\exists^{\nabla\overline{\lambda\bar{\alpha}.\tau}}(\bar{\beta}').\forall\alpha.\sigma_1 \to \sigma_2\left[\bar{\beta} \mapsto \overline{\beta'\,\bar{\alpha}}\right]$, which is the key transformation for lifting existentials out of applicative functor bodies. Its implementation is obvious and given in [2, **??**].

$$
\begin{array}{rl}
 & \exists^{\blacktriangledown}(\mathbb{E} : \forall\varkappa.\,\varkappa \to (\varkappa \to \star) \to \star). \\
\tau_{\mathbb{E}} \triangleq & \left\{\begin{array}{ll}
\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 \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). \\
 & \qquad (\forall(\beta:\omega).\mathbb{E}\,\varkappa\,(\alpha\,\beta)\;(\varphi\,\beta))\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).\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\}
\end{array}
$$

$\tau_0 \triangleq \Lambda\varkappa.\lambda(\alpha:\varkappa).\lambda(\varphi:\varkappa\to\star).\varphi\,\alpha$ $\qquad$ $e_{\mathbb{E}} \triangleq \mathsf{pack}\,\langle\tau_0, e_0\rangle\;\mathsf{as}\;\tau_{\mathbb{E}}$

Fig. 10. Implementation of transparent existentials as a library in F^{ω} with (predicative) kind polymorphism. Notably, the type operator $\mathbb{E}$ has a polymorphic kind $\forall\varkappa.\varkappa\to(\varkappa\to\star)\to\star$

5.5 Implementation of Transparent Existential Types in F^{ω}

Interestingly, transparent existential types are completely definable in plain F^{ω} (with kind polymorphism). A concrete implementation is given on Figure 10. The implementation e_0 is not itself of much interest: most expressions are η-expansions of the identity. However, using regular F^{ω} existentials, e_0 can be abstracted into $e_{\mathbb{E}} = \mathsf{pack}\,\langle\tau_0, e_0\rangle\;\mathsf{as}\;\tau_{\mathbb{E}}$ where τ_0 is the interface type that hides the implementation of the type $\mathbb{E}$. Using this definition, we may see a program e using transparent existential types as a program $\mathsf{unpack}\,\langle\mathbb{E}, x_{\mathbb{E}}\rangle = e_{\mathbb{E}}\;\mathsf{in}\;e$ in plain F^{ω}, with the following additional syntactic sugar[26]:

$$
\begin{array}{rcl@{\qquad}rcl}
\exists^{\triangledown\tau}(\beta:\kappa).\sigma & \triangleq & \mathbb{E}\,\kappa\,\tau\,(\lambda(\beta:\kappa).\sigma) & \mathsf{seal}\;e & \triangleq & x_{\mathbb{E}}.\mathsf{Seal}\;\_\;\_\;e \\
\mathsf{pack}\;e\;\mathsf{as}\;\exists^{\triangledown\tau}(\alpha).\sigma & \triangleq & x_{\mathbb{E}}.\mathsf{Pack}\;\tau\;(\lambda(\alpha:\_).\sigma)\;e & \mathsf{lift}^{\to}e & \triangleq & x_{\mathbb{E}}.\mathsf{Lift}^{\to}\;\_\;\_\;e \\
\mathsf{repack}^{\triangledown}\,\langle\alpha,x\rangle = e_1\;\mathsf{in}\;e_2 & \triangleq & x_{\mathbb{E}}.\mathsf{Repack}\;\_\;\_\;\_\;(\Lambda(\alpha:\_).\lambda(x:\alpha).e_2) & \mathsf{lift}^{\forall}\;e & \triangleq & x_{\mathbb{E}}.\mathsf{Lift}^{\forall}\;\_\;\_\;e
\end{array}
$$

We also write $\mathsf{repack}^{\diamond}\,\langle\alpha,x\rangle = e_1\;\mathsf{in}\;e_2$ and $\mathsf{lift}^{\diamond}\,\langle\bar{\alpha}, x_1 = e_1\;@\;e_2\rangle$ where $\diamond$ stands for either $\triangledown$ or $\blacktriangledown$.

5.6 Elaboration Judgments

As for M^{ω}, the elaboration relies on a subtyping judgment and a typing judgment for both signatures and modules. However, as M^{ω} signatures are already F^{ω} types, we can reuse the M^{ω} typing judgment (although we should now reread it with implicit kinds). Specifically, neither M^{ω} signatures nor its typing contexts mention transparent existential types. This is a key observation: transparent existential types may only appear in types of module expressions. This means that values of such types are never bound to a variable (during elaboration), which would otherwise force them to appear in the typing context. Instead, transparent existential types are always lifted to the top of the expression (using the three lift operations).

There are two main elaboration judgments, for subtyping and typing.

Subtyping. The judgment $\Gamma \vdash \mathcal{C} \prec \mathcal{C}' \rightsquigarrow f$ extends M^{ω} subtyping to return an explicit coercion function f. The judgment is also defined for declarations $\Gamma \vdash \mathcal{D} \prec \mathcal{D}' \rightsquigarrow f$. Interestingly, as signatures do not contain transparent existential types, subtyping between signatures is (a subcase of) standard subtyping in F^{ω}. As they are similar to M^{ω} subtyping, we omit the rules and refer the

[26] As above _ stands for kinds or types that are left implicit as they can be straightforwardly inferred from other arguments. We also extend transparent existentials with sequences of abstractions as we did for opaque existentials.

reader to [2, ??]. The judgments satisfy the following properties regarding F^ω typing:

$$\Gamma \vdash \mathcal{C} \prec \mathcal{C}' \rightsquigarrow f \implies \Gamma \vdash f : \mathcal{C} \to \mathcal{C}' \qquad \Gamma \vdash \mathcal{D} \prec \mathcal{D}' \rightsquigarrow f \implies \Gamma \vdash f : \{\mathcal{D}\} \to \{\mathcal{D}'\}$$

Typing. To factor notations for the typing judgment, we introduce the meta-variable ϑ that stands for either an opaque existential $\blacktriangledown$ or a transparent one $\triangledown\tau$ together with1 its witness type τ. We write $\mathrm{mode}(\vartheta)$ (*resp.* $\mathrm{mode}(\bar\vartheta)$) for the mode of ϑ (*resp.* the homogeneous sequence $\bar\vartheta$), which is either $\triangledown$ or $\blacktriangledown$. When a mode is expected without a witness type, we may leave the projection implicit and just write $\bar\vartheta$ instead of $\mathrm{mode}(\bar\vartheta)$. The convention is the same as for the M^ω system.

The judgment $\Gamma \vdash^\diamond \mathtt{M} : \exists^{\bar\vartheta}\bar\alpha.\mathcal{C} \rightsquigarrow e$ extends M^ω typing with the elaborated module term e. The mode $\diamond$ must coincide with $\bar\vartheta$, and may be left implicit, as we did for the corresponding M^ω judgment. Hence, we usually just write $\Gamma \vdash \mathtt{M} : \exists^{\bar\vartheta}\bar\alpha.\mathcal{C} \rightsquigarrow e$. A similar, helper judgment $\Gamma \vdash_A^\diamond \mathtt{B} : \exists^{\bar\vartheta}\bar\alpha.\mathcal{D} \rightsquigarrow e$ is also defined for bindings. The properties of both judgments are detailed below. When reading an M^ω type environment Γ in F^ω, we must read $A.(\mathsf{val}\ x : \tau)$ and $A.(\mathsf{module}\ X : \mathcal{C})$ as $A_x : \tau$ and $A_X : \mathcal{C}$, and drop $A.\mathcal{D}$ when $\mathcal{D}$ binds a type or a signature.

Theorem 5.1 (Soundness). *When typing a module, the elaborated module term is well typed regarding F^ω typing, and the source module term is well typed regarding M^ω typing.*

$$\begin{array}{rcl} \Gamma \vdash \mathtt{M} : \exists^{\bar\vartheta}\bar\alpha.\mathcal{C} \rightsquigarrow e & \implies & \Gamma \vdash e : \exists^{\bar\vartheta}\bar\alpha.\mathcal{C} \quad \wedge\ \Gamma \vdash \mathtt{M} : \exists^{\bar\vartheta}\bar\alpha.\mathcal{C} \\ \Gamma \vdash \overline{\mathtt{B}} : \exists^{\bar\vartheta}\bar\alpha.\overline{\mathcal{D}} \rightsquigarrow e & \implies & \Gamma \vdash e : \exists^{\bar\vartheta}\bar\alpha.\{\overline{\mathcal{D}}\} \ \wedge\ \Gamma \vdash \mathtt{M} : \exists^{\bar\vartheta}\bar\alpha.\overline{\mathtt{D}} \end{array} \tag{1}$$

Theorem 5.2 (Completeness). *Well-typed M^ω terms and bindings can always be elaborated:*

$$\begin{array}{rcl} \Gamma \vdash \mathtt{M} : \exists^\diamond\bar\alpha.\mathcal{C} & \implies & \exists e, \bar\vartheta,\ \Gamma \vdash \mathtt{M} : \exists^{\bar\vartheta}\bar\alpha.\mathcal{C} \rightsquigarrow e \ \wedge\ \mathit{mode}(\bar\vartheta) = \diamond \\ \Gamma \vdash \overline{\mathtt{B}} : \exists^\diamond\bar\alpha.\overline{\mathcal{D}} & \implies & \exists e, \bar\vartheta,\ \Gamma \vdash \overline{\mathtt{B}} : \exists^{\bar\vartheta}\bar\alpha.\overline{\mathcal{D}} \rightsquigarrow e \ \wedge\ \mathit{mode}(\bar\vartheta) = \diamond \end{array} \tag{2}$$

Proof Sketch. Soundness is by induction on the typing derivation. Completeness can be easily established as the elaboration rules mimic the M^ω typing rules with no additional constraints on the premises, except for transparent existentials. However, these only appear on the types of elaborated modules as a positive information, which is never restrictive. In particular, a transparent existential type is always used abstractly and pushed in the context after dropping the witness type exactly as an opaque existential type, i.e., as in M^ω. □

5.7 Elaborated Typing Rules

We only present an excerpt of the most significant elaboration rules for expressions. The full set of elaboration rules is given in [2, ??]. The key rule for structures is the sequence rule that combines bindings. It may be concisely written as follows for generative and applicative modes:

E-Typ-Bind-SeqGen

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

E-Typ-Bind-SeqApp

$$\frac{\Gamma \vdash_A \mathtt{B} : \exists^{\triangledown\overline{\tau}_1}(\overline{\alpha}_1).\mathcal{D} \rightsquigarrow e_1 \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D} \vdash_A \overline{\mathtt{B}} : \exists^{\triangledown\overline{\tau}_2}(\overline{\alpha}_2).\overline{\mathcal{D}} \rightsquigarrow e_2}{\Gamma \vdash_A \mathtt{B}; \overline{\mathtt{B}} : \exists^{\triangledown\overline{\tau}_1\overline{\tau}_2}(\overline{\alpha}_1\overline{\alpha}_2).(\mathcal{D}, \overline{\mathcal{D}}) \rightsquigarrow \mathsf{lift}^{\triangledown}\langle \overline{\alpha}_1, x_1 = e_1 \ @\ (\mathsf{let}\ A_{I_1} = x_1.\ell_{I_1}\ \mathsf{in}\ e_2)\rangle}$$

The single field of e_1 is concatenated with the fields of e_2 after lifting out their existential bindings. In both cases, the field of e_1 is made visible in e_2, as well as the existentials in front of e_1—but abstractly. Interestingly, the generative and applicative versions can be factored as follows:

$$\frac{\Gamma \vdash_A \mathtt{B} : \exists^{\bar\vartheta_1}\overline{\alpha}_1.\mathcal{D} \rightsquigarrow e_1 \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D} \vdash_A \overline{\mathtt{B}} : \exists^{\bar\vartheta_2}\overline{\alpha}_2.\overline{\mathcal{D}} \rightsquigarrow e_2 \qquad \diamond = \mathrm{mode}(\bar\vartheta_1\bar\vartheta_2)}{\Gamma \vdash_A \mathtt{B}, \overline{\mathtt{B}} : \exists^{\bar\vartheta_1\bar\vartheta_2}\overline{\alpha}_1\overline{\alpha}_2.(\mathcal{D}, \overline{\mathcal{D}}) \rightsquigarrow \mathsf{lift}^{\diamond}\langle \overline{\alpha}_1, x_1 = e_1 \ @\ (\mathsf{let}\ A_{I_1} = x_1.\ell_{I_1}\ \mathsf{in}\ e_2)\rangle} \quad \text{(E-Typ-Bind-Seq)}$$

We also have a unified rule for typing structures in both modes:

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

By default, elaboration is done in applicative mode, hence inferring transparent existentials, but it can be turned into generative mode when required, using Rule E-Typ-Mod-Seal. Since signature ascription is defined on paths, it is applicative (rule E-Typ-Sig-App). That is, signature ascription $(P : \mathsf{S})$ may introduce new abstract types $\overline{\alpha}$ as prescribed by the (elaboration $\lambda\overline{\alpha}.\mathcal{C}$ of the) signature S, but these are transparent existentials in the type of $(P : \mathsf{S})$.

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-Sig-App
$$\frac{\Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C} \qquad \Gamma \vdash P : \mathcal{C}' \rightsquigarrow e \qquad \Gamma \vdash \mathcal{C}' \prec \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow f}{\Gamma \vdash (P : \mathsf{S}) : \exists^{\nabla\overline{\tau}}(\overline{\alpha}).\mathcal{C} \rightsquigarrow \mathsf{pack}\ f\ e\ \mathsf{as}\ \exists^{\nabla\overline{\tau}}(\overline{\alpha}).\mathcal{C}}$$

Elaboration of functors. At first glance, the elaboration of functors seems to differ more significantly in the applicative and generative cases:

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

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

The body of an applicative functor is elaborated to transparent existentials which are lifted through λ's, while in the generative case, the existentials are opaque and cannot be lifted. However, this difference is largely artificial as a result of using a special argument $()$ to enforce generativity. Otherwise, the main difference lies in enforcing the body of the functor to be typed in generative mode, hence with an opaque existential type. Since lift^* is neutral on terms that do not have transparent existential types, the elaboration of the generative case could also be written $\mathsf{lift}^*\lambda(\_ : ()).e$, so that the two cases only differ by the modes of elaboration of their bodies.

6 DISCUSSION

6.1 Related Works

The literature regarding ML modules is rich and varied. The link between abstract types in ML module systems and existential types in F^ω was initially explored by Mitchell and Plotkin [17]. This vision was opposed by MacQueen [15] who considered existential types to be too weak and proposed using a restriction of dependent types (strong sums) to describe module systems. Further work on phase separation by Harper et al. [11] supported the idea that dependent types may actually be too powerful for module systems. SML modules were described by Harper et al. [11]. Two approaches for the formalization and improvement of abstract types in SML were later independently yet simultaneously described by Leroy [12] using manifest types and Harper and Lillibridge [10] via an adapted F^ω with translucent sums. The genesis of the OCaml module system was specified by Leroy [12, 14] with, later, an extension to applicative functors [13].

The key idea for a simplified link between modules and F^ω, developed by Russo [25], was to use existential types to interpret signatures. Pursing a related objective, Dreyer [6] proposed to model generativity using stamps instead of existential types, while Montagu and Rémy [19] proposed a similar, but logically-based approach, through the concept of open existential types.

Pushing Russo's idea further, a milestone was achieved by Rossberg et al. [23] with the elaboration of an expressive module system into F^ω, dubbed the *F-ing* approach. *F-ing* gives a *syntactic* translation from the syntax directly into F^ω, thus providing semantics by elaboration. *F-ing* is

safe by construction[27], inheriting the property from F^ω, but requires the programmer to think in terms of the elaboration, which is quite involved in some cases, and only sees the elaborated types instead of the usual signatures. Our anchoring algorithm removes the need for the user to know the underlying F^ω encoding, except for a deep understanding of the tricky cases of signature avoidance.

Moving one step further, Rossberg [21] achieved a unification of the core and module languages (thus, unstratified), called 1ML, using F^ω as the underlying programming language and seeing module constructs as syntactic sugar. This is appealing, even though the prototype implementation only covered the generative case: the applicative case might have been unusable in practice, due to a priori extrusion of quantifiers over the whole context. Hopefully, this could be fixed by applying our *a posteriori* lifting of transparent-existential-types technique to 1ML. We decided to build on the *F-ing* approach rather than 1ML because the clear separation between core and module languages seems better suited to model a real-world language as OCaml, where the core language has a lot of additional features which are orthogonal to the module system.

More recently, Crary [5] used involved focusing techniques to solve signature avoidance in the singleton-type approach for SML modules in a manner that turns out to have many similarities with *F-ing*. Our work provides complementary information on the understanding of signature avoidance, neither on its origin nor on how to avoid it, which was already well-understood in *F-ing*, but on the difficulties and the principled way to solve it in the path-based approach of OCaml.

6.2 Features not Included

In our formalization, we omitted some features of OCaml both at the module level and at the interface between the module and core languages. We see no difficulty in adding the following features (already covered by [23]): first-class modules which are just values injected in and out of the core language, (S with type $t = \tau$) and (S with type $t := \tau$) which only operate at the level of signatures, (include P), which just flattens structures, and (open P) which just extends the environment. Other main omitted features are discussed below.

Extracting signatures from modules. OCaml features a construct (module type of M). We could easily add such a construct restricted to cases where M does not introduce any abstract type, which includes all paths P, and return the signature $\mathcal{C}$ of M[28]. This would be analogous to the (like M) construct of *F-ing* [23]. However, this does not reflect the OCaml semantics, which returns a signature (in the source language, corresponding to the M^ω signature) $\exists^\diamond\overline{\alpha}.\mathcal{C}$ with abstract type fields $\overline{\alpha}$ whenever the definition of the module at P introduced abstract types $\overline{\alpha}$. This leads to a surprising situation[29] where (module type of P) and (module type of P') may differ when P' is an alias of P and P is a module definition with abstract types.

Recursive modules. This raises both typechecking and compilation issues. Typechecking recursive modules poses the *double-vision* problem explained and solved in [6] and also solved in [18]. A similar solution should also apply to our encoding in F^ω—after adding recursion at both the term and type levels. A more ambitious solution would be to extend M^ω with *mixin* modules altogether [22]. This would not help improve their compilation, though.[30]

[27] Besides, their work has also been mechanized in Coq for the generative case. A Coq formalization of our approach, including the applicative case, would be welcomed. It is left for future work.

[28] To preserve decidability of subtyping, they restricted the feature to their syntactic notion of *explicit* signatures, which we would also need to do.

[29] To circumvent this behavior, users sometimes write module type of (struct include P end) to force the "strengthening" and obtain the concrete signature $\mathcal{C}$ rather then the abstract signature $\exists^\diamond\overline{\alpha}.\mathcal{C}$. This trick does not apply to functors.

[30] Their compilation in a call-by-value setting requires a static analysis to ensure that the recursion is always well-founded so that values will eventually be constructed before being destructed. OCaml uses a simple static analysis together with a *back-patch* semantics, which can fail at runtime.

Abstract module types. Abstract module types are a particularity of OCaml, that does not fit well in our framework. Instantiating an abstract module type A appearing in a signature S by another signature containing abstract types (or abstract module types) amounts in M^ω to introducing new quantifiers deep inside the signature, which must be universal or existential depending on the polarity of the occurrences of A in S. This is not doable with F^ω type instantiation alone. Yet, adding kind-level lambdas to F^ω (with predicative kind polymorphism) could cover a restricted case, where an abstract module type cannot be instantiated by a signature that does itself contain abstract module types. This seems to be sufficient for the cases found in real-world projects.

Richer type declarations. The type declarations mechanism of OCaml is much richer than what we model. Adding parametric type definitions should not raise any problem, as F^ω already features type functions. The various annotations for type parameters (variance, boxing, etc.) should be encodable in a (light) extensions of F^ω with similar features. Algebraic datatypes (ADT) can be represented as an abstract type definition followed by value declarations for the constructors, as mentioned by [12]. Recent work has shown that generalized ADT are encodable in an extension of F^ω [28], which might be usable to extend our support for such type declarations.

6.3 Future Works

We have introduced and formalized M^ω, a middle point between the source path-based module system used in OCaml and F^ω. First, we gave an improved elaboration of modules into F^ω, using the new notion of *transparent existentials* to treat applicative functors in almost the same simple way as generative functors. Then, using M^ω as an intermediate language, we shone a new light on the mechanisms of the OCaml type system, and provided a detailed description of the solvable and unsolvable cases of signature avoidance.

An immediate application of our work is to use M^ω-signatures as an intermediate typing representation for OCaml. We avoided the difficulty of maintaining module type names from the source by inlining them, while a real implementation will definitely need strategies to maintain them. Extending our formalization to do so would be an interesting, but orthogonal contribution.

We are currently faced with the following dilemma: we can present inferred signature to users in the source syntax at the cost of dealing with the signature avoidance problem and explain it to the user. Alternatively, M^ω signatures eliminate this artificial problem altogether but depart from the path-based source notation that has proven user-friendly in many cases. Giving the user access to full M^ω signatures would make subtyping undecidable. Finding a set of *good sense* restrictions to maintain decidability, as well as mixing the path-based and M^ω signatures constitutes an interesting research and engineering topic. Characterizing the *artifacts* described in §4.4 and changing M^ω or the anchoring to remove them in final signatures is also a topic of interest for future works.

The module identities of OCaml are probably abstraction safe[31], as hinted by our Theorem 3.1. Yet, the type-safety of F^ω is not sufficient to *show* abstraction-safety, even with the full tracking of values *à la F-ing* (section 8), which seems obviously abstraction safe. A full semantic model of M^ω types would be needed and constitutes an interesting future work. This would probably benefit from the insights of the recent works of Crary [3] [4] on logical relations and abstraction properties for a rich module calculus.

The introduction of transparent existential types makes the treatment of applicative and generative functors much closer to one another. It could benefit other existing approaches to ML modules which removed sealing inside applicative functors due to the cost of *a-priori* skolemization. Finally, it could be interesting to explore extending F^ω with minimalist constructs so that we may program *with modules* directly.

[31] Assuming that the typechecker could enforce that applicative functors only contain pure values.

REFERENCES

[1] Sandip K. Biswas. 1995. Higher-Order Functors with Transparent Signatures. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 154–163. https://doi.org/10.1145/199448.199478

[2] Clément Blaudeau, Didier Rémy, and Gabriel Radanne. 2024. Fulfilling OCaml modules with transparency (supplementary material). https://doi.org/10.1145/3649818

[3] Karl Crary. 2017. Modules, Abstraction, and Parametric Polymorphism. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 100–113. https://doi.org/10.1145/3009837.3009892 event-place: Paris, France.

[4] Karl Crary. 2019. Fully abstract module compilation. *Proc. ACM Program. Lang.* 3, POPL, Article 10 (jan 2019), 29 pages. https://doi.org/10.1145/3290323

[5] Karl Crary. 2020. A focused solution to the avoidance problem. *Journal of Functional Programming* 30 (2020), e24. https://doi.org/10.1017/S0956796820000222

[6] Derek Dreyer. 2007. Recursive type generativity. *Journal of Functional Programming* 17, 4-5 (2007), 433–471. https://doi.org/10.1017/S0956796807006429

[7] Derek Dreyer, Karl Crary, and Robert Harper. 2003. A type system for higher-order modules. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, Alex Aiken and Greg Morrisett (Eds.). ACM, 236–249. https://doi.org/10.1145/604131.604151

[8] Derek Dreyer, Robert Harper, and Karl Crary. 2005. *Understanding and evolving the ML module system.* Ph. D. Dissertation. USA. AAI3166274.

[9] Jacques Guarrigue and Leo White. 2014. Type-level module aliases: independent and equal *(ML Family/OCaml Users and Developers workshops)*. https://www.math.nagoya-u.ac.jp/~garrigue/papers/modalias.pdf

[10] Robert Harper and Mark Lillibridge. 1994. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 123–137. https://doi.org/10.1145/174675.176927

[11] Robert Harper, John C. Mitchell, and Eugenio Moggi. 1989. Higher-Order Modules and the Phase Distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '90)*. Association for Computing Machinery, New York, NY, USA, 341–354. https://doi.org/10.1145/96709.96744

[12] Xavier Leroy. 1994. Manifest Types, Modules, and Separate Compilation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 109–122. https://doi.org/10.1145/174675.176926

[13] Xavier Leroy. 1995. Applicative functors and fully transparent higher-order modules. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95*. ACM Press, San Francisco, California, United States, 142–153. https://doi.org/10.1145/199448.199476

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

[15] David B. MacQueen. 1986. *Using Dependent Types to Express Modular Structure.* Association for Computing Machinery, New York, NY, USA, 277–286. https://doi.org/10.1145/512644.512670

[16] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: library operating systems for the cloud. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, Vivek Sarkar and Rastislav Bodík (Eds.). ACM, 461–472. https://doi.org/10.1145/2451116.2451167

[17] John C. Mitchell and Gordon D. Plotkin. 1985. Abstract Types Have Existential Types. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '85)*. Association for Computing Machinery, New York, NY, USA, 37–51. https://doi.org/10.1145/318593.318606

[18] Benoît Montagu. 2010. *Programming with first-class modules in a core language with subtyping, singleton kinds and open existential types. (Programmer avec des modules de première classe dans un langage noyau pourvu de sous-typage, sortes singletons et types existentiels ouverts).* PhD Thesis. École Polytechnique, Palaiseau, France. https://tel.archives-ouvertes.fr/tel-00550331

[19] Benoît Montagu and Didier Rémy. 2009. Modeling Abstract Types in Modules with Open Existential Types. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL'09)*. Savannah, GA, USA, 354–365. https://doi.org/10.1145/1480881.1480926

[20] Gabriel Radanne, Thomas Gazagnaire, Anil Madhavapeddy, Jeremy Yallop, Richard Mortier, Hannes Mehnert, Mindy Perston, and David Scott. 2019. Programming Unikernels in the Large via Functor Driven Development. arXiv:1905.02529 [cs.PL]

[21] Andreas Rossberg. 2018. 1ML - Core and modules united. *J. Funct. Program.* 28 (2018), e22. https://doi.org/10.1017/S0956796818000205
[22] Andreas Rossberg and Derek Dreyer. 2013. Mixin'Up the ML Module System. *ACM Trans. Program. Lang. Syst.* 35, 1 (April 2013), 2:1–2:84. https://doi.org/10.1145/2450136.2450137
[23] Andreas Rossberg, Claudio Russo, and Derek Dreyer. 2014. F-ing modules. *Journal of Functional Programming* 24, 5 (Sept. 2014), 529–607. https://doi.org/10.1017/S0956796814000264
[24] Claudio V. Russo. 2000. First-Class Structures for Standard ML. *Nord. J. Comput.* 7, 4 (2000), 348–374.
[25] Claudio V. Russo. 2004. Types for Modules. *Electronic Notes in Theoretical Computer Science* 60 (2004), 3–421. https://doi.org/10.1016/S1571-0661(05)82621-0
[26] Chung-Chieh Shan. 2004. Higher-order modules in System F^{ω} and Haskell. (01 2004).
[27] Zhong Shao. 1999. Transparent Modules with Fully Syntactic Signatures. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999.* ACM, 220–232. https://doi.org/10.1145/317636.317801
[28] Filip Sieczkowski, Sergei Stepanenko, Jonathan Sterling, and Lars Birkedal. 2024. The Essence of Generalized Algebraic Data Types. *Proc. ACM Program. Lang.* 8, POPL, Article 24 (jan 2024), 29 pages. https://doi.org/10.1145/3632866
[29] Leo White, Frédéric Bour, and Jeremy Yallop. 2014. Modular implicits. In *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014.* 22–63. https://doi.org/10.4204/EPTCS.198.2

Received 21-OCT-2023; accepted 2024-02-24