Fulfilling OCaml modules with transparency

CLÉMENT BLAUDEAU and DIDIER RÉMY, Cambiun, INRIA, France
GABRIEL RADANNE, CASH, INRIA, EnsL, UCBL, CNRS, LIP, France

ML modules come as an additional layer on top of the core language that offers large-scale notions of composition and modularity. They have been essential for developing complex applications and largely contributed to the success of OCaml and SML. While modules are easy to write for common cases, their intensive or advance use may become tricky. Additionally, despite a long line of work, 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.

We propose a new comprehensive description of a large subset of OCaml modules, with both applicative and generative functors, and extended with transparent ascription. Building on a previous translation from ML modules to F^ω, we introduce an intermediate system, called M^ω, that mediates between the source language and F^ω, combining the convenient path-based syntactic notation of ML modules with the precise and well-established type-theory of F^ω. From the M^ω system, we elaborate terms into F^ω using the new technique of *transparent existential types*, which ensures type soundness.

ACM Reference Format:
Clément Blaudeau, Didier Rémy, and Gabriel Radanne. 2023. Fulfilling OCaml modules with transparency. 1, 1 (January 2023), 29 pages.

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 components. By channeling interactions through reduced APIs, this makes complexity manageable and increases code sharing. A component may be, for instance, the implementation of a data-structure along with the functions to handle it. Implementation details, such as internal invariants, can be kept hidden from the public interface thanks to language-level mechanisms. A wide variety of techniques can be used to apply modularity concepts to software development: simple compilation units, classes, packages, crates, etc.

In ML, modularity is provided by a *module system*, which forms a separate language layer built on top of the core language. To quote Rossberg [17], effectively, ML is two languages in one. The interactions between modules are controlled statically by a strict type system, making modularity work in practice with little run-time overhead. A module is described by its interface, called a *signature*, which serves both as a light specification and as an API.

The OCaml module system is quite rich and extensively used: all sizable OCaml projects use modules to access libraries or define parametric instances of data structures (sets, hash-tables, streams, etc.); several successful projects have made heavy use of modules, as in MirageOS [12] where modules and functors are assembled on demand using a DSL [16].

However, despite the successes and the interest of the community regarding ML modules, giving them a formal type-theoretic definition and proving its properties, especially type soundness, has proven to be highly technical. In the case of OCaml, the foundational works of Leroy [9, 10], have not been extended to include the numerous new features. The current situation is a module system

Authors' addresses: Clément Blaudeau; Didier Rémy, Cambiun, INRIA, France; Gabriel Radanne, CASH, INRIA, EnsL, UCBL, CNRS, LIP, France.

2023. XXXX-XXXX/2023/1-ART $15.00
https://doi.org/

that is widely used but still unspecified, with typechecking failures or surprising behaviors due to edge-cases or undocumented heuristics.

Besides the academic interest, lacking a formal specification has become problematic when trying to evolve the module system. Adding, modifying, or even fixing a feature of the module system requires a deep knowledge of the technical internals of the typechecker. For more substantial extensions such as transparent ascription or *modular implicits* [24], lacking a specification is a show-stopper, as it could have unforeseen breaking changes.

Combining ideas from many years of research (see §6 for related works), a particularly successful and elegant approach to model ML module systems is the translation of ML-modules into F^ω, the high-order polymorphic lambda calculus. Extending the approach of Russo [20], who provided a type system for modules using F^ω types, a milestone was achieved by Rossberg et al. [18], who gave a full elaboration of both types and terms of a significant subset of SML into F^ω. The elaboration showed that ML modules can be encoded as F^ω terms, which ensure soundness of the system, and correspondingly, ML-signatures can thus be understood as F^ω-types.

We build on the insights of their work, which we adapt and improve for an OCaml-like language (extended with *transparent ascription*). To separate the concerns of specification and soundness, we first present a standalone type system, called M^ω, that produces signatures in an OCaml-like syntax extended with F^ω quantifiers, *à la* Russo [20]. The M^ω system is central to our work: it provides an up-to-date specification of OCaml modules; it was designed to serve as a new internal representation of signatures for the typechecker; it can also be used to reason about the design and issues of module systems, such as the signature avoidance problem.

Signatures of M^ω are nothing else but syntactic sugar over F^ω types. To ensure type soundness of M^ω, we give an elaboration *à la* Rossberg et al. [18]. 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.

Besides its applications to OCaml, the *simplification* of the encoding of modules into F^ω could also help conduct proofs of modular programs—a topic of quickly growing importance. Moreover, the simplification of the encoding into F^ω in the applicative case should also benefit to the unification of the module and core languages proposed by Rossberg [17] and, perhaps with a few helper constructs, let modules be directly programmable into the core language.

Contributions:

- A simple specification of a large subset of OCaml modules in M^ω, including both applicative and generative functors, using ML-style signature syntax but explicit F^ω-style quantifiers.
- The specification of a new form of *transparent ascription* (called *concrete ascription*) to subsume and fix the module aliasing system of OCaml and that is better suited to the reconstruction of source signatures than previous solutions.
- A source-to-source encoding of *aliasing* (a key to *abstraction safety*) relying solely on type abstraction—removing the need for a primitive treatment by the type system.
- An *anchoring* algorithm that translates canonical signatures back into the path-based source signatures with a rigorous approach to the signature avoidance problem.
- 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.
- A simpler encoding of applicative functors based on transparent existential types that reduces the difference between their treatment and the one of generative functors. As a result, a simpler encoding of ML modules in F^ω.

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

Fig. 1. Basic modularity.

Plan. The paper is organized as follows. In §2, we start with an overview of the key features, strengths, and weaknesses of the OCaml module system. In §3, we give a precise, formal specification of M^ω that bridges the gap between source signatures and F^ω types. In **??**, we discuss the signature avoidance problem and define *anchoring*: the backward translation from canonical signatures to source signatures. In §5, we present the elaboration of modules into F^ω terms as an extension of the M^ω typing. To this end, we introduce using transparent existential types and show how they simplify the treatment of applicative functors. Finally, we discuss related and future works in §6 and §7.

2 A MODERN MODULE SYSTEM

We start with a quick introduction of the basic features of ML module systems: structures, signatures, sealing, and functors. We then focus on the difference between generative and applicative functors, and the level of *granularity* that comes with applicative functors. The OCaml choice leads to notion of module level aliases and *module identity*. To improve the current situation of module aliases, we discuss the proposed extension of concrete ascription. Finally, we explain the *signature avoidance* problem. This section constitutes an introduction to the problem space of ML-modules and presents concepts and solutions that are already described in more details in the literature, but helpful to understand the rest of the paper.

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* and can be type declarations (line 2), values (line 3), submodules, or *module types*, also called *signatures* (line 6). Definitions inside a signature are called *declarations*. Type declarations can be left abstract, as the one at line 7.

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 11). 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, modules types can be used to restrict how a given module depends on other modules. This is achieved by turning a 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`

```
1  module Tokens () = (struct
2      type t = int
3      let x = ref 0   ...
4    end : sig type t ... end)
5  module PublicTokens = Tokens()
6  module PrivateTokens = Tokens()
7  (** 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.

```
8  module OrderedSet (E:Ordered) = (struct
9      type t = E.t list
10     let empty : t = []   ...
11   end : sig type t ... end)
12 module S1 = OrderedSet(Integers)
13 module S2 = OrderedSet(Integers)
14 (** 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.

can be applied to modules satisfying `Ring` such as `Complex` and `CRing` (lines 18), but also the output of `Polynomials` itself (line 20). 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.

2.2 Applicative and generative functors

Both modules and functors can be used to either structure the code base or to make reusable components. A program might therefore contain several applications of the same (reusable) functor to the same argument. When such a functor *produces* abstract types, a question arises: should the abstract types be deemed equal, i.e. should the functions and values of one application of the functor be compatible with other applications of the same functor to the same argument? 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.

Calling a generative functor twice *generates* two incompatible modules, with incompatible abstract types. A *generative functor* is typically used when the produced module is stateful, and should therefore not be compatible with other instances. A generative functor can create its own internal state, produce effects, or dynamically choose between several implementations of its abstract types (using *first-class modules*). Generativity can also be used to force incompatibility between otherwise 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 "`()`"[1].

Conversely, an *applicative functor* may be used for pure[2] modules that have static definitions of the internal representations of their abstract types: calling an applicative functor twice with the *same argument* produces compatible pieces of code where values from the first module can be handled by the second one. Technically, this is achieved by having the same functor applications produce the *same*, hence compatible, abstract types. Applicative functors are especially useful when the provides generic functionalities (such as hash-maps[3], sets, lists, etc.) that may appear in several places and yet be compatible. Applicative functors are the default in OCaml.

[1] We expand on the reasons behind this choice in §3.1.
[2] In OCaml, it is left to the user's responsibility to mark impure functors as generative, the typechecker does not track effects.
[3] `Hashtbl.Make` is pure as it does not produce a new hash table itself, even though it contains impure functions.

2.3 Abstraction safety and granularity of applicativity

A key design point is the *granularity* of applicative functors: when are two modules considered as being *the same* when used as argument to an applicative functor, i.e., when should two functor applications produce the same abstract types?

A first option is to consider modules to be *similar* (i.e.,, the same when used as argument) when they have the same type definitions (both names and content). This criterion is called *static equivalence* [4, 19, 22]. It is *type-safe*, as the actual implementation of the abstract types produced by the functor can only depend statically[4] on its parameters , thus only on its *statically known* type fields. Moscow ML uses this static equivalence criterion for its applicative functors.

However, relying only on type fields can make two functor applications produce equal abstract types while they actually have different internal invariants. In the example on Figure 2, it is true that actual implementation of sets `type t = E.t list` only depends on the type fields of the parameter. However, the code maintains a stronger invariant that depends also on value fields of the parameter: the lists are ordered with respect to the comparison function of `E`. The property that type abstraction also protects arbitrary invariants is called *abstraction safety*.As pointed out in [18], producing the same abstract types as soon as two module arguments have the same type fields may break this property.

In order 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. An approximation tracking the equality of both values and type fields would be too fine-grained and cumbersome, as modules may have numerous value fields. OCaml follows a compromise, coarser-grained approach to enforce abstraction-safety by tracking equalities only at the module level: two modules are deemed similar when they can be resolved to the same original module. This was originally introduced as a *syntactic criterion* by Leroy [9], which was latter extended to a static tracking of module aliases.

2.4 Aliasing and ascription

To allow tracking of aliasing, OCaml offers a notion of *module alias* [5][5]: the signature language is extended with the *alias signature* $(= P)$ to express that the module is a statically known alias of (the module at the path) P. We say that the two modules share the same *identity* when one is an alias of the other. Keeping as much aliasing information as possible allows for more type equalities when using applicative functors[6].

Unfortunately, the aliasing feature is actually heavily restricted in OCaml because it interacts with ascription in a subtle way. To see why, we need to consider some details of the compilation scheme of ML modules. First, aliases induce a type-driven compilation: if a module has an alias signature $(= P)$ it can only be compiled *as P is compiled*. Second, all ML-module systems, and OCaml in particular, structures are compiled via *static dispatch* (i.e.,, accesses inside a structure made with fixed offsets, not with an access table) for performance trade-offs reasons. Therefore, all module that are aliases of each other must have the same memory representation.

However, in a static dispatch setting, as ascription can drop and reorder fields, it can change the memory representation. That is why ascription should *break* modules aliases. However, as ascription also occurs at functor call, aliases between a functor parameter and its body must be

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

[5]OCaml actually offers two distinct notions of aliases, which are respectively *present* or *absent* at runtime. Here, we only consider *present* aliases.

[6]Historically, the main motivation behind module aliases was rather the finer grained control of compilation units than the interaction with applicative functors

removed, which limits their usefulness. This problem induced a set of ad-hoc, fairly complex restrictions in OCaml that can actually be bypassed in some edge-cases[7].

As an example, let us consider the functor $(Y : \mathtt{S}) \longrightarrow Y$. It cannot be given the expected type $(Y : \mathtt{S}) \longrightarrow (= Y)$. Indeed, when applying it to an argument P that has a signature not exactly equal to `S`, the result has a different memory representation than P.

Interestingly, *transparent ascription*, an operation written `(M : S)` in SML, helps solving this issue. It restricts the outside view of a module `M` to the fields present in `S` while preserving all type equalities. However, this feature does not increase expressivity in SML as a similar result could be obtained via an *usual* (opaque) ascription of SML `(M :> S')` with a signature `S'` 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 a 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)`, hereafter called *concrete ascription* to avoid the confusion with other notions of transparency, is just syntactic sugar for an opaque ascription with a concrete signature (see §3.1).

Thanks to concrete signatures, aliasing information can be preserved through ascription (explicit or implicit at functor calls). As OCaml features applicative functors (unlike SML), this would increase the expressivity of the signature language. Figure 3 demonstrates a code pattern, present in the OCaml ecosystem [23], where this would be useful. This pattern combines a functor (`Make3D`) that reexports its argument (`K`) with an applicative functor called twice (`Set`), once on the argument and once on the reexported argument so that the modules resulting from both applications may interact. When using fixed interfaces (here, `VectorSpace`), as usually done in libraries, type-level sharing is not sufficient and module-level aliasing via transparent ascription is required. By contrast, when the granularity of applicativity relies on static equivalence as in Moscow ML, the example would typecheck without requiring concrete ascription: there is a design trade-off between abstraction safety and flexibility.

The notion of module identity is also essential to *modular implicits* [24], a proposal aiming to leave some module expressions implicit, inferring them 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. Thanks to concrete signatures that enables more sharing of identities in signatures of inferred modules, aliasing become a good static approximation of 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 expressivity 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

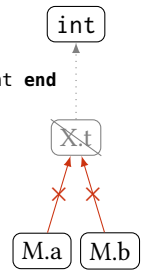
[7] See the following issues: OCaml #7818, OCaml #2051, OCaml #10435, OCaml #10612 and OCaml #11441.
[8] OCaml #10612. Transparent ascription is written `(P :> S)` in the OCaml #10612 proposal, the opposite of the SML convention.

```
1  module type Field = ...
2  module type VectorSpace = sig
3    module Scalar : Field
4    ... (** more fields *)
5  end
6  module Set(Y:...) = ...
7  module LinAlgebra(V:VectorSpace) = struct
8    module SSet = Set(V.Scalar)  ...
9  end
10 module Make3D(K:Field) = (struct
11   module Scalar = K
12   ... (** built from K *)
13 end : sig
14   module Scalar : (= K < Field)  ...
15 end)
16 module Reals = ...
17 module Space3D = LinAlgebra(Make3D(Reals))
18 (** Space3D.SSet.t =? Set(Reals).t *)
```

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

Fig. 4. Example of a signature avoidance situation and the associated type-dependencies tree. The module `M` is built by projecting *only* the sub-module `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 Example 3.1 of the supplementary materials.)

```
1  module type S = sig type t end
2  module M = (struct
3    module X:S = struct type t = int end
4    module Y = struct
5      type a = X.t * bool
6      type b = X.t * int
7    end
8  end).Y
9  let f ((x,_): M.a): M.b = (x, 3)
```



while a value of type `t list` is still in scope. An example of such pattern is given in Figure 4. Sometimes, no possible signature exists for a module; other times there are several incompatible ones. Specifically with applicative functors, when higher-order abstract types are out of scope, there are often only incompatible 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 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 typing 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 if it would be practical.

Signature avoidance in practice. OCaml developers 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 4) 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 typing 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 both module expression and source signatures. 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, while hiding the complexity and artifacts that come from the actual encoding of module expressions into F^ω. 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 5, is built on top of a core language of expressions $\mathtt{e}$ and types $\mathtt{u}$ 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 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*, $\mathtt{M}$, $\mathtt{S}$, 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 ($\mathtt{struct}_A \ \ldots \ \mathtt{end}$), 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.
- Prefixes, written with the letter Q, range over either a path P or a self reference A.
- *Abstract types* are specified as types pointing to themselves, e.g., $\mathtt{type}\ t = A.t$ where A is the self-reference of the current structure (and often grayed out for readability).
- As a convention, bindings and declarations can be written in the form $I : \mathtt{B}$ and $I : \mathtt{D}$, with the identifier extracted, i.e., a type declaration $\mathtt{type}\ t = \mathtt{u}$ can be written as $t : \ \mathtt{type}\ \mathtt{u}$, a module binding $\mathtt{module}\ X : \mathtt{M}$ can be written as $X : \ \mathtt{module}\ \mathtt{M}$.

Path and Prefix

$P ::= Q.X$ (Access)
$\mid Y$ (Functor argument)
$\mid P(P)$ (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)
$\mid () \longrightarrow \mathtt{M}$ (Generative)
$\mid (Y : \mathtt{S}) \longrightarrow \mathtt{M}$ (Applicative)
$\mid \mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end}$ (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 signatures

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

Source declarations

$\mathtt{D} ::= \mathtt{val}\, x : \mathtt{u}$ (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. 5. Syntax of the source language.

M^ω Types

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

Environment

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

Opacity

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

M^ω signatures

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

M^ω declarations

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

Fig. 6. Syntax of M^ω signatures.

- We use a distinct class of variables, written Y, for functor parameters, which can be freely renamed. By contrast, and as usual with modules, neither identifiers X and T for module expressions and signatures, nor the identifiers x and t for core language expressions and types can be renamed, as they play the role of both an internal and an external name.

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. [4]. Contrary to *F-ing*, but following previous approaches of Leroy [9] and Russo [20], we chose to define a syntactic notion of path and use it to restrict qualified accesses and functor applications, while allowing a general form of projection:

- Qualified access and projection inside a *module type* are disallowed, since a prefix Q may not originate from a module type identifier T.
- A qualified access inside a generative functor application, which would be of the form $G().x$, 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.
- We allow projection on any module expression, but we restrict both types of functor applications to paths. OCaml does the opposite, mainly to prevent cases prone to trigger signature avoidance. Our choice is more general, as the OCaml one can be easily encoded, while the converse requires an explicit signature annotation on the functor argument. We could add the following syntactic sugar:

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

$$\mathtt{M}() \triangleq (\mathtt{struct}_A\ \mathtt{module}\, G = \mathtt{M}\ \ \mathtt{module}\, \mathsf{Res} = A.G()\ \mathtt{end}).\mathsf{Res}$$

  In our setting, the only expression that can "cause" signature avoidance is the projection.
- Similarly, we restricted ascription to paths. The general form can be added as syntactic sugar:

$$(\mathtt{M} : \mathtt{S}) \triangleq (\mathtt{struct}_A\ \mathtt{module}\, X = \mathtt{M}\ \ \mathtt{module}\, \mathsf{Res} = (A.X : \mathtt{S})\ \mathtt{end}).\mathsf{Res}$$

$$(\mathtt{M} < \mathtt{S}) \triangleq (\mathtt{struct}_A\ \mathtt{module}\, X = \mathtt{M}\ \ \mathtt{module}\, \mathsf{Res} = (A.X : ((= A..X < \mathtt{S}))\ \mathtt{end}).\mathsf{Res}$$

However, as both path and module expressions feature a projection dot, the grammar is slightly ambiguous. This does not impact the typing system, as we always see paths as a subset of module expressions and only consider the more general projection dot of module expressions for 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 nullary functor. Indeed, we could add the usual currying notation sugar:

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

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: $(Y : S1)(Y : S2)() \to M$ is *fully generative* (every instance is new), while $(Y : S1)()(Y : S2) \to M$ is generative with regard to the first argument, then applicative with regard to the second one.

3.2 M^ω overview

In Figure 6, we introduce the syntax for M^ω-signatures $\mathcal{C}$ (and M^ω-declarations $\mathcal{D}$), a more expressive signature language. By convention, we use curvy capitals ($\mathcal{C}, \mathcal{D}, \ldots$) for M^ω-objects. M^ω objects also use M^ω-types τ instead of source types u, obtained by replacing qualified types $Q.t$ by abstract types $\alpha(\overline{\tau})$ (or concrete types τ) where α range over (a new collection of) abstract type variables.

We annotate existential types with an opacity flag $\diamond$ which can be either opaque ($\blacktriangledown$) or transparent[9] ($\triangledown$). A transparent existential ensures that its witness is actually statically known, while by default, the witness may depend on a dynamic choice, in and we use an opaque existential type. This distinction will be used for the typing of applicative and generative functors.

M^ω signatures $\mathcal{C}$ use explicit quantification over abstract types $\overline{\alpha}$. This quantification is universal in the signature of an applicative functor $\forall\overline{\alpha}.\mathcal{C} \to \mathcal{C}$, and existential in the signature of the body of a generative functor $\exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}$ ($\blacktriangledown$ indicates that this existential is opaque, i.e., a standard existential type).

[9]This notion of transparency is *unrelated* to concrete ascription (called transparent ascription in SML).

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

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

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

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

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

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

Fig. 7. Signature typing rules.

Module types $\lambda\overline{\alpha}.\mathcal{C}$ are parametric [10] in each type variable α, which may later become universally quantified (for the parameter of a functor), existentially quantified (for an abstract type) or replaced by a concrete instance.

Typing environments contain three types of bindings: an abstract type variable α, a functor argument $Y : \mathcal{C}$, or an identifier prefixed by a self-reference $A.I : \mathcal{D}$. To prevent shadowing, we use the *well-formedness* predicate over environments $\mathrm{wf}(\Gamma)$ that also check that identifiers appear at most once—in addition to recursively checking well-formedness of all bindings in Γ. As a simplifying convention for the rest of this paper, we consider well-formedness of the environment as a precondition to all rules.

OCaml allows shadowing of values and some form of shadowing of type bindings in module expressions via the *include/open* operators. Bindings and declarations inside a submodule can also locally shadow a definition made in an enclosing structure. In our setting, we reject shadowing of both values and types, as typing environments must have distinct bindings[11]. However, local definitions that shadow a definition made in an enclosing module are allowed, as they would have a different self-reference prefix and are therefore considered as two different bindings.

The M^ω system uses three main judgments: typechecking of signatures, subtyping, and typechecking of expressions, which we present in this order.

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 type parameters $\overline{\alpha}$ explicit. The set of rules is given in Figure 7 and discussed below.

Declarations. Typechecking of signatures uses a helper judgment for typechecking declarations $\Gamma \vdash_A \mathsf{D} : \lambda\overline{\alpha}.\mathcal{D}$ for which, we only give the key rules, referring to §4 of the supplementary materials for the full set. 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 actually introduced by abstract type declarations:

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

[10] Here, we follow Russo [20] rather than [18] and use a lambda quantifier for signatures.
[11] This is not restrictive as this can always be solved by appropriate renaming.

An abstract type is accessible *not only* in the local module, but in all the following ones. Therefore, the λ-binder for that abstract type α must be *lifted* to enclose the whole region where α is accessible.

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

Thus, in Rule M-Typ-Decl-Mod for typing a submodule declaration $\mathtt{module}\, X : \mathsf{S}$, the set of abstract types $\overline{\alpha}$ quantified in the submodule M^ω-signature $\lambda\overline{\alpha}.\mathcal{C}$ of S is extruded in front of the M^ω signature of the declaration $\lambda\overline{\alpha}.(\mathtt{module}\, X : \mathcal{C})$. In addition, in Rule M-Typ-Decl-Seq for merging a list of declarations $\mathsf{D}_0, \overline{\mathsf{D}}$, the types $\overline{\alpha}_0$ introduced by the first declaration D_0 are put in the context for typing the rest of declarations $\overline{\mathsf{D}}$ and both sets of abstract types $\overline{\alpha}_0$ and $\overline{\alpha}$ are merged together in front of the list of M^ω-declarations $\mathcal{D}_0, \overline{\mathcal{D}}$. Here, we use the convention that declarations can be seen as having the identifier extracted in front: the identifier $A.I$ is implicitly extracted from $\mathcal{D}_0$, seen as $A.I : \mathcal{D}_0$.

Signatures. Typing rules for signatures can be found on Figure 7. 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. For a concrete signature $(= P < \mathsf{S})$, the signature S is elaborated into an M^ω-signature $\lambda\overline{\alpha}.\mathcal{C}'$ and checked against the M^ω-signature $\mathcal{C}$ of path P (Rule M-Typ-Sig-Con). The result signature is *not* $\mathcal{C}'$ but $\mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]$ 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 introduced in their bodies is limited: the lifting of abstract types $\overline{\alpha}$ stops at 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 $\overline{\alpha}$, which matches with the specification for generative functors. By contrast, two applications of the *same* applicative functor to the *same* argument should produce the *same* abstract types. This means that applicative functors cannot 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, and neither of the form $\lambda\beta.\forall\overline{\alpha}.\mathcal{C} \to \mathcal{C}'$, where all applications would share the same types regardless of their argument.

The solution, introduced by Biswas [1] is to use an higher-order abstract type β' applied to the universally quantified variables $\overline{\alpha}$, to capture the fact that the type is *some type function* of the arguments. This gives a signature of the form $\lambda\overline{\beta'}.\forall\overline{\alpha}.\mathcal{C} \to \mathcal{C}[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}]$ which can be seen in the rule M-Typ-Sig-AppFct.

3.4 Subtyping

The subtyping judgment $\Gamma \vdash \mathcal{C} < \mathcal{C}'$ (with the helper judgment $\Gamma \vdash \mathcal{D} < \mathcal{D}'$) checks that a signature $\mathcal{C}$ is *more restrictive* than a signature $\mathcal{C}'$, meaning that the former has more fields and introduces less abstract types. It ensures that a module with signature $\mathcal{C}$ can be coerced into a module of signature $\mathcal{C}'$. Such coercions usually imply copying and are not code-free.

We only highlight two key rules below, referring to §4.1 of the supplementary materials for the full set of rules:

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

Rule M-Sub-Sig-Struct compares two structural signatures where deletion and reordering of fields is allowed; using a subset of the left-hand side declarations compared against the full set of right-hand side ones. Rule M-Sub-Sig-GenFct for generative functors amounts to check subtyping

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

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

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

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

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

Fig. 8. Module (and path) typing rules.

between existential types $\Gamma \vdash \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C} < \exists^{\blacktriangledown} \overline{\alpha}'.\mathcal{C}'$, 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 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. [18], and shown decidable for certain pairs of signatures $(\mathcal{C},\ \mathcal{C}')$ satisfying a syntactic condition, 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. One exception would be the construct `module type of` P, if it were not omitted, as it could inject inferred signatures as source-like signatures. The solution is then to restrict the typing of `module type of` P to cases where the inferred signatures of P is indeed a source-like one, which is called an *explicit* signature in [18]. The same argument would apply to the M^{ω} system.

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 usually 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. When there is no existential type and the mode is omitted, it can be any mode $\diamond$. 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 §4.2 of the supplementary materials. Typing rules for expressions M-Typ-Mod-* are given on Figure 8.

Skolemization. The need for two modes of typing comes from the treatment of applicative functors, specifically the 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 side of the arrow) by making them higher-order. This is known as *skolemization* and was introduced by Russo [20]. However, this is sound only if the abstract types have a statically known witness,

which we enforce by requiring transparent existentials for the body of the functor. The technical reasons behind the need for a statically known witness are given in §5.4.

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, computing impure core expressions[12], 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 (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 :

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

We also rely on a core-language expression typing judgment[13] $\Gamma \vdash^{\diamond} \mathsf{e} : \tau$ equipped with a mode that tracks the presence of effects[14]. 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., with the convention that 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 ($= P < \mathsf{S}$): 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 ($P : (= P < \mathsf{S})$), i.e., the opaque ascription of P to the concrete signature ($= P < \mathsf{S}$), 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, making the projection of a submodule $\mathsf{M}.X$ delicate: the dependencies of X might become dangling after the other components of signature of M have been lost. By contrast, dependencies in M^{ω} signatures only consist of the use of a common abstract type that has previously been lifted so as to be in scope. Thus, M^{ω} signatures do not have hidden internal dependencies and so, do not need a self-reference. The projection rule M-Typ-Mod-Proj just keeps the subset $\overline{\alpha}'$ of abstract types $\overline{\alpha}$ that are free in the submodule signature $\mathcal{C}$ to ensure that it has no dangling dependency. This rule performs garbage collection of unused abstract types.

[12] While a full tracking of effect in the core-language would be needed, the current implementation of OCaml only prevents module-related operations: unpacking a first-class module and calling a generative functor

[13] This judgment is trivially extended by rules for accessing module paths; the added rules are given in §4.2 of supplementary materials.

[14] In current OCaml, side effects in the core language are not tracked by the typing system, it is the user's responsibility to require the generative mode in such cases.

Tagging

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

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

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

Signatures

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

Fig. 9. Source-to-source transformation introducing identity types for structures and functors using two reserved identifiers `id` and `Val`. Bindings and declarations are transformed by immediate map over submodules and submodule-types.

It would also be sound not to enforce it, by letting $\overline{\alpha}'$ range between $\mathsf{fv}(\mathcal{C}) \cap \overline{\alpha}$ and $\overline{\alpha}$ (anchoring would later take care of it). Notice, that opaque existential quantifiers may then all disappear in the conclusion, but the mode of the judgment will remain generative in the conclusion.

3.6 Identity, aliasing, and type abstraction

So far, our system handles applicativity with a granularity of *type-only* applicativity, as promoted by [20] and *F-ing*. To obtain *abstraction safety*, Rossberg et al. [18] 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 (*à la* OCaml) applicativity by marking only modules. The downside of this approach is to clutter the typing rules.

However, as phantom 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 tags as a special type field in Figure 9. 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 an abstract type binding and the actual value. New (abstract) identity tags are introduced only when typing structures and functors, or via an ascription with a signature that has an abstract id field.

Controlling the applicativity granularity by a source-to-source transformation avoids cluttering the 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.

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).

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

Proof sketch. The proof uses bounded polymorphism to add a supertype bound to every identity field, 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 subtyping, which in turn ensures that the type of a *Val* field is always a supertype of the bound of its *id* field. Details can be found in §4.3 of supplementary materials. □

4 REBUILDING SOURCE SIGNATURES

In this section we present the reverse translation from M^ω signatures back into the source OCaml syntax, which we call *anchoring*. This translation is not complete because the source syntax has a less expressive signature language. However, using M^ω signatures, we can precisely describe the different signature avoidance cases, from which we extract three *guidelines* for a reverse translation algorithm. In the perspective of using M^ω as an internal representation of the typechecker, anchoring is a necessary step to output familiar source signatures to the user.

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 ($\mathtt{type}\, t \;=\; \mathsf{u}$) and that of an *abstract* type ($\mathtt{type}\, t = A.t$). An abstract type declaration $\mathtt{type}\, t = A.t$ 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 $\mathtt{type}\, t = \mathsf{u}$ 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 use new abstract types without having a type field 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. In an untagged setting, source signatures can only express identity sharing via *concrete* signatures ($= P < \mathsf{S}$), 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 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:

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

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 depends on the subtyping between S and S':

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

Guideline 3. Source signatures can only express sharing of higher-order abstract types when all occurrences can be written as paths to the anchoring point.

While an anchoring algorithm that follows only this principle for higher-order types would be sound, we argue that it would be too permissive.

Decidability. In the general case, 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 limiting the type declarations type $t = \varphi(\overline{\alpha})$ deemed suitable for anchoring to those that are parameterized exactly *as the original definition* point, i.e., to occur inside an applicative functor that is parametric in *exactly* $\overline{\alpha}$. In the example given above, the functor F is parameterized over $\overline{\alpha}$, the same set of variables as the original definition point. Hence, the anchoring point would be deemed valid.

Disabling functor applications out of thin air. While sound and decidable, the heuristic described above 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 could be quite surprising, if not misleading, as it suggests a computation that will never happen.

For instance, as in 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 expressivity and usability, we argue that this should be accepted only when F' is an alias of F, but rejected when F' happens to copy the type field of F, as it might also perform additional computation and might have never been called on argument X.

To distinguish between the two cases, we slightly instrument the typing rules. Interestingly, we only have to track functor applications: if a type declaration type $t = \varphi(\overline{\alpha})$ is obtained via a functor application, we mark it as *unsuitable* for anchoring. By contrast, the original type declaration, is not obtained via an application and is left unmarked, hence available for anchoring. Similarly, aliasing a functor copies it signature unmarked, making type declarations available for anchoring.

4.2 The anchoring process

For pedagogic purposes, the anchoring algorithm is split in two steps, first we translate canonical 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, leaving the full sets of rules in **??**.

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[15]. We also instrument Rule C-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 a flat list.

Marking higher-order abstract types. Finally, we modify the typing rule for functor application C-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 $t = \tau$ have been rewritten into type $t = \tau^\dagger$. The marking does not go inside the body of functors, or inside module types. Therefore, we only change the resulting signature of

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

the rule C-Typ-Mod-AppApp to a marked signature $\mathcal{C}'^{\dagger}[\overline{\alpha} \mapsto \overline{\tau}]$. Marks are kept once introduced but ignored by typing and subtyping rules, which may freely erase them (by β-reduction).

4.2.1 From canonical signature to tagged source signatures. The algorithm proceeds by visiting the canonical signature in left-to-right depth-first order. Along the way, it removes all universal and existential quantifiers from the canonical signature and replaces occurrences of the corresponding abstract types either 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 canonical types to type expressions with qualified types is built and updated during the visit. The algorithms is defined by mutually recursive judgments:

- **Environment anchoring** $\Gamma \hookrightarrow \theta_\Gamma$ checks that θ_Γ is a correct anchoring map for Γ.
- **Signature anchoring** $\Gamma ; \theta_\Gamma \vdash \mathcal{C} \overset{P}{\hookrightarrow} \mathtt{S} : \theta$, given a path P (which is actually shallow, i.e., taking one of the three forms Y, $A.X$, or $A.Val(Y)$), translates the canonical signature $\mathcal{C}$ into a tagged source signature $\mathtt{S}$ and produces a (possibly empty) local anchoring map θ of the abstract types anchored in $\mathtt{S}$, prefixed by P. We also define declaration anchoring $\Gamma ; \theta_\Gamma \vdash \mathcal{D} \overset{A}{\hookrightarrow} \mathtt{D} : \theta$, with a self reference A in place of the path P.
- **Local anchoring** $\Gamma ; \theta_\Gamma \vdash \mathcal{C} \hookrightarrow \mathtt{S} : (\overline{\alpha} \mapsto \_)$ just checks that signature $\mathcal{C}$ can be translated into $\mathtt{S}$ producing a local map of domain $\overline{\alpha}$ that is ignored afterwards.
- **Type anchoring** $\Gamma ; \theta_\Gamma \vdash \tau \hookrightarrow \mathtt{u}$ translates the canonical type τ to a source type $\mathtt{u}$, replacing each canonical variables by a path to its anchor, as described in θ_Γ.

The key rules are those that extend, update, or use the anchoring map to reconstruct source type expressions and are highlighted below. We refer the reader to **??** 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 not anchored yet, i.e., not in the domain of θ_Γ (simplified version for first-order types):

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

For first-order types, we ensure that the type declaration is not made inside an applicative functor, i.e., the environment contains no universally quantified types[16] since the introduction of α. We also check that the type was not introduced before the left-most barrier, by requiring $\alpha \in \Delta$. We then return the singleton map $(\alpha \mapsto A.t)$. The general version of the rule considers a declaration for a higher-order *unmarked* type expression φ:

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

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). When either one of the conditions is unsatisfied, the anchoring fails. The process could be made more permissive or more restrictive by tweaking this rule, as discussed in §4.1.

The anchoring map θ is *updated* in two places, corresponding to the points were access paths to types change as we exit scopes, i.e., when leaving :

- a *structural signature*: in Rule A-Sig-StrPath, the self-reference A is replaced by P in every occurrence of θ – making abstract types locally anchored inside the signature reachable via P.

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

- or an *applicative functor*: in Rule A-Sig-FctApp, $\lambda Y.\theta$ means the point-wise (mathematical) abstraction of the anchoring map θ by the functor parameter Y,

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

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

By contrast, the anchoring map of the body of a generative functor is thrown away (Rule A-Sig-FctGen), as generative functors cannot appear in paths[17], and a barrier is added in the premise: the body cannot capture types defined outside of the functor:

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

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

We ensure that *exactly* the abstract types $\overline{\alpha}$ of the body are anchored inside it using the premise $\Gamma, \overline{\alpha}\,;\,\theta_\Gamma \vdash \mathcal{C} \hookrightarrow \mathsf{S} : (\overline{\alpha} \mapsto \_)$. This is actually just a helper judgment define by the single rule A-Sig-StrNone.

Finally, the anchoring map is used for translating canonical types τ into source ones. Here, we consider types stripped of their marks, e.g., by reducing $\varphi^\dagger$ to φ. Rule A-Type-Application applies when τ is of the form $\varphi\langle\tau_1 \dots\rangle \dots \langle\tau_n \dots\rangle$ and $\theta_\Gamma(\varphi)$ is a (mathematical) function of the form $\lambda Y_k. \dots \lambda Y_n.P.t$. Types τ_k to τ_n may only be resolved to tags $P_k.id$ to $P_n.id$. Then, the resulting type u is the path (resulting from the mathematical application) $\theta_\Gamma(\varphi)(P_k) \dots (P_n)$. However, u must be retyped to ensure that paths occurring in τ only contain valid functor applications and returns the same as the input type τ.

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

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 $\Gamma \vdash \mathsf{S} \hookrightarrow \mathsf{S}'$, for which the two interesting rules are for untagging structural signatures:

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

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

[17] Similarly, Rule A-Decl-ModType only checks for anchorability of the body of a module type, and then discard the anchoring map, since module types cannot appear in paths.

4.3 Properties of anchoring

Anchoring of canonical signatures returns source signatures that are strongly linked with their input. Several source signatures may actually express the same sharing: their typechecking induces an equivalence on source signatures as follows:

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

Anchoring and signature typing are inverse of each other up to equivalence of source signatures.

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

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

The reverse statement is more subtle: canonical signatures can express the fact that they are inside an applicative functor (their abstract types are applied to universally quantified type variables), while source signatures cannot, hence the correspondence is up to *skolemization*:

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

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

Both results are proven by induction. Some details are given in **??**.

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

Proof sketch. The proof is by induction, using an invariant to ensure that the source signature has a correct tagging structure (identity tags are present according to the source to source transformation and not duplicated), which is maintained by typing and the first step of anchoring. □

4.4 Discussion

Anchorable signatures correspond to the elaboration of source signatures. Canonical signatures are richer and can express a finer grained sharing than source signatures. In particular, they can track sharing through complex module operations, including after projections (which are a typical source of anchoring failure). However, canonical signatures may also keep *too much information*, revealing the history of the module operations without providing more type-sharing, which may actually be another source of anchoring failure. This typically happens when a type variable has become "unreachable". We identify two patterns where canonical signatures have *artifacts* revealing history and preventing anchoring.

Loss of a type argument. The signature[18] $\exists\varphi,\alpha.\ \mathcal{C}[\varphi,(\varphi\,\alpha)]$, could be the result of a functor that has been reexported and also applied to a module that became unreachable. Such a situation typically arises when an unnamed module is used as argument, e.g., as in $F(\texttt{struct}_A\ \ldots\ \texttt{end})$. The application $\varphi\,\alpha$ represents the abstract type created by the application of the functor, which keeps track of the (abstract type of the) argument to which the functor has been applied: if this application were repeated, it would return the same abstract type. However, since the argument is no longer accessible, this information became useless. Hence, we could give the module the simpler signature $\exists\varphi,\beta.\ \mathcal{C}[\varphi,\beta]$ cutting the (original) link to the functor. There is actually a retyping function[19] from the former to the latter signature, which we can state as a subtyping relationship:

$$\exists\varphi,\alpha.\ \mathcal{C}[\varphi,(\varphi\,\alpha)] \ \preceq\ \exists\varphi,\beta.\ \mathcal{C}[\varphi,\beta]$$

[18] Here, we use the notation $\mathcal{C}[\alpha,\ldots]$ to indicate that α appears freely in $\mathcal{C}$ and the notation $\mathcal{C}[(\varphi\,\alpha),\ldots]$ 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 $\mathcal{C}$.
[19] A function without computational content, i.e.,. that is $\beta\eta$-equivalent to the identity.

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

Fig. 10. Syntax of F^ω

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 these signatures are not isomorphic, as there is no coercion going in the opposite direction.

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 could be simplified by subtyping:

$$\exists\, \varphi, \alpha, \beta.\ \mathcal{C}\,[\alpha, \beta, (\varphi\ \alpha), (\varphi\ \beta)] \ \preceq\ \exists \alpha, \beta, \alpha', \beta'.\ \mathcal{C}\,[\alpha, \beta, \alpha', \beta']$$

Again however, subtyping does not work in the opposite direction.[20]

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 the encoding in F^ω. Yet, the encoding of module expressions and signatures in F^ω 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. It can also be used to understand modular programming, directly in F^ω or how to conduct proofs of modular programs.

The encoding is largely based on the work of Rossberg et al. [18], but differs in a key manner for the treatment of skolemization, used to encode abstract types of applicative functors. One of our contributions 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 process, as well as the resulting programs.

5.1 F^ω with kind polymorphism

We use a variant of explicitly typed F^ω with primitive records, 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.

[20] Actually, if the functor where called only once, we would also have $\exists \alpha, \alpha'.\ \mathcal{C}\,[\alpha, \alpha'] \preceq \exists\, \varphi, \alpha.\ \mathcal{C}\,[\alpha, (\varphi\ \alpha)]$, hence there is an isomorphism between those types.

The syntax of F^ω is given in Figure 10. Typing rules are standard and available in §5 of supplementary materials. 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 the core language types u and expressions e, 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 even often leave them implicit. We actually always do so in the elaboration typing rules below.

For convenience, we use n-ary notations for homogeneous sequences of type-binders. We introduce a let-binding $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \triangleq (\lambda(x : \_).\, e_2)\, e_1$ and n-ary pack and unpack operators defined in the following way:

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

5.2 Encoding of signatures

M^ω signatures are actually F^ω types with some syntactic sugar. 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):

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

5.3 Sharing existential types by repacking

While the correspondence between M^ω signatures and F^ω types is straightforward, the actual encoding of module expressions as F^ω terms is slightly more involved. Although structures and functors are simply encoded as records and functions, a serious 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* [14] and of *recursive type generativity* [3]. Here, in order to stay in plain F^ω, we 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 $\mathsf{repack}^\blacktriangledown\,\langle\bar\alpha, x\rangle = e_1\ \mathsf{in}\ e_2$, which allows the abstract types of e_1 to appear in the type of e_2, which we leave implicit here since the type of repacking is here fully determined by the combination of $\bar\alpha$ and the type of e_2:

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

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 $\overline{\alpha}$:

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

It is better understood by its use in the example of §3.1.3 and the derived typing rule:

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

5.4 Transparent existential types and their lifting through function types

The repacking pattern allows lifting existential types outside of product 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 [21]. 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.

We create a transparent existential type with the expression $\mathsf{pack}\ e\ \mathsf{as}\ \exists^{\nabla\tau}(\alpha : \kappa).\sigma$, which behaves much as $\mathsf{pack}\ \langle\tau, e\rangle\ \mathsf{as}\ \exists^\blacktriangledown(\alpha : \kappa).\sigma$, 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 $\mathsf{unpack}\ \langle \alpha, x\rangle = \mathsf{seal}\ e_1$ 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:

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

Their typing rules are given in §5 of supplementary materials. 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 in a similarly 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 given in §5.1 of supplementary materials.

5.5 Implementation of transparent existential types in plain F^{ω}

Interestingly, transparent existential types are completely definable in plain F^{ω}. A concrete implementation is given in §5.2 of supplementary materials. 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 $e_{\mathbb{E}} = \mathsf{pack}\ \langle \tau_0, e_0\rangle$ 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 $\mathsf{unpack}\ \langle \mathbb{E}, x_{\mathbb{E}}\rangle = e_{\mathbb{E}}$ in e in plain F^{ω}, with the following additional

syntactic sugar[21]:

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

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 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 left the rules in §6.1 of supplementary materials. The judgments have the following property regarding F^ω typing:

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

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 $\nabla\tau$ together its a 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 ∇ 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, when the mode $\diamond$ should coincide with $\bar\vartheta$ when present and is actually left implicit as for the M^ω system. Hence, we usually just write $\Gamma \vdash \mathtt{M} : \exists^{\bar\vartheta}\bar\alpha.\mathcal{C} \rightsquigarrow e$. The judgment is also defined for bindings $\Gamma \vdash^{\bar\vartheta}_{A} \mathtt{B} : \exists^{\bar\vartheta}\bar\alpha.\mathcal{D} \rightsquigarrow e$ (with the mode left implicit). The properties of the two judgments are detailed below.

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.*

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

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

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

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

[21] 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.

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

The full set of typing rules for expressions is given in §6.2 of supplementary materials. Below, we only present an excerpt of the most significant rules.

Elaboration of structures. 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

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

E-Typ-Bind-SeqApp

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

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:

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

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

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

Modes and sealing. By default, typing 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 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 [13]. This vision was opposed by MacQueen [11] 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. [7] supported the idea that dependent types may actually be too powerful (thus, unnecessarily complex) for module systems. SML modules were first described by Harper et al. [7]. Two approaches for the formalization and improvement of abstract types in SML were later independently yet simultaneously described by Leroy [8] using manifest types and Harper and Lillibridge [6] via an adapted F^ω with translucent sums. The genesis of the OCaml module system was specified by Leroy [8, 10] with, later, an extension to applicative functors [9].

The key idea for a simplified link between modules and F^ω, developed by Russo [20], was to use existential types to interpret signatures. Pursing a related objective, Dreyer [3] proposed to model generativity using stamps instead of existential types, while Montagu and Rémy [15] proposed a similar, but logically-based approach, through the concept of open existential types.

Pushing Russo's idea further, an important step forward was achieved by Rossberg et al. [18] with the elaboration of a large subset of the SML module system into F^ω, dubbed the *F-ing* approach. *F-ing* gives a *syntactic* translation from SML syntax directly into F^ω, thus providing semantics by elaboration. *F-ing* is safe by construction[22], 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. This makes direct reasoning on the source program difficult, if at all feasible for the programmer. By contrast, M^ω gives a specification directly on source terms, without having to think in terms of encoding, but leveraging the insights provided by the elaboration to F^ω. Our anchoring algorithm allows for the reverse translation from encoded signature to source syntax, improving the understanding of the cases of signature avoidance. A more detailed, technical comparison can be found in §2 of supplementary materials.

Moving one step further, Rossberg [17] 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 existentials types technique to 1ML.

More recently, Crary [2] 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, not on its origin nor 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.

7 DISCUSSION AND 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. Using M^ω, we first shone a new light on the mechanisms of the

[22]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.

OCaml type system, and provided a detailed description of the solvable and unsolvable cases of signature avoidance. Second, 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.

A few features from current OCaml have been omitted, most of which for sake of simplicity, as they should not impact the overall structure of the system, only adding more cases in the set of rules. However, it is not obvious how to extend the M^ω system to cope with abstract signatures, which are a particularity of OCaml. While the F^ω-style polymorphism could perhaps be sufficient to solve the very few useful cases, a general approach is likely to be more challenging.[23]

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 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.

The introduction of transparent existential types makes the treatment of applicative and generative functors much closer to one another: existentials types are introduced transparently when assembling components of modules and only turned into opaque ones by need when hitting a generative component. Then, neither functors nor applications of functors need to be aware of the mode. Functors move transparent existential types in front of the functors, leaving opaque ones in the body. Applications need not even be aware of existential types and are just a standard application in F^ω. This considerably simplifies the treatment of applicative functors, which should ease the mechanized proof of Theorem 5.1, which we plan as future work. It should also benefit to the appealing approach of 1ML. While the goal of 1ML is to remove the stratification between core and module layers, and directly program modules in F^ω, we may also explore another path, extending F^ω with minimalist constructs, typically for dealing with primitive lifting of existentials, so that we may program *with modules* in this light extension of F^ω. Besides programming directly in F^ω, this should also help structure and conduct proofs of modular programs.

REFERENCES

[1] S. K. Biswas. Higher-order functors with transparent signatures. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 154–163, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897916921. doi: 10.1145/199448.199478. URL https://doi.org/10.1145/199448.199478.

[2] K. Crary. A focused solution to the avoidance problem. *Journal of Functional Programming*, 30:e24, 2020. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796820000222. URL https://www.cambridge.org/core/product/identifier/S0956796820000222/type/journal_article.

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

[4] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In A. Aiken and G. Morrisett, editors, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 236–249. ACM, 2003. doi: 10.1145/604131.604151. URL https://doi.org/10.1145/604131.604151.

[23] More details on omitted features and abstract signature can also be found in supplementary materials.

[5] J. Guarrigue and L. White. Type-level module aliases: independent and equal. ML Family/OCaml Users and Developers workshops, 2014. URL https://www.math.nagoya-u.ac.jp/~garrigue/papers/modalias.pdf.

[6] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, page 123–137, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 0897916360. doi: 10.1145/174675.176927. URL https://doi.org/10.1145/174675.176927.

[7] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, page 341–354, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913434. doi: 10.1145/96709.96744. URL https://doi.org/10.1145/96709.96744.

[8] X. Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, page 109–122, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 0897916360. doi: 10.1145/174675.176926. URL https://doi.org/10.1145/174675.176926.

[9] X. Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95*, pages 142–153, San Francisco, California, United States, 1995. ACM Press. ISBN 978-0-89791-692-9. doi: 10.1145/199448.199476. URL http://portal.acm.org/citation.cfm?doid=199448.199476.

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

[11] D. B. MacQueen. *Using Dependent Types to Express Modular Structure*, page 277–286. Association for Computing Machinery, New York, NY, USA, 1986. ISBN 9781450373470. URL https://doi.org/10.1145/512644.512670.

[12] A. Madhavapeddy, R. Mortier, C. Rotsos, D. J. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: library operating systems for the cloud. In V. Sarkar and R. Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, pages 461–472. ACM, 2013. doi: 10.1145/2451116.2451167. URL https://doi.org/10.1145/2451116.2451167.

[13] J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, page 37–51, New York, NY, USA, 1985. Association for Computing Machinery. ISBN 0897911474. doi: 10.1145/318593.318606. URL https://doi.org/10.1145/318593.318606.

[14] B. Montagu. *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, 2010. URL https://tel.archives-ouvertes.fr/tel-00550331.

[15] B. Montagu and D. Rémy. Modeling Abstract Types in Modules with Open Existential Types. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL'09)*, pages 354–365, Savannah, GA, USA, Jan. 2009. ISBN 978-1-60558-379-2. doi: http://doi.acm.org/10.1145/1480881.1480926.

[16] G. Radanne, T. Gazagnaire, A. Madhavapeddy, J. Yallop, R. Mortier, H. Mehnert, M. Perston, and D. Scott. Programming unikernels in the large via functor driven development, 2019.

[17] A. Rossberg. 1ML - Core and modules united. *J. Funct. Program.*, 28:e22, 2018. doi: 10.1017/S0956796818000205. URL https://doi.org/10.1017/S0956796818000205.

[18] A. Rossberg, C. Russo, and D. Dreyer. F-ing modules. *Journal of Functional Programming*, 24(5):529–607, Sept. 2014. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796814000264. URL https://www.cambridge.org/core/product/identifier/S0956796814000264/type/journal_article.

[19] C. V. Russo. First-Class Structures for Standard ML. *Nord. J. Comput.*, 7(4):348–374, 2000.

[20] C. V. Russo. Types for modules. *Electronic Notes in Theoretical Computer Science*, 60:3–421, 2004. ISSN 1571-0661. doi: https://doi.org/10.1016/S1571-0661(05)82621-0. URL https://www.sciencedirect.com/science/article/pii/S1571066105826210.

[21] C.-C. Shan. Higher-order modules in system f^{ω} and haskell. 01 2004.

[22] Z. Shao. Transparent modules with fully syntactic signatures. In D. Rémy and P. Lee, editors, *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999*, pages 220–232. ACM, 1999. doi: 10.1145/317636.317801. URL https://doi.org/10.1145/317636.317801.

[23] TyXML. *TyXML*. http://ocsigen.org/tyxml/, 2017.

[24] L. White, F. Bour, and J. Yallop. Modular implicits. In *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014.*, pages 22–63, 2014. doi: 10.4204/EPTCS.198.2. URL https://doi.org/10.4204/EPTCS.198.2.