Fulfilling OCaml modules with transparency

ANONYMOUS AUTHOR(S)

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 $\mathcal{F}^\omega$, we introduce an intermediate system, called $\mathcal{M}^\omega$, that mediates between the source language and $\mathcal{F}^\omega$, combining the convenient path-based syntactic notation of ML modules with the precise and well-established type-theory of $\mathcal{F}^\omega$. From the $\mathcal{M}^\omega$ system, we elaborate terms into $\mathcal{F}^\omega$ using the new technique of transparent existential types, which ensures type soundness.

ACM Reference Format:

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 encouraging interactions through reduced APIs, this reduces complexity and increases code sharing. A component may be, for instance, the implementation of a reusable, often polymorphic, data-structure. Implementation details, such as internal invariants, are 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. 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 as both 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 [11] where modules and functors are assembled on demand using a DSL [15].

However, despite the successes and the interest of the community regarding ML modules, giving them a formal type-theoretic definition and establishing its properties, especially type soundness, has proven to be a difficult task. In the case of OCaml, the foundational works of Leroy [8, 9], have not been extended to include the numerous new features. The current situation is a module system 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 [23], lacking a specification is a show-stopper, as it could have unforeseen breaking changes.
Combining ideas from many years of research (see §5 for related works), a particularly successful and elegant approach to model ML module systems is the translation of ML-modules into $\text{F}^{\omega}$, the high-order polymorphic lambda calculus. Extending the approach of Russo [19], who provided a type system for modules using $\text{F}^{\omega}$ types, a milestone was done by Rossberg et al. [17], who gave a full elaboration of both types and terms of a significant subset of SML into $\text{F}^{\omega}$. The elaboration showed that ML modules can be encoded as $\text{F}^{\omega}$ terms, which ensure soundness of the system, and correspondingly, ML-signatures can thus be understood as $\text{F}^{\omega}$-types.

We build on the insights of their work, which we adapt and improve for an OCAML-like language (extended with a new form of transparent ascription). To separate the concerns, we exhibit an intermediate type system, called $M^{\omega}$, with a new signature syntax that resembles the usual OCAML syntax but extended with $\text{F}^{\omega}$ quantifiers, à la Russo [19]. The $M^{\omega}$ system is central to our work: it provides an up-to-date specification of OCAML modules; it could 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 (infamous) signature avoidance problem.

Signatures of $M^{\omega}$ are nothing else but syntactic sugar for $\text{F}^{\omega}$ types. We also give an elaboration à la Rossberg et al. [17], which ensures type soundness. 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 $\text{F}^{\omega}$ could also help conduct proofs of modular programs—a topic of quickly growing importance. Several projects that aim at analyzing or proving OCaml programs, such as Salto and Gospel, have excluded programs with modules by lack of a specification of the module system. Moreover, the simplification of the encoding into $\text{F}^{\omega}$ in the applicative case should also benefit to the unification of the module and core languages proposed by Rossberg [16] and, perhaps with a few helper constructs, let modules be directly programmable into the core language.

Our contributions are:

- A simple specification of a large subset of OCAML modules in $M^{\omega}$, including both applicative and generative functors, using ML-style signature syntax but explicit $\text{F}^{\omega}$-style quantifiers.
- The specification of a new form of transparent ascription (called concrete ascription) to subsume and fix the module aliasing system of OCAML.
- A source-to-source encoding of aliasing (a key to abstraction safety) relying solely on type abstraction—removing the need from a primitive treatment by the type system.
- The introduction of transparent existential types in $\text{F}^{\omega}$, a weaker form of existential types that allows their lifting through arrows 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 $\text{F}^{\omega}$.

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 Momega, which bridges the gap between source signatures and $\text{F}^{\omega}$ types. At the end of this section, we discuss the signature avoidance problem and the strategies for a reverse translation from $M^{\omega}$-signatures back to OCAML ones. In §4, we present the elaboration of modules into $\text{F}^{\omega}$ terms as an extension of the $M^{\omega}$ typing. To this end, we introduce using transparent existential

1https://salto.gitlabpages.inria.fr/index.html
2https://github.com/ocaml-gospel/gospel
module Complex = struct
  type t = int * int
  let zero = (0, 0) let one = (1, 0)
  let add u v = ...
end

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

module CRing = (Complex : Ring)

module Polynomials = functor (R : Ring) → struct
  type t = R.t list
  let zero = [] let one = [R.one]
  let add = ...
end
module CX = Polynomials(Complex)
module PolynomialsXY (R : Ring) = Polynomials(Polynomials(R))

Fig. 1. Basic modularity.

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 granularity of OCaml leads to tracking aliasing at the level of modules and the notion of module identity. To treat this module identity properly, we propose an extension of the signature language with transparent signatures. Finally, we explain the signature avoidance problem.

2.1 Basic ML modularity

An introductory example is given in Figure 1. Modules are created by gathering term and type definitions in a structure, which can be named, as illustrated by module Complex. Definitions inside a structure are called bindings 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. The outside view of a module can be restricted to protect internal invariants by an explicit ascription to a given interface (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 available, but its implementation as a pair of integer is hidden. A module can also be parameterized by another one by defining a functor: Polynomials takes any implementation R satisfying the Ring interface and returns an implementation of the ring of polynomials over R. The body of the functor is polymorphic with respect to the abstract type fields of its argument, and thus, does not depend on their actual implementations. Functors can then be called and composed: Polynomials can be applied to modules satisfying Ring such as Complex and CRing (lines 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

Functors are often used as reusable units and pieces of libraries. A program might then contain several applications of the same 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
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 argument to a functor application, i.e., when should two functor applications produce compatible abstract types?

A first option is to consider modules to be similar when they have the same type fields. This is called static equivalence \[4, 18, 21\]. This solution is type-safe, as the actual implementation of the abstract types produced by the functor can only depend statically on its parameters, thus only on its \textit{statically known} type fields. Moscow ML uses this static equivalence for its applicative functors.

However, relying only on type fields can make two functor applications compatible while they actually have different internal invariants. In the example on Figure 2, the internal representation of sets \( \text{type } t = \text{E.t list} \) only depends on the type fields of the parameter, but the code maintains

\begin{verbatim}
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 *)
\end{verbatim}

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

\begin{verbatim}
8 module OrderedSets (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 *)
\end{verbatim}

(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. Example of generative and applicative functors.

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 \textit{generates} two incompatible modules, with incompatible abstract types. A \textit{generative functor} is therefore used when the module is thought of as a \textit{sub-program}, dynamically parameterized by its argument. Its application can create its own internal state, produce effects, or dynamically choose the internal representation of its abstract types. \textit{Generativity} protects the internal invariants by making two successive applications incompatible. 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 “()”\textsuperscript{3}.

Conversely, an \textit{applicative functor} may be used for pure\textsuperscript{4} modules that have static definitions of the internal representations of their abstract types: calling an applicative functor twice with the \textit{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 typically used when the module is thought of as a \textit{library}, statically parameterized by its argument, that provides generic functionalities (such as hash-maps\textsuperscript{5}, sets, lists, etc.).

\textsuperscript{3}We expand on the reasons behind this choice in §3.1.

\textsuperscript{4}This is so far not checked in OCaml. It is left to the user’s responsibility to mark impure functors as generative.

\textsuperscript{5}Hashtbl.Make is pure as it does not produce a new hash table itself, even through it contains impure functions.

\textsuperscript{6}This property, introduced by Harper et al. [6], is known as phase-distinction: types components of a module can only depend on types of other modules, not on values.
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 idea that type abstraction also protects arbitrary invariants is called abstraction safety. As pointed out in \textit{F-ing}, 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. \textsc{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 \cite{leroy} and has then been extended to a static tracking of module aliases.

### 2.4 Aliasing and subtyping

To allow tracking of aliasing, \textsc{Ocaml} offers a notion of module alias\(^7\): a module can be typed as an alias of another module to mean that they share the same identity. This is done by extending the signature language with the alias signature (\( = X \)) to tell that this is an alias of \( X \). Keeping as much aliasing information as possible allows for more type equalities when using applicative functors.

However, module aliases are incompatible with subtyping by coercion, which can change the memory representation by dropping and reordering fields: considering a module as an alias of (hence interchangeable with) another module while its memory representation is different would be unsound. Yet, coercive subtyping is commonly used for modules systems (and \textsc{Ocaml} in particular), which relies on types to perform fast, static access to module fields.

Thus, aliases cannot be maintained through subtyping, which happens at ascription and at functor application. Consequently, aliases between a functor parameter and its body must be removed, which limits their usefulness. This problem induced a set of ad-hoc, fairly complex restrictions in \textsc{Ocaml} that can actually be bypassed\(^8\). For instance, the functor \(( Y : S ) \rightarrow Y \) cannot be given the type \(( Y : S ) \rightarrow ( = Y ) \). 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 \(( M :- S )\), already present in \textsc{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 expressiveness in \textsc{Sml} as a similar result could be obtained via a usual ascription \(( M : S ' )\) with a signature \( S ' \) where all type equalities have been made explicit. A proposal for \textsc{Ocaml}\(^9\) 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). Transparent ascription à la \textsc{Sml} \(( M <: S )\), hereafter called concrete ascription to avoid the confusion with other notions of transparency, is just syntactic sugar for \(( M : ( = P < S ) )\), i.e., a “normal” ascription of \( P \) with a concrete signature \(( = P < S )\).

Thanks to concrete signatures, aliasing information can be preserved through subtyping operations (explicit through ascription or implicit at functor calls). Figure 3 demonstrate a code

\(^7\)
\textsc{Ocaml} actually offers two distinct notions of aliases, which are respectively present or absent at runtime. Here, we only consider present aliases.

\(^8\)See the following issues: \textsc{Ocaml} #7818, \textsc{Ocaml} #2051, \textsc{Ocaml} #10435, \textsc{Ocaml} #10612 and \textsc{Ocaml} #11441.

\(^9\)\textsc{Ocaml} #10612. Transparent ascription is written \(( P :-: S )\) in the \textsc{Ocaml} #10612 proposal.
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 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.

pattern, present in the OCaml ecosystem [22], 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 [23], 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 expressiveness of the module and signature languages: the reachable space of possible module expressions is larger than the describable space of signatures: some modules simply cannot be described by a signature. This mismatch is caused by the interaction of three mechanisms. First, type abstraction creates new types that are only compatible with themselves (and their aliases). Then, sharing abstract types between modules, which is essential for module interactions, produces inter-module dependencies. Finally, hiding type or module components (either by a projection or by implicit subtyping at a functor application) can break such dependencies by removing type aliases from scope while they are still being referenced. For instance, an abstract type t can be hidden while a value of type t list is still in scope. An example of such pattern is given in Figure 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 $M^\omega$ 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 typechecking 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 $M^\omega$ APPROACH

In this section, we present the typing system $M^\omega$ that covers the set of features informally explained in the previous section without suffering from the signature avoidance problem. $M^\omega$ distinguishes between source signatures written by the user and $M^\omega$-signatures used for typechecking both module expression and source signatures. Canonical signatures use explicit binders (existential, universal, lambda) as in $F^\omega$ (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 $F^\omega$. We start with the grammar of source expressions (§3.1) and an overview of $M^\omega$ (§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).
Path and Prefix

\[ P ::= Q.X \] (Access) \[ | Y \] (Functor argument) \[ | P(P) \] (Applicative application) \[ | Q ::= A | P \] (Prefix) \[ | () \rightarrow S \] (Generative) \[ | (Y : S) \rightarrow S \] (Applicative) \[ | \text{sig}_A \overline{B} \text{ end} \] (Structural signature) \[ | (\_ \_ < S) \] (Concrete signature)

Module Expression

\[ M ::= P \] (Path) \[ | M.X \] (Projection) \[ | (P : S) \] (Ascription) \[ | P() \] (Generative application) \[ | (Y : S) \rightarrow M \] (Applicative) \[ | \text{struct}_A \overline{B} \text{ end} \] (Structure)

Source signatures

\[ S ::= Q.T \] (Module type) \[ | () \rightarrow S \] (Generative) \[ | () \rightarrow \text{sig}_A \overline{B} \text{ end} \] (Structural signature) \[ | (\_ \_ < S) \] (Concrete signature)

Source declarations

\[ D ::= \text{val} \ x : u \] (Value) \[ | \text{type} \ t = u \] (Type) \[ | \text{module} X : S \] (Module) \[ | \text{module type} \ T = S \] (Module type) \[ | I ::= x \mid t \mid X \mid Y \mid T \] (Any identifier)

Core language

\[ e ::= Q.x \] (Qualified value) \[ | \_ \_ \] (Other expression) \[ | \_ \_ \] (Other type)

Canonical Types

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

Environment

\[ \Gamma ::= \emptyset \] (Empty) \[ | \Gamma, \alpha \] (Abstract type) \[ | \Gamma, (Y : C) \] (Functor Argument) \[ | \Gamma, (A.I : D) \] (Declaration)

Opacity

\[ \phi ::= \psi \] (Transparent) \[ | \phi \] (Opaque)

Canonical signatures

\[ C ::= \text{sig} \overline{D} \text{ end} \] (Structural signature) \[ | \forall \alpha. C \rightarrow C \] (Applicative functor) \[ | () \rightarrow \exists \alpha. C \] (Generative functor)

Canonical declarations

\[ D ::= \text{val} \ x : \tau \] (Values) \[ | \text{type} \ t = \tau \] (Types) \[ | \text{module} X : C \] (Modules) \[ | \text{module type} \ T = \lambda \overline{\alpha} C \] (Module types)

3.1 The source language

The source grammar, given on Figure 5, is built on top of a core language of expressions \( e \) and types \( u \) which are mostly left abstract. We only consider value identifiers \( x \) and type identifiers \( t \), extended with qualified values \( Q.x \) and qualified types \( Q.t \); these are the only way for the core level to access the module level. The abstract syntax of module expressions and signatures is rather standard and mostly follows the current OCaml syntax. There are a few minor technical choices:

- Module-related meta-variables use typewriter uppercase letters, \( M, S \), etc., while lowercase letters are used for expressions and types of the core language. Lists are written with an overhead bar. Identifiers \( I \) and paths \( P \) use a standard font.
- In order to simplify the treatment of scoping and shadowing, we introduce self-references, ranged over by letter \( A \), in both structures and signatures. They are used to refer to the current
object; their binding occurrence appears as a subscript to the structure or signature they belong to (struct\_A ... end), so that self-references can freely be renamed. They are not present in OCAML and should be thought of as being added by a first pass before typing. We explain how they 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., type \( t = A \cdot t \) 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 \( I : B \) and \( I : D \), with the identifier extracted, i.e., a type declaration type \( t = u \) can be written as \( t : \) type \( u \), a module binding module \( X : M \) can be written as \( X : \) module \( M \).
- 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 [8] and Russo [19], 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 \( G() \cdot x \), is syntactically ill-formed, as paths do not contain the unit argument ()). By contrast, a qualified access inside an applicative functor application \( F(X) \cdot t \) 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:

\[
M(M') \triangleq (\text{struct}_A \ module F = M \ module X = M' \ module Res = A \cdot F(A \cdot X) \ end) \cdot Res
\]
\[
M() \triangleq (\text{struct}_A \ module G = M \ module Res = A \cdot G() \ end) \cdot Res
\]

In our setting, the only expression that can “cause” signature avoidance is the projection. 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 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:

\[
(Y : S) () \rightarrow M \triangleq (Y : S) \rightarrow (() \rightarrow M)
\]

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: \( (Y : S1)(Y : S2)() \rightarrow M \) is fully generative (every instance is new), while \( (Y : S1)()(Y : S2) \rightarrow M \) is generative with regard to the first argument, then applicative with regard to the second one.
3.2 Canonical system overview

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

We annotate existential types with an opacity flag $\diamond$ which can be either opaque ($\Box$) or transparent $^\omega$ ($\forall$). A transparent existential ensures that its witness is actually statically known, while by default, the witness may depend on a dynamics choice, in and we use an opaque existential type.

This distinction will be used for the typing of applicative and generative functors.

Canonical signatures $\mathcal{C}$ use explicit quantification over abstract types $\bar{\tau}$. This quantification is universal in the signature of an applicative functor $\forall \mathcal{C} \rightarrow \mathcal{C}$, and existential in the signature of the body of a generative functor $\exists \tau \mathcal{C}$ ($\forall$ indicates that this existential is opaque, i.e., a standard existential type). Module types $\lambda \bar{\tau}.\mathcal{C}$ are parametric $^\omega$ in each type variable $\alpha$, 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 $\alpha$, 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 $\text{wf}(\Gamma)$ that also check that identifiers appear at most once—in addition to recursively checking well-formedness of all bindings in $\Gamma$. 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 types 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. However, local definitions that shadow a definition made in an enclosing module are authorized, as they would have a different self-reference prefix and are therefore considered as two different bindings.

The $M^\omega$ 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 $M^\omega$ can be illustrated with the typechecking of signatures $\Gamma \vdash S : \lambda \bar{\tau}.\mathcal{C}$, which translates a source signature $S$ into its $M^\omega$ counterpart $\lambda \bar{\tau}.\mathcal{C}$, making the set of type parameters $\bar{\tau}$ explicit. The set of rules is given in Figure 7 and discussed below.

Declarations. Typechecking of signatures uses a helper judgment for typechecking declarations $\Gamma \vdash_A D : \lambda \bar{\tau}.\mathcal{D}$ 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:

$$\Gamma \vdash_A (\text{type } t = A.I) : \lambda \alpha.\text{(type } t = \alpha)$$  \hspace{1cm} (M-Typ-Decl-TypeAbs)

10This notion of transparency is unrelated to concrete ascription (called transparent ascription in SML).

11Here, we follow Russo [19] rather than [17] and use a lambda quantifier for signatures.

12This is not restrictive as this can always be solved by appropriate renaming.

, Vol. 1, No. 1, Article . Publication date: October 2023.
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 \( \alpha \) must be lifted to enclose the whole region where \( \alpha \) is accessible.

Thus, in Rule M-Typ-Decl-Mod for typing a submodule declaration module \( X : S \), the set of abstract types \( \overline{\alpha} \) quantified in the submodule \( M^{\overline{\alpha}} \)-signature \( \overline{\lambda \alpha.\overline{C}} \) of \( S \) is extruded in front of the \( M^{\overline{\alpha}} \)-signature of the declaration \( \overline{\lambda \alpha.(\text{module } X : C)} \). In addition, in Rule M-Typ-Decl-Seq for merging a list of declarations \( D_0, \overline{D} \), the types \( \overline{\alpha}_0 \) introduced by the first declaration \( D_0 \) are put in the context for typing the rest of declarations \( \overline{D} \) and both sets of abstract types \( \overline{\alpha}_0 \) and \( \overline{\alpha} \) are merged together in front of the list of \( M^{\overline{\alpha}} \)-declarations \( D_0, \overline{D} \). 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 \( D_0 \), seen as \( A.I : D_0 \).

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 \( M^{\overline{\alpha}} \) signatures do not have a counterpart for module types \( Q.T \) in source signatures. For a concrete signature \((= P < S)\), the signature \( S \) is elaborated into an \( M^{\overline{\alpha}} \)-signature \( \overline{\lambda \alpha.\overline{C}} \) and checked against the \( M^{\overline{\alpha}} \)-signature \( C \) of path \( P \) (Rule M-Typ-Sig-Con). The result signature is not \( C' \) but \( C'[\overline{\alpha} \mapsto \overline{\tau}] \) after applying the matching substitution \([\overline{\alpha} \mapsto \overline{\tau}] \) to \( C' \). 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 \( \exists^\forall \overline{\alpha}.\overline{C} \) 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 \( \forall \overline{\alpha}.\overline{C} \to \exists^\forall \overline{\beta}.\overline{C}' \), where all applications would produce new types, and neither of the form \( \lambda \beta.\forall \overline{\alpha}.\overline{C} \to C' \), where all applications would share the same types regardless of their argument.

The solution, introduced by Biswas [1] is to use higher-order abstract types \( \overline{\beta}' \) applied to the universally quantified variables \( \overline{\alpha} \). This gives a signature of the form \( \lambda \overline{\beta}'.\forall \overline{\alpha}.\overline{C} \to C'[\overline{\beta} \mapsto \overline{\beta}'(\overline{\alpha})] \) which can be seen in the rule M-Typ-Sig-AppFct.
3.4 Subtyping

The subtyping judgment \( \Gamma \vdash C < C' \) (with the helper judgment \( \Gamma \vdash \mathcal{D} < \mathcal{D}' \)) checks that a signature \( C \) is more restrictive than a signature \( C' \), meaning that the former has more fields and introduces less abstract types. It ensures that a module with signature \( C \) can be coerced into a module of signature \( C' \). 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:

\[
\begin{align*}
\text{M-Sub-Sig-Struct} & : \frac{\mathcal{D}_0 \subseteq \mathcal{D}}{\Gamma \vdash \text{sig} \mathcal{D}_0 \text{ end} < \text{sig} \mathcal{D}' \text{ end}} \\
\text{M-Sub-Sig-GenFct} & : \frac{\Gamma \vdash C < C'[\pi \mapsto \tau]}{\Gamma \vdash \exists \pi. \mathcal{C}'}
\end{align*}
\]

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 between existential types \( \Gamma \vdash \exists \pi. \mathcal{C} < \exists \pi'. \mathcal{C}' \), which in turn amounts to finding an instantiation \( [\pi \mapsto \tau] \) of the abstract types so that \( C \) is a subtype of \( C'[\pi \mapsto \tau] \). 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. [17], and shown decidable for certain pairs of signatures \((C, C')\) 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 \( C' \) that are the elaboration of source signatures. One exception would be the construct \text{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 \text{module type} of \( P \) to cases where the inferred signatures of \( P \) is indeed a source-like one, which is called an \text{explicit} signature in [17]. The same argument would apply to the \( M^{\alpha} \) system.

3.5 Typechecking of module expressions

Typechecking of expressions \( \Gamma \vdash M : \exists \pi . \mathcal{C} \) infers the \( M^{\alpha} \)-signature \( \exists \pi . \mathcal{C} \) of a source module \( M \). 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 $\Gamma \vdash^\phi M : \exists^\phi \overline{\pi}.C$ 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 $\Gamma \vdash M : \exists^\phi \overline{\pi}.C$ or $\Gamma \vdash M : \exists^\diamond \overline{\pi}.C$ when $\overline{\pi}$ is empty, we actually mean $\Gamma \vdash^\phi M : C$ and $\Gamma \vdash^\diamond M : C$. The same convention applies to typing rules for bindings $\text{M-Typ-Decl}^\ast$ which can be found in §4.2 of the supplementary materials. Typing rules for expressions $\text{M-Typ-Mod}^\ast$ are given on Figure 8.

Skolemization. The need for two modes of typing comes from the treatment of applicative functors, specifically the rule $\text{M-Typ-Mod-AppFct}$. 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 [19]. 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 §4.4.

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\textsuperscript{13}, 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 $\text{M-Typ-Mod-AppFct}$) and opaquely when it is generative (Rule $\text{M-Typ-Mod-GenFct}$). Rule $\text{M-Typ-Bind-Seq}$ forces all components of a structural signature to have the same opacity:

\[
\text{M-Typ-Bind-Seq} \quad \frac{\Gamma \vdash A \, B_0 : \exists^\phi \overline{\pi}_0 . D \quad \Gamma, \overline{\pi}_0, A.I : D \vdash \overline{\beta} : \exists^\phi \overline{\pi}, \overline{\eta} \quad \Gamma \vdash \overline{\beta} : \exists^\phi \overline{\pi}, \overline{\eta} \, D, \overline{\eta} \!}{\Gamma \vdash A \, B_0 : \exists^\phi \overline{\pi}_0 . D, \overline{\eta} \, D, \overline{\eta}}
\]

\[
\text{M-Typ-Bind-Let} \quad \frac{\Gamma \vdash \overline{\beta} : \exists^\phi \overline{\pi}, \overline{\eta} \quad \Gamma \vdash \overline{\beta} e : r \quad (\text{let } x = e : (\text{val } x : r)}
\]

We also rely on a core-language expression typing judgment\textsuperscript{14} $\Gamma \vdash^\diamond e : r$ equipped with a mode that tracks the presence of effects\textsuperscript{15}. When typing a value field, the mode is propagated via an empty existential (Rule $\text{M-Typ-Bind-Let}$). Signatures can also be downgraded from transparent to opaque via subsumption (Rule $\text{M-Typ-Mod-Seal}$). All other rules are agnostic of the typing mode. With the convention that $\text{M-Typ-Mod-Seal}$ 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 $\text{M-Typ-Mod-Ascr}$ for signature ascription ($P : S$) has some resemblance with Rule $\text{M-Typ-Sig-Con}$ for typechecking of concrete signatures ($= P < S$): in both cases, we check that the $M^\omega$ signature of $P$ is a subtype of the $M^\omega$-signature $\lambda \overline{\pi}.C'$ of $S$. By contrast, however, we here drop the matching substitution in the result signature $\exists^\phi \overline{\pi}.C'$ and instead introduced the abstract types $\overline{\pi}$, transparently. In particular, when $S$ is concrete, i.e., $\overline{\pi}$ is empty, no abstract type is actually introduced. That is, concrete ascription ($P < S$), which is syntactic sugar for ($P : (= P < S)$), i.e., the opaque ascription of $P$ to the concrete signature ($= P < S$), behaves as expected, filtering out components of $P$ as prescribed by $S$ but without creating new abstract types.

\textsuperscript{13}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.

\textsuperscript{14}This judgment is trivially extended by rules for accessing module paths; the added rules are given in §4.2 of supplementary materials.

\textsuperscript{15}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.

, Vol. 1, No. 1, Article . Publication date: October 2023.
Tagging

\[\text{Tag } \{M\} \triangleq \text{struct}_A \text{ module } \text{Val} = M \text{ type } \text{id} = A \text{.id end} \]

\[\text{Tag } \{S\} \triangleq \text{sig}_A \text{ module } \text{Val} : S \text{ type } \text{id} = A \text{.id end} \]

Paths

\[\text{[A.X]} \triangleq A.X \quad \text{[P.X]} \triangleq [P].\text{Val.X} \]

\[\text{[Y]} \triangleq Y \quad \text{[P(P')] } \triangleq [P].\text{Val([P'])} \]

Module expressions

\[[M.X] \triangleq [M].\text{Val.X} \quad [P()] \triangleq [P].\text{Val()} \]

\[\text{[(P : S)] } \triangleq ([P] : [S]) \]

\[\text{[}() \rightarrow [M] \text{]} \triangleq \text{Tag } (() \rightarrow [M]) \]

\[\text{[(Y : S) } \rightarrow [M] \text{]} \triangleq \text{Tag } ((Y : [S]) \rightarrow [M]) \]

\[\text{[struct}_A \text{ [B] end} \triangleq \text{Tag } [\text{struct}_A \text{ [B] end} \]

Signatures

\[[A.T] \triangleq A.T \quad [P.T] \triangleq [P].\text{Val.T} \]

\[[ (= P < S) ] \triangleq (= [P] < [S]) \]

\[\text{[}() \rightarrow [S] \text{]} \triangleq \text{Tag } (() \rightarrow [S]) \]

\[\text{[(Y : S_a) } \rightarrow [S] \text{]} \triangleq \text{Tag } ((Y : [S_a]) \rightarrow [S]) \]

\[\text{[sig}_A \text{ [B] end} \triangleq \text{Tag } [\text{sig}_A \text{ [B] end} \]

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.

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 there 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 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^{\omega} signatures only consist of the use of a common abstract type that has previously been lifted so as to be in scope. Thus, M^{\omega} 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 \(\pi^{\omega}\) of abstract types \(\pi\) that are free in the submodule signature \(C\) to ensure that it has no dangling dependency. This rule performs garbage collection of unused abstract types. It would also be sound not to enforce it, by letting \(\pi^{\omega}\) range between \(fv(C) \cap \pi\) and \(\pi\) (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 [19] and F-ing. To obtain abstraction safety, Rossberg et al. [17] 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.

, Vol. 1, No. 1, Article . Publication date: October 2023.
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).**
\[
\Gamma \vdash [M_1] : \text{sig module } Val : C_1 \text{ type id }=\tau \text{ end }
\]
\[
\Gamma \vdash [M_2] : \text{sig module } Val : C_2 \text{ type id }=\tau \text{ end }
\]
\[\implies \exists C_0, \quad \Gamma \vdash C_0 < C_1 \quad \Gamma \vdash C_0 < C_2
\]

**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 \(M^\omega\) 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. \(\square\)

### 3.7 Revisiting the signature avoidance problem

In this section we present some insights about the reverse translation from \(M^\omega\) signatures back
into the source OCAML syntax, which we call anchoring. This translation is not complete because
the source syntax has the signature avoidance problem. However, using \(M^\omega\) signatures, we can
precisely describe the different signature avoidance cases, from which we extract three guidelines
for a reverse translation algorithm.

#### 3.7.1 Abstract type fields

A first key insight is the difference in the source syntax between
the declaration of a concrete type (type \(t = u\)) and that of an abstract type (type \(t = A.t\)). An
abstract type declaration type \(t = A.t\) in covariant position effectively creates a new abstract type
(introducing an existential quantifier in \(M^\omega\)) and adds a type field \(t\) to the signature, while a concrete
type definition type \(t = u\) in covariant position only introduces structural information—adding a
field \(t\) to refer to the existing type \(u\). By contrast, \(M^\omega\) 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 \(\alpha\) is in a type declaration type \(t = \alpha\), called the anchoring point for the type \(\alpha\).

#### 3.7.2 Module identities

In an un-tagged setting, source signatures can only express identity sharing via concrete signatures (\(P < S\)), 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.

#### 3.7.3 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, \(M^\omega\) signatures can express sharing of a higher-order abstract type between functors
with arbitrary domains. Let us consider the following \(M^\omega\) signature, resulting from a projection
where the functor that introduced \(\varphi\) became unreachable while two uses of \(\varphi\) remain:

\[
\exists \varphi \cdot \text{module } M : \text{sig module } \varphi F : \forall \alpha.\mathcal{C} \rightarrow \text{sig type } t = \varphi(\alpha) \text{ end }
\]
\[
\text{module } F' : \forall \alpha.\mathcal{C}' \rightarrow \text{sig type } t = \varphi(\alpha) \text{ end end}
\]
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 $S'$:

\[
\text{module } M : \text{sig} \begin{align*}
\text{module } F &: (Y : S) \rightarrow \text{sig}_A \text{ type } t = A.t \end{align*} \text{ end} \\
\text{module } F' &: (Y' : S') \rightarrow \text{sig}_A \text{ type } t = F(Y).t \text{ end} \\
\]

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

4 **THE FOUNDATIONS: $\text{F}^\omega$ ELABORATION**

The $\text{M}^\omega$ 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 $\text{F}^\omega$. Yet, the encoding of module expressions and signatures in $\text{F}^\omega$ served as a basis for the design of $\text{M}^\omega$ 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 $\text{F}^\omega$ or how to conduct proofs of modular programs.

The encoding is largely based on the work of Rossberg et al. [17], 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.

4.1 **$\text{F}^\omega$ with kind polymorphism**

We use a variant of explicitly typed $\text{F}^\omega$ 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,
\[\begin{align*}
\zeta &:= \star | \kappa \mid \zeta \to \zeta \\
\kappa &:= \zeta \mid \forall \kappa. \kappa \mid \kappa \to \kappa \\
\tau &:= \alpha \mid \tau \to \tau \mid \{\ell : \tau\} \mid \forall (\alpha : \kappa). \tau \mid \exists (\alpha : \kappa). \tau \mid \lambda (\alpha : \kappa). \tau \mid \tau \tau \mid \forall \kappa. \tau \mid \Lambda \kappa. \tau \mid \tau \zeta \mid () \\
e &:= x \mid \lambda (x : \tau). e \mid e \mid \Lambda (\alpha : \kappa). e \mid e \tau \mid \Lambda \kappa. e \mid e \zeta \mid e \odot e \mid \{\ell = e\} \mid e, \ell \\
&\mid \text{pack } \langle \tau, e \rangle \text{ as } \exists \gamma (\alpha : \kappa). \tau \mid \text{unpack } \langle \alpha, x \rangle = e \text{ in } e \mid () \\
\Gamma &:= \cdot \mid \Gamma, \kappa \mid \Gamma, \alpha : \kappa \mid \Gamma, x : \tau
\end{align*}\]

Fig. 10. Syntax of \(F^\omega\)

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 \(F^\omega\)-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.

The syntax of \(F^\omega\) is given in Figure 10. Typing rules are standard and available in §5 of supple-
mentary materials. Type equivalence, defined by \(\beta\eta\)-conversion and reordering of record fields, is
also standard and omitted. We use letters \(\tau\) and \(e\) to range over types and expressions to distinguish
them from the core language types \(\tau\) and expressions \(e\), even though these should actually be seen
as a subset of \(\tau\) and \(e\). We consider \(F^\omega\) to be explicitly typed and explicitly kinded. As a conve-
nption, 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 \(F^\omega\) terms. We write \(k\) for kind variables, \(\alpha\) and \(\beta\) for type variables of any kind, and
\(\varphi\) and \(\psi\) for type variables known to be of higher-order kinds. Application of expressions \(e, \zeta\) and
types \(\tau, \eta\) to kinds are restricted to small kinds \(\zeta\). In expressions and type expressions, we actually
write kinds \(\kappa\) (and kind abstraction \(\Lambda \kappa.\)) 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 \(x = e_1\) in \(e_2 \triangleq (\lambda (x : \_). e_2)\) \(e_1\) and \(n\)-ary pack and unpack operators
defined in the following way:

\[
\begin{align*}
\text{pack } \langle \tau, e \rangle &\triangleq \exists \gamma (\alpha : \kappa). \sigma & \text{pack } \langle \tau, e \rangle &\text{ as } \exists \gamma (\alpha : \kappa) [\alpha \mapsto \tau] \text{ as } \exists \gamma (\alpha : \kappa) \sigma \\
\text{unpack } \langle \alpha \beta, x \rangle &\triangleq e_1 \text{ in } e_2 & \text{unpack } \langle \alpha, x \rangle &\triangleq e_1 \text{ in } e_2 \\
\text{pack } \langle \emptyset, e \rangle &\triangleq e & \text{unpack } \langle \emptyset, x \rangle &\triangleq e_1 \text{ in } e_2 \triangleq \text{let } x = e \text{ in } e_2
\end{align*}
\]

4.2 Encoding of signatures

Canonical signatures are actually \(F^\omega\) types with some syntactic sugar. We assume a collection \(l_I\) of
record labels indexed by identifiers \(I\) of the source language. Structural signatures \(\text{sig } D\) end are
just syntactic sugar for record types \(\{ D \}\).

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).

\[
\langle \tau : \kappa \rangle \triangleq \Lambda (\varphi : \kappa \to \star). \lambda (x : \varphi \tau). x \quad \text{Term} \\
\langle \tau : \kappa \rangle \triangleq \forall (\varphi : \kappa \to \star). \varphi \tau \to \varphi \tau \quad \text{Type}
\]
The type $\tau$ is used as argument of a higher-kind type operator $\varphi$ 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):

$$\text{val } x : \tau = \ell_x : \tau \quad \text{module } X : \mathcal{C} = \ell_X : \mathcal{C}$$

$\text{type } t = \tau \iff \ell_t : \langle \langle \tau \rangle \rangle \quad \text{module type } T = \lambda \pi. \mathcal{C} \iff \ell_T : \langle \langle \lambda \pi. \mathcal{C} \rangle \rangle$

4.3 Sharing existential types by repacking

While the correspondence between $M^\omega$ signatures and $F^\omega$ types is straightforward, the actual encoding of module expressions as $F^\omega$ 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 in for handling a term with an abstract type is the 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 $F^\omega$ only provides non-dependent records.

This observation was at the core of the design of open existential types [13] and of recursive type generativity [3]. Here, in order to stay in plain $F^\omega$, 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 $\text{repack}^\varphi \langle \alpha, x \rangle = e_1$ 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 $\alpha$ and the type of $e_2$:

$$\text{repack}^\varphi \langle \alpha, x \rangle = e_1 \text{ in } e_2 \iff \text{unpack } \langle \alpha, x \rangle = e_1 \text{ in } \text{pack } \langle \alpha, e_2 \rangle \text{ as } \exists^\alpha \bar{\alpha}.$$  

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 $\bar{\alpha}$:

$$\text{lift}^\varphi \langle \bar{\alpha}, x = e_1 @ e_2 \rangle \iff \text{repack}^\varphi \langle \bar{\alpha}, x \rangle = e_1 \text{ in } \text{repack } \langle \bar{\beta}, x_2 \rangle = e_2 \text{ in } x_1 @ x_2.$$  

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

$$\Gamma \vdash e_1 : \exists^\alpha \bar{\alpha}. \{ \ell_1 : \tau_1 \} \quad \Gamma, \bar{\alpha}, x : \{ \ell \bar{\alpha} : \tau_1 \} \vdash e_2 : \exists^\bar{\beta}. \{ \ell \bar{\beta} : \tau_2 \} \quad \ell_1 \neq \ell_2$$  

4.4 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 $M^\omega$, is not definable in $F^\omega$.

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 [20]. 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 $\mathcal{M}^\omega$ system, but we tweak the definition of existential types to make their lifting through universal types definable. Namely, we introduce transparent existential types, written $\exists^\tau (\alpha : \kappa). \sigma$ to described types that behave as usual existentials $\exists^\tau (\alpha : \kappa). \sigma$ but remembering the witness type $\tau$ for the abstract type $\alpha$.

We create a transparent existential type with the expression pack $e$ as $\exists^\tau (\alpha : \kappa). \sigma$, which behaves much as pack $\langle \tau, e \rangle$ as $\exists^\tau (\alpha : \kappa). \sigma$, except that the witness type $\tau$ 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 $\alpha$ cannot be turned back into its witness type $\tau$ 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 comes 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 $\exists^\tau (\alpha : \kappa). \sigma$ into one of type $\exists^\tau (\alpha : \kappa). \sigma$.

Transparent existential types may also be used abstractly, with the expression repack $^\tau \langle \alpha, x \rangle = e_1$ in $e_2$, which is the analog of the expression repack $^\tau \langle \alpha, x \rangle = e_1$ in $e_2$ but when $e_1$ is a transparent existential type $\exists^\tau (\alpha : \kappa). \sigma_1$. In both cases, $e_2$ is typed in a context extended with the abstract types $\overline{\tau}$ and a variable $x$ of type $\sigma_1$. Crucially, $e_2$ cannot see the witnesses $\overline{\tau}$. However, the abstract type variables $\overline{\tau}$ may still appear in the type $\sigma_2$ of the expression $e_2$, and therefore it is made transparent again in the result type of repack $^\tau \langle \alpha, x \rangle = e_1$ in $e_2$, which is $\exists^\tau (\alpha : \kappa). \sigma_2$. We do not need a primitive transparent version unpack $^\tau \langle \alpha, x \rangle = e_1$ in $e_2$, since it can be defined as unpack $\langle \alpha, x \rangle = \text{seal} e_1$ 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 lift $^\tau e$, turns an expression of type $\sigma_1 \to \exists^\tau (\alpha : \kappa). \sigma_2$ into one of type $\exists^\tau (\alpha : \kappa). (\sigma_1 \to \sigma_2)$ as long as $\alpha$ is fresh for $\sigma_1$. Since we can observe the witness $\tau$, we can ensure that the choice of the witness does not depend on the value (of type $\sigma_1$), 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 $\langle \text{int}, 42 \rangle$ as $\exists^\tau \alpha. \alpha$ else pack $\langle \text{float}, 0.5 \rangle$ as $\exists^\tau \alpha. \alpha$ It has type $\text{bool} \to \exists^\tau \alpha. \alpha$, but it would be unsound to consider it at the type $\exists^\tau \alpha. \text{bool} \to \alpha$.

Similarly, lifting across a universal type variable $\beta$ of kind $\kappa'$, written lift $^\tau e$, turns an expression of type $\Lambda(\beta : \kappa'). \exists^\tau (\alpha : \kappa). \sigma$ into one of type $\exists^\tau (\alpha : \kappa). \forall(\beta : \kappa'). \sigma[\alpha \mapsto \alpha' : \beta']$, provided $\beta$ is fresh for $\tau$, using skolemization of both the existential variable $\alpha$ and its witness type $\tau$.

To summarize, we have extended the syntax of $\mathcal{F}^\omega$ as follows:

\[
\begin{align*}
\tau & ::= \ldots \mid \exists^\tau (\alpha : \kappa). \sigma \\
e & ::= \ldots \mid \text{pack } e \text{ as } \exists^\tau (\alpha : \kappa). \sigma \mid \text{seal } e \mid \text{repack }^\tau \langle \alpha, x \rangle = e_1 \text{ in } e_2 \mid \text{lift }^\tau e \mid \text{lift }^\tau e
\end{align*}
\]

Their typing rules are given in §5 of supplementary materials. These constructs have no additional computational content, namely $\text{repack }^\tau \langle \alpha, x \rangle = \tau$ in $\sigma$ 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 $\text{seal}^n$ for $n$ applications of $\text{seal}$.
We can define a lifting operation \( \text{lift}^\omega \langle \alpha, x_1 = e_1 \, @ \, e_2 \rangle \) 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 \( \text{lift}^\omega e \) that uses a combination of the primitive \( \text{lift}^\omega \) and \( \text{lift}^\omega \) to turn an expression \( e \) of type \( \forall \alpha. \sigma_1 \rightarrow \exists^\omega_\alpha (\beta). \sigma_2 \) into one of type \( \exists^\omega_\alpha (\beta'). \forall \alpha. \sigma_1 \rightarrow \sigma_2 \left[ \beta \mapsto \beta' \, \, \alpha \right] \), which is the key transformation for lifting existentials out of applicative functor bodies. Its implementation is given in §5.1 of supplementary materials.

### 4.5 Implementation of transparent existential types in plain \( F^\omega \)

Interestingly, transparent existential types are completely definable in plain \( F^\omega \). A concrete implementation is given in §5.2 of supplementary materials. The implementation \( e_0 \) is not itself of much interest: most expressions are \( \eta \)-expansions of the identity. However, using regular \( F^\omega \) existentials, \( e_0 \) can be abstracted into \( e_\Xi = \text{pack} \langle \tau_0, e_0 \rangle \) as \( \tau_\Xi \) where \( \tau_0 \) is the interface type that hides the implementation of the type \( \Xi \). Using this definition, we may see a program \( e \) using transparent existential types as a program unpack \( \langle \Xi, x_\Xi \rangle = e_\Xi \) in \( e \) in plain \( F^\omega \), with the following additional syntactic sugar\(^{16}\):

\[
\exists^\omega\tau (\beta : \kappa). \sigma = \Xi : \tau (\lambda (\beta : \kappa). \sigma) \quad \text{seal } e = x_\Xi.\text{Seal} \_ \_ e \\
\text{pack } e \text{ as } \exists^\omega\tau (\alpha). \sigma \equiv \Xi : \exists \tau. \text{Pack} (\lambda (\alpha : \_). \sigma) e \quad \text{lift}^\omega e = x_\Xi.\text{Lift}^\omega \_ \_ e \\
\text{repack}^\omega (\alpha, x) = e_1 \text{ in } e_2 \equiv \Xi : \exists \tau. \text{Repack} \_ \_ e_1 (\lambda (\alpha : \_). \lambda (x : \alpha). e_2) \quad \text{lift}^\omega e = x_\Xi.\text{Lift}^\omega \_ \_ e
\]

### 4.6 Elaboration judgments

As for \( M^\omega \), the elaboration relies on a subtyping judgment and a typing judgment for both signatures and modules. However, as \( M^\omega \) signatures are already \( F^\omega \) types, we can reuse the \( M^\omega \) typing judgment (although we should now reread it with implicit kinds). Specifically, neither \( M^\omega \) 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 \( \Gamma \vdash C < C' \rightsquigarrow f \) extends \( M^\omega \) subtyping to return an explicit coercion function \( f \). The judgment is also defined for declarations \( \Gamma \vdash D < D' \rightsquigarrow f \). Interestingly, as signatures do not contain transparent existential types, subtyping between signatures is (a subcase of) standard subtyping in \( F^\omega \). As they are similar to \( M^\omega \) subtyping, we left the rules in §6.1 of supplementary materials. The judgments have the following property regarding \( F^\omega \) typing:

\[
\Gamma \vdash C < C' \rightsquigarrow f \implies \Gamma \vdash f : C \rightarrow C' \\
\Gamma \vdash D < D' \rightsquigarrow f \implies \Gamma \vdash f : \{ D \} \rightarrow \{ D' \}
\]

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

The judgment \( \Gamma \vdash \omega M : \exists^\omega \alpha. C \rightsquigarrow e \) extends \( M^\omega \) typing with the elaborated module term \( e \), when the mode \( \hat{\theta} \) should coincide with \( \hat{\theta} \) when present and is actually left implicit as for the \( M^\omega \)

---

\(^{16}\)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.
system. Hence, we usually just write $\Gamma \vdash M : \exists^\emptyset \bar{a}.\gamma \Rightarrow e$. The judgment is also defined for bindings $\Gamma \vdash \bar{A} : \exists^\emptyset \bar{a}.\Delta \Rightarrow e$ (with the mode left implicit). The properties of the two judgments

**Theorem 4.1 (Soundness).** When typing a module, the elaborated module term is well typed regarding $P\alpha$ typing, and the source module term is well typed regarding $M\omega$ typing.

\[
\begin{align*}
\Gamma \vdash M : \exists^\emptyset \bar{a}.\gamma &\Rightarrow e \implies \Gamma \vdash e : \exists^\emptyset \bar{a}.\gamma \\
\Gamma \vdash \bar{B} : \exists^\emptyset \bar{a}.\bar{D} &\Rightarrow e \implies \Gamma \vdash e : \exists^\emptyset \bar{a}.\Delta \end{align*}
\]

(1)

**Theorem 4.2 (Completeness).** Well-typed $M\omega$ terms or bindings can always be elaborated:

\[
\begin{align*}
\Gamma \vdash M : \exists^\emptyset \bar{a}.\gamma &\Rightarrow \exists e, \delta, \Gamma \vdash M : \exists^\emptyset \bar{a}.\gamma \Rightarrow e \land \text{mode}(\delta) = \diamond \\
\Gamma \vdash \bar{B} : \exists^\emptyset \bar{a}.\bar{D} &\Rightarrow \exists e, \delta, \Gamma \vdash \bar{B} : \exists^\emptyset \bar{a}.\Delta \Rightarrow e \land \text{mode}(\delta) = \diamond
\end{align*}
\]

(2)

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

### 4.7 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:

\[
\begin{align*}
\text{E-Typ-Struct} &\quad \Gamma \vdash A : \exists^\emptyset \bar{a}_1.\gamma \Rightarrow e_1 \\
&\quad \Gamma, \bar{a}_1, A_1 : \Delta \vdash \bar{B} : \exists^\emptyset \bar{a}_2.\Delta \Rightarrow e_2 \\
&\quad \text{lift}^\gamma (\bar{a}_1, x_1 = e_1 \@ (\text{let } A_1 = x_1.f_1 \text{ in } e_2) ) \\
\end{align*}
\]

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:

\[
\begin{align*}
\text{E-Typ-SeqApp} &\quad \Gamma \vdash A : \exists^\emptyset \bar{a}_1.\gamma \Rightarrow e_1 \\
&\quad \Gamma, \bar{a}_1, A_1 : \Delta \vdash \bar{B} : \exists^\emptyset \bar{a}_2.\Delta \Rightarrow e_2 \\
&\quad \text{lift}^\gamma (\bar{a}_1, x_1 = e_1 \@ (\text{let } A_1 = x_1.f_1 \text{ in } e_2) ) \\
\end{align*}
\]

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

\[
\begin{align*}
\Gamma \vdash A : \exists^\emptyset \bar{a}.\Delta \Rightarrow e \\
&\quad \text{A} \notin \Gamma \\
\Gamma \vdash \text{structure}_A \bar{B} \text{ end} : \exists^\emptyset \bar{a}.\bar{\gamma} \Rightarrow e \end{align*}
\]

**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 $(P : S)$ may introduce new abstract types $\bar{a}$ as prescribed by the (elaboration $\lambda \bar{a}.\gamma$ of the) signature $S$, but these are transparent existentials in the type of $(P : S)$. 

\[
\begin{align*}
\text{E-Typ-Mod-Seal} &\quad \Gamma \vdash M : \exists^\emptyset \bar{a}.\gamma \Rightarrow e \\
\Gamma \vdash \lambda \bar{a}.\gamma \Rightarrow e \\
\Gamma \vdash P : C' \Rightarrow e \\
\Gamma \vdash C' : C[\bar{a} \mapsto \bar{\tau}] \Rightarrow f \\
\end{align*}
\]

\[
\begin{align*}
\text{E-Typ-Sig-App} &\quad \Gamma \vdash S : \lambda \bar{a}.\gamma \Rightarrow e \\
&\quad \Gamma \vdash P : C' \Rightarrow e \\
&\quad \Gamma \vdash C' : C[\bar{a} \mapsto \bar{\tau}] \Rightarrow f \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash (P : S) : \exists^\emptyset \bar{a}.\gamma \Rightarrow \text{pack } f \text{ as } \exists^\emptyset \bar{a}.\gamma \\
\end{align*}
\]

, Vol. 1, No. 1, Article . Publication date: October 2023.
Elaboration of functors. At first glance, the elaboration of functors seems to differ more significantly in the applicative and generative cases:

\[
\begin{align*}
\text{E-Typ-Mod-AppFct} & \quad \Gamma \vdash \pi: \lambda \pi.\mathcal{C}_a \\
& \quad \Gamma, \pi, \alpha: \mathcal{C}_a \vdash \mathcal{M} : \exists^\forall (\beta).\mathcal{C} \rightsquigarrow e \\
& \quad \Gamma \vdash (Y : S) \rightarrow \mathcal{M} : \exists^\forall (\beta).\mathcal{C} \\
& \quad \forall \pi.\mathcal{C}_a \rightarrow \mathcal{C}[\beta \mapsto \beta'(\alpha)] \\
& \quad \rightsquigarrow \text{lif}^\prime(\lambda\pi.\mathcal{C})(Y : \mathcal{C}_a).e
\end{align*}
\]

The body of an applicative functor is elaborated to transparent existentials which are lifted through \(\lambda\)’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 \(\text{lif}^\ast\) is neutral on terms that do not have transparent existential types, the elaboration of the generative case could also be written \(\text{lif}^\ast(\lambda(\_ : ()).e, so that the two cases only differ by the modes of elaboration of their bodies.

5 RELATED WORKS

The literature regarding ML modules is rich and varied. The link between abstract types in ML module systems and existential types in \(F^{\omega}\) was initially explored by Mitchell and Plotkin [12]. This vision was opposed by MacQueen [10] 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. [6] 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. [6]. Two approaches for the formalization and improvement of abstract types in SML were later independently yet simultaneously described by Leroy [7] using manifest types and Harper and Lillibridge [5] via an adapted \(F^{\omega}\) with translucent sums. The genesis of the OCAML module system was specified by Leroy [7, 9] with, later, an extension to applicative functors [8].

The key idea for a simplified link between modules and \(F^{\omega}\), developed by Russo [19], 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 [14] 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. [17] with the elaboration of a large subset of the SML module system into \(F^{\omega}\), dubbed the \(F\)-ing approach. \(F\)-ing gives a syntactic translation from SML syntax directly into \(F^{\omega}\), thus providing a semantic by elaboration. \(F\)-ing is safe by construction [17], inheriting the property from \(F^{\omega}\), 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, \(M^{\omega}\) gives a specification directly on source terms, without having to think in terms of encoding, but leveraging the insights provided by the elaboration to \(F^{\omega}\). 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 [16] achieved a unification of the core and module languages (thus, unstratified), called 1ML, using \(F^{\omega}\) 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

\[17\text{Besides, their work has also been mechanized in Coq for the generative case. A Coq formalization of our approach, including the applicative case, would be welcomed. It is left for future work.}\]
We have introduced and formalized $M^\omega$, a middle point between the source path-based module system used in OCAML and $F^\omega$. Using $M^\omega$, we first shone a new light on the mechanisms of the OCAML type system, and provided a detailed description of the solvable and unsolvable cases of signature avoidance. Second, we gave an improved elaboration of modules into $F^\omega$, 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 $M^\omega$ system to cope with abstract signatures, which are a particularity of OCAML. While the $F^\omega$-style polymorphism could perhaps be sufficient to solved the very few useful cases, a general approach is likely to be more challenging.

An immediate application of our work is to use $M^\omega$-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, $M^\omega$ 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 $M^\omega$ signatures would make subtyping undecidable. Finding a set of good sense restrictions to maintain decidability, as well as mixing the path-based and $M^\omega$ 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 $F^\omega$. This considerably simplifies the treatment of applicative functors, which should ease the mechanized proof of Theorem 4.1, 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 $F^\omega$, we may also explore another path, extending $F^\omega$ with minimalist constructs, typically for dealing with primitive lifting of existentials, so that we may program with modules in this light extension of $F^\omega$. Besides programming directly in $F^\omega$, this should also help structure and conduct modular program proofs.

\section{Discussion and Future Works}

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.

\footnote{More details on omitted features and abstract signature can also be found in supplementary materials.}
REFERENCES


Fulfilling OCaml modules with transparency

Supplementary material

ANONYMOUS AUTHOR(S)

This document contains the supplementary material for the submission Fulfilling OCaml modules with transparency. We discuss the features left out of the scope of our work in §1. We detail more differences with F-ing in §2. We give the full set of typing rules for the $M^{\omega}$ system in §4. The implementation of transparent existentials, and the typing rules of $F^{\omega}$ are given in §5. The elaboration rules are given in §6.

ACM Reference Format:
Anonymous Author(s). 2023. Fulfilling OCaml modules with transparency: Supplementary material. 1, 1 (October 2023), 15 pages.

Contents

Abstract 1
Contents 1
1 Omitted features 2
2 Detailed comparison with F-ing modules 3
3 Examples 4
4 $M^{\omega}$ System 6
5 System $F^{\omega}$ 11
6 Elaboration rules 14
1 OMITTED FEATURES

We see no difficulty in adding the following features:

- first-class modules
- explicit constraints \( S \) with type \( t = u \) and deleting constraints \( S \) with type \( t : = u \)
- include and open operators.

We did not include the signature expression module type of \( M \) either. The most common restricted form module type of \( P \) could be added provided we restrict to explicit signatures to preserve decidability of subtyping. Still, \texttt{module type of} \( P \) in the \( M^\omega \) system will always return a concrete signature \( C \), while OCAML has a strange inconsistent behavior, as it returns an abstract signature when \( P \) is the original path to a module, but a concrete (strengthened) signature when it is an alias. In the former case, OCAML users often use \texttt{module type of} \( \text{sig}\_A \text{include} \_P \text{end} \) to always obtain the concrete (strengthened) version of the signature.

The use of module type of \( M \), allowed in OCAML but mainly used in the expression to obtain the concrete signature of a path, may not be very useful, and ad hoc in the \( M^\omega \) system. Indeed, the type of a module \( M \) is actually an abstract signature of the form \( \exists \alpha.\bar{\alpha}.C \) with some transparent or opaque existential types, while a \( M^\omega \) signature is \( \lambda\alpha.C \) with \( \lambda \) binders instead of existential binders. Turning existential into \( \lambda \) binders is syntactic possible but logically strange, especially since the opacity information would be lost along the way.

We did not cover recursive modules. We did not either take into consideration more subtle cases of interaction between the typing system of the core and module languages (unresolved inference variables, GADTS, etc.), which will also be quite important when adding modular implicits. We believe these are orthogonal issues.

Abstract signatures. Still, abstract signatures, which are a particularity of OCAML, are a feature that does not fit well in our framework. Abstract signatures allow to reveal the existence of a submodule whose signature is kept fully abstract; they can also be used in contravariant positions to write functors that are polymorphic in the signature of their argument. Even though there are very few real uses of abstract signatures in practice, they do not fit well in the \( M^\omega \) system, which relies on the placement of universal or existential quantifiers for abstract types driven by their polarity in the structure of signatures. Therefore, instantiating an abstract signature by a concrete signature in the source would amount to introducing new quantifiers, deeper inside the structure but according to polarities, which is not doable with \( F^\alpha \) type instantiation alone.

Perhaps, the \( F^\omega \)-style polymorphism would be sufficient to solved the very few useful cases, without covering the full OCAML-style abstract signatures mechanism.
2 DETAILED COMPARISON WITH F-ING MODULES

While based on the work of Rossberg et al. [17], our approach differs from it in some design choices and technical aspects. Below, we only list the difference between features that are related. We do not repeat differences mentioned earlier.

Technical, minor choices

- We follow the OCAML grammar and syntactically distinguished applicative and generative functors using a special unit argument. Instead, in F-ing, signatures of functors are annotated by either a $P$ to indicate pure, applicative functors or an resp. $I$ for impure, generative functors.
- We use separate identifier categories for types, values, modules, modules types and functor parameters. F-ing only has one identifier category (namely, $X$) and uses one-field records with reserved keywords to distinguish between declarations.
- We distinguish between declarations and signatures, removing the need for the notion of atomic signatures.

Key differences

- As it uses normal existential types (which cannot be skolemized), the F-ing approach for applicative functors is to abstracts over the whole typing environment. This a-priori skolemization makes the typing rules more complex (especially the rule B-seq) and the actual elaborated terms cluttered. In addition, it does not convey the intuition of locality: we believe that a-posteriori skolemization better corresponds to the underlying mechanism of applicative functors. Besides, it make applicative and generative functors closer to each other.
- Our identity tags are related to the notion of semantic paths in F-ing. The idea of using abstract types to track identities was already understood in F-ing. We introduce identity tags as a separate source-to-source transformation, showing that the underlying mechanism can be completely emulated with type abstraction, without tweaking the typing rules (as done in F-ing).
3 EXAMPLES

3.1 Signature avoidance

All examples refer to a signature $S$ defined as:

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

3.1.1 In valid OCaml syntax.

Example 3.1. A basic case of signature avoidance similar to Figure 4 in valid OCaml syntax. The anonymous projection is replaced by an anonymous functor call. The key is that the type field of the argument $Y.t$ is inaccessible outside of the functor’s body and must therefore be avoided.

```ocaml
1 module M =
2   (functor (X: S) -> struct
3      type a = X.t * bool
4      type b = X.t * int
5    end)
6   (struct type t = int end : S)
7 let f ((x,_) : M.a) = ((x,42) : M.b)
```

As the typechecker abstracts both $a$ and $b$, it fails at line 9 with

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

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

Example 3.2. A modified version of Example 3.1 with a solvable case of signature avoidance. Here, abstracting $a$ and rewriting the type declaration for $b$ as type $b = a \times \text{int}$ would keep all type-sharing.

Source code

```ocaml
1 module M =
2   (functor (X: S) -> struct
3      module X1 = Y
4      module X2 = Y
5    end)
6   (struct type t = int end : S)
```

Inferred signature

```ocaml
module M : sig
  type a = A.a
  type b = A.a \times \text{int} end
```

Anchored signature

```ocaml
module M : sig
  module X1 : S
  module X2 : (= A.X1 < S) end
```

Example 3.3. A variant of the previous example at the module level, with module aliases instead of type equalities.

Source code

```ocaml
1 module M =
2   (functor (X: S) -> struct
3      module X1 = Y
4      module X2 = Y
5    end)
6   (struct type t = int end : S)
```

Inferred signature

```ocaml
module M : sig
  module X1 : S
  module X2 : S end
```

Anchored signature

```ocaml
module M : sig
  module X1 : S
  module X2 : (= A.X1 < S) end
```
3.1.3 Module encoding in $F^\omega$.

**Example 3.4.** A simple module $M$ with three type fields, on the left-hand side. The raw encoding on the right-hand side shows how abstract types are shared between components via lifting.

**Source code**

```ocaml
module M = struct
  type t = A.t
  type u = A.u
  type v = A.t × A.u
end
```

**Encoding of $e$**

\[
e = \text{lift}^\gamma \langle \alpha, x_1 = \text{pack} ((), \{ \ell_t : \langle \alpha \rangle \}) \text{ as } \exists^\gamma \alpha. \{ \ell_t : \langle \alpha \rangle \} \\
@ \text{lift}^\gamma \langle \beta, x_2 = \text{pack} ((), \{ \ell_u : \langle \beta \rangle \}) \text{ as } \exists^\gamma \beta. \{ \ell_u : \langle \beta \rangle \} \\
@ \{ \ell_v : \langle \alpha \times \beta \rangle \} \rangle
\]

**Signature of $e$**

\[
S = \exists \alpha, \beta. \{ \ell_t : \langle \alpha \rangle ; \ell_u : \langle \beta \rangle ; \ell_v : \langle \alpha \times \beta \rangle \}
\]
4 M\(^{\omega}\) SYSTEM

In this section we gather the typing rules of the M\(^{\omega}\) system.

4.1 Subtyping

4.1.1 Signature subtyping

\[
\frac{\text{M-Sub-Sig-Struct}}{\frac{\Gamma \vdash D : \lambda \alpha_1. D}{\frac{\Gamma, \tau \vdash C < C' \ [\pi \mapsto \tau]}{\frac{\Gamma \vdash \text{sig} D \text{ end} < \text{sig} D' \text{ end}}{\frac{\Gamma \vdash \text{M-Sub-Sig-AppFct}}{\frac{\Gamma, \tau' + \lambda \alpha.C_a \ [\pi \mapsto \tau] < C'}{\frac{\Gamma \vdash (\forall \alpha.C_a \rightarrow C' \forall \alpha'.C_a' \rightarrow C''}}}}}}
\]

4.1.2 Declaration subtyping

\[
\frac{\text{M-Sub-Decl-Val}}{\frac{\Gamma \vdash \text{val} x : \tau < (\text{val} x : \tau)}{\frac{\text{M-Sub-Decl-Type}}{\frac{\Gamma \vdash \text{type} t = \tau < (\text{type} t = \tau)}{\frac{\text{M-Sub-Decl-ModType}}{\frac{\Gamma \vdash \text{module type} \ C < \text{module type} \ C'}{\frac{\Gamma \vdash \text{module type} \ T = \lambda \alpha.C < (\text{module type} \ T = \lambda \alpha.C')}{\text{M-Sub-Decl-ModType}}}}}}}}}
\]

4.2 Typing

4.2.1 Signature typing

\[
\frac{\text{M-Typ-Sig-ModType}}{\frac{\Gamma \vdash P : \text{sig} D \text{ end}}{\frac{\Gamma \vdash P.T : \lambda \alpha.C}{\frac{\Gamma \vdash P : \text{module type} \ T = \lambda \alpha.C \in D}{\text{M-Typ-Sig-LocalModType}}}}}
\]

\[
\frac{\text{M-Typ-Sig-GenFct}}{\frac{\Gamma \vdash S : \lambda \alpha.C}{\frac{\Gamma \vdash () \rightarrow S() \rightarrow \exists \alpha'.C'}{\text{M-Typ-Sig-AppFct}}}}}
\]

\[
\frac{\text{M-Typ-Sig-Str}}{\frac{\Gamma \vdash \text{sig} \ D \text{ end} : \lambda \alpha. \text{sig} D \text{ end}}{\text{M-Typ-Sig-Con}}}}}
\]

4.2.2 Declaration typing

\[
\frac{\text{M-Typ-Decl-Val}}{\frac{\Gamma \vdash u : \tau}{\frac{\Gamma \vdash A (\text{val} x : u) : (\text{val} x : \tau)}{\text{M-Typ-Decl-Type}}}}}
\]

\[
\frac{\text{M-Typ-Decl-TypeAbs}}{\frac{\Gamma \vdash (\text{type} t = A.t : \lambda \alpha. \text{type} t = \alpha)}{\text{M-Typ-Decl-ModType}}}}}
\]

\[
\frac{\text{M-Typ-Decl-Mod}}{\frac{\Gamma \vdash S : \lambda \alpha.C}{\frac{\Gamma \vdash (\text{module} X : S) : \lambda \alpha. (\text{module} X : C')}{\text{M-Typ-Decl-Seq}}}}}
\]

\[
\frac{\text{M-Typ-Decl-Empty}}{\frac{\Gamma \vdash \emptyset : \emptyset}{\text{M-Typ-Decl-Seq}}}}}
\]

\[
\frac{\text{M-Typ-Decl-Seq}}{\frac{\Gamma \vdash \emptyset : \lambda \alpha_1. D}{\frac{\Gamma, \pi_1, A.I : \emptyset \vdash \emptyset}{\text{M-Typ-Decl-Seq}}}}}}}
\]

\text{Anon.}, Vol. 1, No. 1, Article . Publication date: October 2023.
4.2.3 Core type checking extension

\[
\Gamma \vdash P : \text{sig } \overline{D} \text{ end} \quad \text{type } t = \tau \in \overline{D} \quad \Gamma \vdash P.t : \tau
\]

4.2.4 Module typing

\[
\begin{array}{l}
\text{M-Typ-Var} \quad (Y : C) \in \Gamma \\
\text{M-Typ-Local} \quad (A.X : \text{module } C) \in \Gamma \\
\text{M-Typ-Struct} \quad \Gamma \vdash B : \exists^\alpha \pi.D \quad A \notin \Gamma
\end{array}
\]

4.2.5 Binding typing

\[
\begin{array}{l}
\text{M-Typ-Let} \quad \Gamma \vdash e : \tau \\
\text{M-Typ-Type} \quad \Gamma \vdash u : \tau \\
\text{M-Typ-AbsType} \quad \Gamma \vdash A \vdash (\text{type } t = \text{type } \alpha) \\
\text{M-Typ-Mod} \quad \Gamma \vdash M : \exists^\alpha \pi.C
\end{array}
\]

4.2.6 Core expression typing extension

\[
\Gamma \vdash [M_1] : \text{sig module } Val : C_1 \quad \text{type } id = \tau \text{ end} \\
\Gamma \vdash [M_2] : \text{sig module } Val : C_2 \quad \text{type } id = \tau \text{ end} \\
\end{array}
\]

\[
\exists C_0 \left\{ \begin{array}{l}
\Gamma \vdash C_0 < C_1 \\
\Gamma \vdash C_0 < C_2
\end{array} \right. \\
\text{For this proof, we consider a slightly modified } M^{\omega} \text{ system based on } F^{\omega}_{<} \text{ rather than } F^{\omega}. \text{This enriched system is built in two steps. First, we extend the quantifiers } \exists^\alpha, \forall, \text{ and } \lambda \text{ to support bounded}
\]

\[
\begin{array}{l}
\Gamma \vdash P : \text{sig } \overline{D} \text{ end} \quad \text{val } x : \tau \in \overline{D} \\
\Gamma \vdash \text{A.x : val } \tau \in \Gamma
\end{array}
\]
quantification for abstract types that serve as identities (the other types being bound by the top bound \( \top \)). We modify the typing and subtyping rules accordingly, the only crucial change being some additional subtyping premises when instantiating bounded variables (for functor application and sealing). Then, we modify the typing rules for introducing abstract tags (\textit{M-Typ-Decl-TypeAbs} and \textit{M-Typ-Bind-AbsType}) by distinguishing between identity tags (Rules \textit{M-Typ-Decl-TypeAbsId} and \textit{M-Typ-Bind-AbsTypeId}) and other abstract types. While abstract types are simply bounded by \( \top \), identities are bounded by the signature of the associated value. The four affected rules (for binding and declarations) are as follows:

\[
\begin{align*}
\text{M-Typ-Decl-TypeAbs} & \quad t \neq \text{id} \\
\Gamma \vdash \text{A.} \, \text{type} \, t = \text{A.} \, \text{id} : \lambda (\alpha < \top). \, (\text{type } t = \alpha) \\
\text{M-Typ-Bind-AbsType} & \quad t \neq \text{id} \\
\Gamma \vdash \text{A.} \, \text{type} \, t = \text{A.} \, \text{id} : \exists \, (\alpha < \top). \, (\text{type } t = \alpha)
\end{align*}
\]

\[
\begin{align*}
\text{M-Typ-Decl-TypeAbsId} & \quad \Gamma \vdash \text{id} \vdash \text{id} : \text{C} \\
\Gamma \vdash \text{A.} \, \text{Val} : \text{C} \\
\text{M-Typ-Bind-AbsTypeId} & \quad \Gamma \vdash \text{id} \vdash \text{id} \vdash \text{id} : \text{C} \\
\Gamma \vdash \text{A.} \, \text{Val} : \text{C}
\end{align*}
\]

The proof proceeds in two successive steps:

1. We show that the enriched system maintains a notion of identity wellformedness: values are always supertypes of the bound of their associated identity.

More formally, we consider a judgment \( \Gamma \vdash C : \text{id}_w f \) that goes through signatures, with a special rule for (possibly high-order) identities:

\[
\Gamma \vdash \tau < C \\
\Gamma \vdash \text{sig module Val : C type id} = \tau \text{ end : id}_w f
\]

We show that the enriched typing judgments (for modules and signatures) only produce signatures with wellformed identities.

2. Using the previous result, we prove that typability in the \( M^\omega \) system implies typability in the enriched typing.

The first step goes by a simple induction over the typing derivation: identities are either introduced fresh, in which case the bound is equal to the signature of the value, or obtained via subtyping, in which case we use transitivity of subtyping.

The rest of this section is dedicated to the proof of the second step. The only difference between the two systems comes from additional subtyping conditions between the bounds. The core of the proof is to show that these are actually not restrictive. This relies on two key facts: (1) subtyping is only done with a right-hand side signature that comes from a source signature, i.e., that is the result of typing a source signature, and (2) such signatures always contain the bounds of their identity types in one positive occurrence.

\textbf{Properties of typed source signatures.} \( M^\omega \) signatures obtained via typing of a source signature (referred to as \text{TSS} in the following) are more restricted than general \( M^\omega \) signatures, specifically regarding the role of bounds of identity types. Bounds allow us to store the signature of the original module that introduced an identity, and every module with the same identity will have a supertype of the origin. In a \text{TSS}, the bound of an identity type \( \alpha \) is always exactly the signature of the first module with that identity, and therefore, the bound appears explicitly in the signature. In contrast, in a signature obtained via module typing, the original module (that had the bound as its signature) can be lost, typically when using projection, leaving only modules with supertypes signatures.
First-order example. Let us consider a simple example, where typing a source signature \( S \) produces a \( M^\omega \) one \( \Gamma \vdash S : \lambda (\alpha < C_\alpha).C \). Inside \( C \) there is a subterm (in positive position) equal to:

\[
\text{sig module } \text{Val} : C_\alpha \text{ type } id = \alpha \text{ end}
\]

When subtyping this signature with another one, say \( \lambda (\beta < C_\beta).C' \), in the enriched system, there is an additional subtyping check between the bound (by comparison with the \( M^\omega \) system):

\[
\Gamma \vdash C_\beta < C_\alpha
\]

If the subtyping between \( C' \) and \( C \) succeeds, then \( C' \) features a subterm of the form

\[
\text{sig module } \text{Val} : C_0 \text{ type } id = \beta \text{ end}
\]

By subtyping, we have \( \Gamma \vdash C_0 < C_\alpha \). Thanks to the invariant of the first part of the proof, we know that \( \Gamma \vdash C_\beta < C_0 \). By transitivity of subtyping, this ensures (1).

Generalization. As said above, the only difference between the enriched and normal systems comes from additional subtyping conditions when instantiating type variables. That is, in the rules for ascription (M-Typ-Mod-Ascr), functor application (M-Typ-Mod-AppApp), concrete signatures (M-Typ-Sig-Con) and subtyping between functors (M-Sub-Sig-AppFct and M-Sub-Sig-GenFct), before instantiating the abstract types \( \overline{\sigma} \) with concrete ones \( \overline{\tau} \), an additional premise is added to ensure that the \( \overline{\tau} \) are subtypes of the bound of the \( \overline{\sigma} \). These rules are of the following form (with either lambda, existential, or universal quantification), shown here with only one type variable for readability:

\[
\ldots [\lambda (\alpha < C_\alpha).C'] \quad \Gamma \vdash C < C'[\alpha \mapsto \tau] \quad (2) \quad \Gamma \vdash \tau < C_\alpha \quad (3) \\
\ldots
\]

The goal of this part of the proof is to show that the additional condition (3) is actually implied by the subtyping on the signatures (2).

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

\[
\begin{align*}
[sig \overline{D} \text{ end}]^+ &= [D]^+ & \text{(Structural signature)} \\
[\forall \alpha.C_\alpha \rightarrow C]^+ &= \forall \alpha.[C]^+ & \text{(Applicative functor)} \\
[() \rightarrow \_]^+ &= \emptyset & \text{(Generative functor)} \\
[\text{module } X : C]^+ &= (\text{module } X : C), [C]^+ & \text{(Submodule declaration)} \\
[D]^+ &= D & \text{(Other declarations)}
\end{align*}
\]

Declarations inside applicative functors are universally quantified. A key observation is that a signature in \( \text{tss} \) form always contain the bound of its identity type among its positive declarations:

\[
\Gamma \vdash S : \lambda (\alpha < C_\alpha).C \implies (\text{module } \text{Val} : C_\alpha) \in [C]^+
\]

\[
\Gamma \vdash S : \lambda (\alpha < \lambda \overline{\beta}.C_\alpha).C \implies \forall \overline{\beta}.(\text{module } \text{Val} : C_\alpha) \in [C]^+
\]

Crucially, all the instantiations we consider feature a \( \text{tss} \) on their right-hand side, i.e., \( C' \) is a \( \text{tss} \). By construction, subtyping between two signatures implies subtyping between declarations of their positive parts. Ignoring applicative functors at first, this gives:

\[
\Gamma \vdash C < C' \implies \forall \overline{D}' \in [C']^+. \exists \overline{D} \in [C]^+. \Gamma \vdash \overline{D} < \overline{D}'
\]
Therefore, there exists a declaration in the positive part of $C$ that is a subtype of the explicit bound of $\alpha$, which appears in $[C']^+$:

$$\exists C_0. \; \Gamma \vdash \text{module } Val : C_0 < \text{module } Val : C_\alpha$$

This implies $\Gamma \vdash C_0 < C_\alpha$. Using the invariant of the first step of the proof, we get $\Gamma \vdash \tau < C_0$, which concludes the proof of transitivity of subtyping.

Higher order declarations with universal quantification originating from applicative functors are treated similarly, only adding universally quantified variables in the typing context $\Gamma$. 
5 SYSTEM $F^\omega$

5.1 lift* operator

The operator lift* is defined as lift*$_p q e$, where $p$ and $q$ represent the size of $\alpha$ and $\beta$, where lift*$_q e$ is itself inductively defined as follows:

\[
\begin{align*}
\text{lift*}_p q e & \triangleq \text{repack}_q (\alpha, x) = \text{lift}_p (\text{lift*}_p e) \text{ in lift*}_q x \\
\text{lift*}_0 q e & \triangleq e
\end{align*}
\]

\[
\begin{align*}
\text{lift} (\Lambda \alpha. \text{lift}_p (e \alpha))
\end{align*}
\]

5.2 Implementation of transparent existentials

Plain $F^\omega$ implementation of transparent existentials:

\[
\begin{align*}
\tau_\Xi & \triangleq \\
\exists^\psi (\Xi : \forall x. \psi \rightarrow (\psi \rightarrow \psi)). \\
\text{Pack} & = \forall x. \forall (\alpha : \psi). \forall (\beta : \psi). \psi \alpha \rightarrow \Xi \times \alpha \psi \\
\text{Seal} & = \forall x. \forall (\alpha : \psi). \forall (\beta : \psi). \psi \alpha \rightarrow \Xi \times \alpha \psi \\
\text{Repack} & = \forall \psi : \Xi. \forall (\alpha : \psi). \psi \alpha \rightarrow \\
& \quad (\forall (\alpha : \psi). \Xi \times \alpha \beta \rightarrow \Xi \times \alpha \beta) \\
\text{Lift} & = \forall \alpha. \forall (\alpha : \psi). \forall (\beta : \psi) \rightarrow \Xi \times \alpha \beta \rightarrow \Xi \times \alpha \beta \\
\end{align*}
\]

\[
\begin{align*}
e_0 & \triangleq \\
\text{Pack} & = \lambda \alpha. \lambda (\alpha : \psi). \forall (\beta : \psi) \rightarrow \Xi \times \alpha \beta \rightarrow \Xi \times \alpha \beta \\
\text{Seal} & = \lambda \alpha. \lambda (\alpha : \psi). \forall (\beta : \psi) \rightarrow \Xi \times \alpha \beta \rightarrow \Xi \times \alpha \beta \\
\text{Repack} & = \lambda \alpha. \lambda (\alpha : \psi). \forall (\beta : \psi) \rightarrow \Xi \times \alpha \beta \rightarrow \Xi \times \alpha \beta \\
\text{Lift} & = \lambda \alpha. \lambda (\alpha : \psi). \forall (\beta : \psi) \rightarrow \Xi \times \alpha \beta \rightarrow \Xi \times \alpha \beta
\end{align*}
\]

5.3 Environment and kind checking

\[
\begin{align*}
\vdash \cdot & \quad \vdash \Gamma & \quad \alpha \notin \Gamma \\
\vdash \Gamma & \quad \vdash \Gamma, \alpha : \kappa & \quad \alpha \notin \Gamma \\
\vdash \Gamma, \alpha : \kappa & \quad \Gamma \vdash \tau : \star & \quad \alpha \notin \Gamma \\
\vdash \Gamma, \alpha : \kappa & \quad \vdash \Gamma, \alpha : \kappa & \quad \vdash \Gamma, \alpha : \kappa \\
\vdash \Gamma, \alpha : \kappa & \quad \Gamma \vdash \kappa \rightarrow \kappa' & \quad \Gamma \vdash \kappa \rightarrow \kappa' \\
\vdash \Gamma, \alpha : \kappa & \quad \vdash \Gamma, \alpha : \kappa & \quad \Gamma \vdash \forall x. \kappa
\end{align*}
\]

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

\[
\begin{align*}
\Gamma &\vdash \tau_1 : \star &\Gamma &\vdash \tau_2 : \star &\Gamma &\vdash \{\ell : \tau\} : \star &\Gamma &\vdash \alpha : \kappa &\in \Gamma &\Gamma, \alpha : \kappa &\vdash \tau : \star \\
\Gamma &\vdash \tau_1 \rightarrow \tau_2 : \star &\Gamma &\vdash \{\ell : \tau\} : \star &\Gamma &\vdash \alpha : \kappa &\Gamma &\vdash \alpha : \kappa &\vdash \tau : \star \\
\Gamma &\vdash \forall \kappa.\tau : \star &\Gamma &\vdash \exists^\kappa(\alpha : \kappa).\tau : \star &\Gamma &\vdash \Lambda(\alpha : \kappa).\tau : \kappa \rightarrow \kappa' \\
\Gamma &\vdash \tau_1 : \kappa' \rightarrow \kappa &\Gamma &\vdash \tau_2 : \kappa' &\Gamma &\vdash \tau : \kappa &\Gamma &\vdash \varsigma : \kappa &\Gamma &\vdash \tau : \forall \kappa.\kappa &\Gamma &\vdash \varsigma
\end{align*}
\]

5.5 Term typing

\[
\begin{align*}
\text{F-Var} & &\Gamma &\vdash x : \tau &\Gamma &\vdash x : \tau &\Gamma &\vdash \lambda(x : \tau).e : \tau \rightarrow \tau'
\\
\text{F-Abs} & &\Gamma &\vdash \text{F-App} & &\Gamma &\vdash e_1 : \tau' \rightarrow \tau &\Gamma &\vdash e_2 : \tau' &\Gamma &\vdash e_1 e_2 : \tau \\
\text{F-Record} & &\Gamma &\vdash e : \tau &\Gamma &\vdash \text{#}(\ell) &\Gamma &\vdash \{\ell = e\} : \{\ell : \tau\}
\\
\text{F-Proj} & &\Gamma &\vdash e : \tau &\Gamma &\vdash \{\ell : \tau, \ell_1 : \tau_1\} &\Gamma &\vdash e.\ell : \tau \\
\text{F-Append} & &\Gamma &\vdash e_1 : \{\ell_1 : \tau_1\} &\Gamma &\vdash e_2 : \{\ell_2 : \tau_2\} &\ell_1 &\text{#} \ell_2 &\Gamma &\vdash e_1 \circ e_2 : \{\ell_1 : \tau_1, \ell_2 : \tau_2\}
\\
\text{F-Tapp} & &\Gamma &\vdash e : \forall \kappa.\tau &\Gamma &\vdash \varsigma &\Gamma &\vdash \tau : \sigma[\varsigma \mapsto \alpha] \\
\text{F-Kapp} & &\Gamma &\vdash e : \forall \kappa.\tau &\Gamma &\vdash \varsigma &\Gamma &\vdash \lambda(\alpha : \kappa).e : \forall(\alpha : \kappa).\tau &\Gamma &\vdash \Lambda(\alpha : \kappa).e : \forall \kappa.\tau \\
\text{F-Tabs} & &\Gamma &\vdash e : \forall \kappa.\tau &\Gamma &\vdash \varsigma &\Gamma &\vdash \tau : \sigma[\varsigma \mapsto \alpha] &\Gamma &\vdash \text{pack}(\tau, e) \text{ as } \exists^\tau(\alpha : \kappa).\sigma &\exists^\tau(\alpha : \kappa).\sigma
\\
\text{F-_unpack} & &\Gamma &\vdash e_1 : \exists^\tau(\alpha : \kappa).\tau &\Gamma &\vdash \alpha : \kappa &\Gamma &\vdash x : \tau &\Gamma &\vdash e_2 : \sigma &\Gamma &\vdash \sigma : \star &\Gamma &\vdash \text{unpack}(\alpha, x) = e_1 \text{ in } e_2 : \sigma
\end{align*}
\]
5.6 Transparent existentials

\[ \Gamma \vdash e : u \]

\[
\begin{align*}
\text{F-HIDE} & \\
\Gamma & \vdash \exists^{\tau}(\alpha : \kappa).\sigma : \star & \Gamma & \vdash e : \sigma[\alpha \mapsto \tau] \\
\Gamma & \vdash \text{pack } e \text{ as } \exists^{\tau}(\alpha : \kappa).\sigma : \exists^{\tau}(\alpha : \kappa).\sigma
\end{align*}
\]

\[
\begin{align*}
\text{F-HIDEN} & \\
\Gamma & \vdash e_1 : \exists^{\tau}(\alpha : \kappa).\sigma & \Gamma, \alpha : \kappa, x : \tau & \vdash e_2 : \sigma' \\
\Gamma & \vdash \text{repack}^{\nu}\langle\alpha, x\rangle = e_1 \text{ in } e_2 : \exists^{\tau}(\alpha : \kappa).\sigma'
\end{align*}
\]

\[
\begin{align*}
\text{F-LIFTALL} & \\
\Gamma & \vdash e : \Lambda(\beta : \kappa').\exists^{\tau}(\alpha : \kappa).\sigma \\
\Gamma & \vdash \text{lift}^{\nu} e : \exists^{\nu}(\beta : \kappa').\tau(\alpha' : \kappa' \rightarrow \kappa).\forall(\beta : \kappa').\sigma[\alpha \mapsto \alpha' \beta]
\end{align*}
\]

\[
\begin{align*}
\text{F-SHIELD} & \\
\Gamma & \vdash e \in e_1 : \exists^{\tau}(\alpha : \kappa).\sigma \\
\Gamma & \vdash \text{seal } e : \exists^{\tau}(\alpha : \kappa).\tau'
\end{align*}
\]

\[
\begin{align*}
\text{F-LIFTARR} & \\
\Gamma & \vdash e : \sigma_1 \rightarrow \exists^{\tau}(\alpha : \kappa).\sigma_2 \\
\Gamma & \vdash \text{lift}^{\nu} e : \exists^{\nu}(\alpha : \kappa).\sigma_1 \rightarrow \sigma_2
\end{align*}
\]
6 ELABORATION RULES

6.1 Subtyping

6.1.1 Signature subtyping

\[
\Gamma \vdash C < C' \leadsto f
\]

6.2 Typing

6.2.1 Module typing

\[
\begin{align*}
\text{E-Sub-Sig-Struct} & \quad \overline{D}_0 \subseteq \overline{D} \quad \Gamma \vdash \overline{D}_0 < \overline{D}' \leadsto \overline{f} \quad \overline{f} = \text{dom}(\overline{f}') \\
\text{E-Sub-Sig-GenFct} & \quad \Gamma, \overline{\alpha} \vdash C < C'[\overline{\alpha} \mapsto \overline{\tau}] \leadsto f \\
\text{E-Sub-Sig-AppFct} & \quad \Gamma, \overline{\alpha} \vdash C < C'[\overline{\alpha} \mapsto \overline{\tau}] \leadsto C \leadsto \Gamma, \overline{\alpha} \vdash x.\Lambda \overline{\alpha}. \lambda : \_ . \Lambda \overline{\alpha} . \lambda : C_a . g(x \overline{\tau}(y)) \\
\text{E-Sub-Sig-Struct} & \quad \Gamma \vdash (\exists \overline{\alpha} . C < () \to \exists \overline{\alpha}' . C' \leadsto \lambda x . \lambda . \_ . \text{unpack } (\overline{\alpha}, y) = x () \text{ in pack } (\overline{\tau}, y) \text{ as } \exists \overline{\alpha}' . C' \\
\text{E-Sub-Sig-GenFct} & \quad \Gamma, \overline{\alpha} \vdash C < C'[\overline{\alpha} \mapsto \overline{\tau}] \leadsto f \\
\text{E-Sub-Sig-AppFct} & \quad \Gamma, \overline{\alpha} \vdash C < C'[\overline{\alpha} \mapsto \overline{\tau}] \leadsto C \leadsto \Gamma, \overline{\alpha} \vdash x.\Lambda \overline{\alpha}. \lambda : \_ . \Lambda \overline{\alpha} . \lambda : C_a . g(x \overline{\tau}(y))
\end{align*}
\]

\[
\begin{align*}
\text{E-Sub-Decl-Val} & \quad \Gamma \vdash (\text{val } x : \tau) < (\text{val } x : \tau) \leadsto \lambda x . x \\
\text{E-Sub-Decl-Var} & \quad \Gamma \vdash (\text{type } t = \tau) < (\text{type } t = \tau) \leadsto \lambda x . x \\
\text{E-Sub-Decl-Mod} & \quad \Gamma \vdash (\text{module } X : C) < (\text{module } X : C') \leadsto f \\
\text{E-Sub-Decl-ModType} & \quad \Gamma, \overline{\alpha} \vdash C < C' \leadsto f \\
\text{E-Sub-Decl-GenApp} & \quad \Gamma \vdash (\exists \overline{\alpha} . C \leadsto \overline{\text{pack }} ) \leadsto \exists \overline{\alpha} . C \leadsto \Lambda \overline{\alpha} . \lambda : \text{unpack } (\overline{\alpha}, y) = x () \text{ in pack } (\overline{\tau}, y) \text{ as } \exists \overline{\alpha}' . C' \\
\text{E-Sub-Decl-GenFct} & \quad \Gamma \vdash M : \exists \overline{\alpha} . C \leadsto e \\
\text{E-Sub-Decl-GenApp} & \quad \Gamma \vdash P : () < () \to \exists \overline{\alpha} . C \leadsto e
\end{align*}
\]
Fulfilling OCaml modules with transparency

6.2.2 Binding typing

\[
\Gamma \vdash S : \lambda \alpha.\ C \quad \Gamma \vdash P : C' \rightarrow e \quad \Gamma \vdash C' < C[\alpha \mapsto \tau] \rightarrow f
\]

\[
\Gamma \vdash (P : S) : \exists \forall \tau(\overline{\tau}).C \rightarrow \text{pack } f \ e \ \text{as } \exists \forall \tau(\overline{\tau}).C
\]

\[
\begin{array}{c}
\text{E-Typ-Mod-Ascr} \\
\Gamma \vdash S : \lambda \alpha.\ C \\
\Gamma \vdash P : C' \rightarrow e \\
\Gamma \vdash C' < C[\alpha \mapsto \tau] \rightarrow f
\end{array}
\]

\[
\begin{array}{c}
\text{E-Typ-Mod-Seal} \\
\Gamma \vdash M : \exists \forall \tau(\overline{\tau}).C \rightarrow e \\
\end{array}
\]

\[
\begin{array}{c}
\text{E-Typ-Mod-Struct} \\
\Gamma \vdash A : \exists \forall \alpha.\ \overline{\alpha.\ D} \rightarrow e \\
A \notin \Gamma
\end{array}
\]

\[
\begin{array}{c}
\text{E-Typ-Mod-Proj} \\
\Gamma \vdash \text{module } X : C \in D \\
\tau = \text{fv}(C) \cap \overline{\tau}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash M.X : \exists \forall \alpha.\ C \rightarrow \text{clean} \langle \overline{\alpha}, \overline{\beta} \rangle \ (\text{repack} \langle \overline{\alpha}, x \rangle = e \ \text{in } x.f_x)
\end{array}
\]

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

\[
\begin{align*}
\text{clean}\langle \overline{\alpha}, \overline{\beta} \rangle \ e & \triangleq \text{repack} \langle \alpha, x \rangle = e \ \text{in } \text{clean} \langle \overline{\alpha}, \overline{\beta} \rangle x \\
\text{clean}\langle \overline{\alpha}, \overline{\beta} \rangle \ e & \triangleq \text{unpack} \langle \alpha, x \rangle = e \ \text{in } \text{clean} \langle \overline{\alpha}, \overline{\beta} \rangle x \\
\text{clean}\langle \overline{\alpha}, \overline{\beta} \rangle \ e & \triangleq e
\end{align*}
\]

\[
\begin{array}{c}
\Gamma \vdash A : D \rightarrow e
\end{array}
\]