Fulfilling OCaml modules with transparent existentials

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

ML modules, provided as an additional layer on top of the core language, offer large-scale notions of composition and modularity that have been essential for developing complex applications. While modules are easy to use for common cases, their intensive use may become tricky. Additionally, despite a long line of works, their meta-theory remains difficult to comprehend, requiring heavy machinery to prove their soundness. As a matter of fact, the module layer of OCaml does not currently have a clear 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, excluding abstract signatures but extended with transparent ascription. Building on a previous translation from ML modules to F^{ω}, we introduce an intermediate system, called *canonical*, that mediates between the source language and F^{ω}, keeping the convenient path-based syntactic notation of ML modules with the precise and well-established type-theory of F^{ω}. From the canonical system, we both elaborate terms in F^{ω}, which ensures type soundness, and extract derived ML-style typing rules, which provides a deeper insight into, and better solutions to, the *signature avoidance* problem.

ACM Reference Format:
Clément Blaudeau, Didier Rémy, and Gabriel Radanne. 2023. Fulfilling OCaml modules with transparent existentials. 1, 1 (March 2023), 38 pages.

1 INTRODUCTION

Modularity is a key technique to build software at scale: by breaking down a complex program into smaller parts, it allows several teams of developers to work simultaneously on the same program. An essential property of modularity is *encapsulation*, which allows hiding away implementation details, and expose a simpler, more abstract, external interface. Instead of dealing with technical details and complex invariants at all times, programmers can split the code base into manageable parts, called *modules*, and structure the relationship between those modules by specifying their interfaces and interactions. Code might be packed into a module to make a component, such as the implementation of a data-structure, reusable and often polymorphic—effectively factorizing development—or just to structure the overall program. Details, such as internal invariants which are not revealed by the public signature, are kept hidden 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 languages of the ML-family, modularity is provided by a *module system*, which forms a separate language layer built on top of the core language. The interactions between modules are controlled statically by a strict type system, making modularity work in practice and with little to no run-time overhead. A module is described by its interface, called a *signature*, which serves as both a light specification and an API.

The OCaml module system is especially rich and still receiving new features. It provides both developer-side and user-side abstraction mechanisms: developers can control the outside view of a module by explicitly restricting its interface via *ascription*, while users can abstract over a module with a given interface using a *functor*. The signature language allows to both *restrict* and *control* the interface of a module, specifically by *hiding* fields (making them *private*), or by abstracting type components—keeping types accessible while hiding their definitions.

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

2023. XXXX-XXXX/2023/3-ART $15.00
https://doi.org/

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 [9] where modules and functors are assembled on demand using a DSL [13]. Despite the successes and the interest of the community regarding ML modules, giving them a formal type-theoretic definition and establishing its properties has proven to be a difficult task.

Besides the academic interest, having a formal semantics is made even more necessary to envision extensions such as modular implicits [19] where new modules could be built automatically from their signatures by applications of functors to other modules. Moreover, while we often just need a clear understanding of programs that typecheck, modular implicit also require a clear specification of those that fail to typecheck, so as to ensure coherence.

A particularly successful and elegant approach to model ML module systems, combining ideas from several years of research (see §6), is the elaboration of a significant subset of SML into F^ω done by Rossberg et al. [15]. We built on the insights of their work that we adapted for an OCaml-like language extended with transparent ascription. However, as the elaboration into F^ω introduces encoding artifacts and complexity—especially with applicative functors—we considered a middle point between the OCaml signature syntax and F^ω, which we call the *canonical system*.

Our contributions are:

- A simple specification of a large subset of OCaml modules, including both applicative and generative functors, and extended with transparent ascription, via a signature syntax enriched with F^ω quantifiers.
- A self-contained path-based specification of the same subset with a detailed explanation of its specific mechanisms (notably, strengthening) and its limitations.
- A principled approach, description, and solution to the signature avoidance problem in the presence of applicative functors and transparent ascription.
- The introduction of *transparent* existential types in F^ω, a weaker form of existential types that allows their lifting through arrows types and universal quantifiers.
- A simpler encoding of applicative functors that reduces the difference between applicative and generative functors down to the transparency and opacity nature of existential types.

Plan. The paper is organized as follows. In §2, we give an overview of the key features, strengths and weaknesses of the OCaml module system and present the syntax of our language. In §3, we give a precise, formal specification of our *canonical* system, which bridges the gap between source signatures and F^ω types. We then provide two independent developments based on our canonical system. In §4, we present a similar type system directly on the source syntax, explained through the lens of the canonical system. This includes detailed description and solution to the signature avoidance problem. In §5, we present our novel elaboration of modules into F^ω through the use of transparent existential types. Finally, we discuss related and future works in §6 and §7.

2 A MODERN MODULE SYSTEM

In this section, we introduce some key features of ML module systems and specificity of the OCaml one, extended to handle *transparent ascription*. We end this section with a detailed grammar of the module system we study and a discussion of some technical choices.

2.1 Applicative and generative functors

The notion of functor is central to ML module systems as it allows to define modules parameterized by other modules: it is both the user-side abstraction mechanism and a means of composition. There are two different types of functors, applicative and generative, that correspond to different use-cases and that have different semantics; both are supported by OCaml. Their key difference comes from the way their abstract types are handled.

A *generative functor* is used when the parameterized module is thought of as a *sub-program*. Such module can thus have its own internal state, can product effect or dynamically choose the internal representation of its abstract types. Calling a generative functor twice thus *generates* two incompatible subprograms: their abstract types are incompatible. Programmers can also use generativity to enforce incompatibility between otherwise compatible data-structures that represent different objects in the program.

An *applicative functor* is used when the parameterized module is thought of as a *library*. Such modules should be pure and have made a static choice of internal representation of their abstract types. Calling an applicative functor twice with the *same argument* produces compatible libraries: values from the first can be handled by the second. Modules that provide generic functionalities (such as hash-maps, sets, lists, etc.) are typically written with applicative functors: values and functions contained there-in might still be impure, but, crucially, the functor in itself is pure.

A key aspect regarding applicative functors is their *granularity*: when are two modules considered as *same arguments*, and thus, when should two applications produce compatible abstract types? A first vision is to consider modules similar when they have the same type fields. This vision is *type-safe*, as the abstract types produced by the functor can only depend on the type fields of its parameter, not the values. However, it is not *abstraction-safe*, as the functor can use values of its parameter. However, tracking the equality of values is undecidable in general and, even in a restricted form, tracking the equality of both values and type fields is too fine-grained as modules may have numerous value fields. OCaml follows a compromise, coarser-grained approach to enforce abstraction-safety: tracking equality 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* [6] and has proved effective in the OCaml ecosystem.

2.2 A key strength: module identity

To allow a fine tracking of aliasing, OCaml offers a notion of *module aliases* in its signature language: a module can be indicated as an alias of another module. This concept of *module identity* is central in OCaml, as the applicativity of functors relies on it. However, module aliasing can clash with subtyping, depending on the semantic model. In OCaml, subtyping of modules, which can reorder and delete fields, is not code-free and can induce a runtime copy of the module. In this case, considering a module as an alias of another module while its memory representation is different would be unsound[1]. This problem motivates the current set of ad-hoc restrictions in OCaml, designed to prevent aliases between a functor body and its parameter, as the argument is subtyped at functor application. Interestingly, a known feature, transparent ascription, would solve this issue. As a generalization of aliasing, transparent ascription consists of extending the signature syntax with an entry $(= P < \mathsf{S})$ that stores both the aliasing information (to some path) and the signature.

Transparent ascription is already implemented in SML, and used to hide fields of modules while keeping all type equalities. This is especially useful to prevent writing cumbersome `with type` equalities on signatures. However, another uses-case would also appear in OCaml: the additional identity information obtained via transparent ascriptions would allow multiple applications of an applicative functor to be considered equal (whereas SML only features generative functors). This pattern of code already exists in the OCaml ecosystem [18], we demonstrate it in Figure 1.

[1]Several OCaml bug reports discuss this issue, notably OCaml#7818, OCaml#2051, OCaml#10435 and OCaml#10612.

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

Fig. 1. An example of code pattern where transparent ascription is necessary. On the left-hand side, `VS` defines an interface for vector spaces which contains a sub-module `Scalar` for the field of scalar numbers. Line 7, the functor `LinearAlgebra` uses a vector space to define linear algebra operations, one of them being sets of scalar numbers. At some other point in the development, here on line 11, 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 19, the module `Space3D` implements linear algebra for the vector space $\mathbb{R}^3$. We want the inner sets `Space3D.SSet.t`, and `Set(Reals).t` to be compatible. This is only possible if the aliasing information between the parameter and the body of the functor `Make3D` is kept. Here, transparent ascription is essential to preserve both identity and subtyping information.

Finally, the notion of module identity is also essential to *modular implicits* [19], a proposed feature which aims at leaving implicit some module expressions, 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. With transparent ascription, more module expressions have the same identity and are considered equivalent, reducing false-positive incoherence errors.

2.3 A key weakness: the signature avoidance problem

The *signature avoidance* problem is a key issue of ML-module systems. It originates from a mismatch, illustrated in Figure 2, 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 can't be described by a signature. This issue can be solved either by sticking to the describable space and ensuring that the typechecker correctly covers it or by extending signatures to make the reachable and describable spaces coincide. This mismatch is caused by the interaction of three mechanisms. First, type abstraction, which is key to control access and protect invariants by typing, creates new types that are only compatible with themselves (and their aliases). Second, sharing abstract types between modules, which is essential for module interactions, produces inter-module dependencies. Finally, hiding type or module components (either explicitly, or via subtyping during functor applications) 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 in scope. An example of such pattern is given in Figure 3. Sometimes, no possible signature exists for a module; other times there are several incompatible ones.

Strategies for solving signature avoidance. When a type declaration is referring to an out-of-scope type (and similarly, when a transparent ascription is referring to an out-of-scope module), there are mainly three strategies to correct the signature: (1) removing the dependency (by abstracting the type field), (2) rewriting the type equalities using in-scope aliases (effectively rewiring the dependency tree), or (3) extending the signature syntax with existential types. The first strategy can lead to loss of type equalities, but is easy to implement—it is the one currently in use in the OCaml typechecker. The cases where the second strategy succeeds constitute the *solvable* cases of signature avoidance. The OCaml typechecker only tries to follow directed edges of the dependency

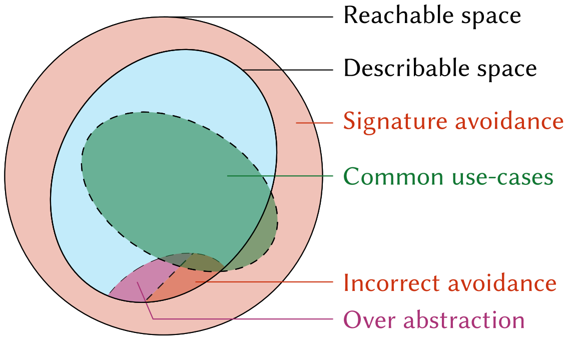
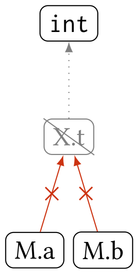


Fig. 2. A representation of the mismatch between the *reachable space* of module expressions (outer-most circle) and the *describable space* of signatures (inner ellipse). The common use-cases of OCaml are mainly within the area where the typechecker behaves correctly. In some cases, the current OCaml typechecker can lose type-equalities while still being in the describable space. This may lead to (1) producing a signature where some type fields are unnecessarily made abstract (over-abstraction) or (2) failing at inferring the signature (incorrect avoidance).

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

```
module type S = sig type t end
module M = (struct
  module X : S = struct
    type t = int
  end
  module Y = struct
    type a = X.t * bool
    type b = X.t * int
  end
end).Y
let f ((x,_): M.a) = ((x, 42): M.b)
```



tree until it finds an accessible type, but does not follow reverse edges and thus does not have a notion of *connected components*. 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.

Signature avoidance in practice. OCaml developers usually get around this limitation by explicitly naming modules before using them, which adds always-accessible root points to the dependency graphs. The module syntax of OCaml actually encourages this approach by limiting the places where inlined, anonymous modules can be used. In particular, projection on an anonymous module (as done in Figure 3) is forbidden. However, explicit naming is sometimes cumbersome, which can limit the usability of module-based programming patterns such as modular implicits. It also prevents a fine-grained management of types shown in public APIs.

2.4 Grammar and technical choices

In this subsection we introduce the language of module expressions and signatures that models the OCaml module system and is used throughout the rest of the paper. Its grammar, given on Figure 4, is built on top of a core language of expressions e and types u which we leave abstract, except for value identifiers x and type identifiers t. We extend expressions and types with qualified accesses.

Syntactic choices. The language of module expressions and signatures is rather standard, except for a few minor design choices:

- We consider the following conventions: 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

Path and Prefix

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

Module Expression

$\mathsf{M} ::= P$	(Path)
$\mid \mathsf{M}.X$	(Projection)
$\mid (P : \mathsf{S})$	(Opaque ascription)
$\mid P()$	(Generative application)
$\mid () \longrightarrow \mathsf{M}$	(Generative Functor)
$\mid (Y : \mathsf{S}) \longrightarrow \mathsf{M}$	(Applicative Functor)
$\mid \mathtt{struct}_A\ \overline{\mathsf{B}}\ \mathtt{end}$	(Structure)

Binding

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

Signature

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

Declarations

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

Identifier

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

Core language

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

Fig. 4. Syntax of the module language

object; their binding occurrence appears as a subscript to the structure or signature they belong to, so that self-references can freely be renamed.

- Prefixes, written with the letter Q, range over either a path P or a self reference A.
- *Abstract types* are specified as types pointing to themselves, e.g., $\mathtt{type}\ t = A.t$ where A is the self-reference of the current structure.
- As a convention, bindings and declarations can be seen of the form $I : \mathsf{B}$ and $I : \mathsf{D}$, with the identifier extracted i.e., a type binding $\mathtt{type}\ t = \mathsf{u}$ can be seen as $t : \mathtt{type}\ \mathsf{u}$.
- 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.

Projections and accesses. Several restrictions are built into the grammar. First, field accesses inside a module *type* are forbidden, since prefixes Q may not originate from a module type identifier T. Second, we allow projection on any module expression, but we restrict functor application to paths. Current OCaml does the opposite, mainly to prevent cases prone to trigger signature avoidance. Our choice is more general, as the OCaml one can be encoded, while the converse requires an explicit signature annotation on the functor argument. That is, we may see $\mathsf{M}(\mathsf{M}')$ as syntactic sugar for $(\mathtt{struct}_A\ \mathtt{module}\ F = \mathsf{M}\ \mathtt{module}\ X = \mathsf{M}'\ \mathtt{module}\ \mathrm{Res} = A.F(A.X)\ \mathtt{end})\,.\mathrm{Res}$.

Transparent ascription. In our grammar, transparent ascription is only available in signatures. Transparent ascription in expressions can be easily encoded as an opaque ascription of a transparent signature: $(P < \mathsf{S})$ is syntactic sugar for $(P : (= P < S))$.

Omitted features. We omit some constructs for the sake of simplicity: the include and open operators, explicit constraints "S with type $t = \mathsf{u}$" (resp. module $X = P$) and deleting constraints

Canonical Types

$$\begin{array}{rll} \tau ::= & \alpha & \text{(Existential identifier)} \\ \mid & \alpha(\overline{\tau}) & \text{(Type application)} \end{array}$$

Environment

$$\begin{array}{rll} \Gamma ::= & \varnothing & \text{(Empty)} \\ \mid & \Gamma, \overline{\alpha} & \text{(Abstract types)} \\ \mid & \Gamma, (Y : \mathcal{C}) & \text{(Functor Argument)} \\ \mid & \Gamma, (A.I : \mathcal{D}) & \text{(Declaration)} \end{array}$$

Mode

$$\begin{array}{rll} \diamond ::= & \triangledown & \text{(Applicative)} \\ \mid & \blacktriangledown & \text{(Generative)} \end{array}$$

Identity signatures

$$\mathcal{C} ::= (\tau, \mathcal{R})$$

Concrete signatures

$$\begin{array}{rll} \mathcal{R} ::= & \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} & \text{(Structural signature)} \\ \mid & \forall \overline{\alpha}.\mathcal{C} \to \mathcal{C} & \text{(Applicative functor)} \\ \mid & () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C} & \text{(Generative functor)} \end{array}$$

Declarations

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

Fig. 5. Syntax of *canonical* signatures

"S with type t := u" (resp. module $X := P$), and the "module type of" operator. We believe that they do not impact the overall structure of the system, only adding more cases in the set of rules. We also omit first-class modules, abstract signatures, and recursive modules.

3 A QUANTIFIER-BASED APPROACH: THE CANONICAL SYSTEM

In this section, we present a typing system, called *canonical*, that covers the set of features informally explained in the previous section and that does not suffer from the signature avoidance problem. The signatures produced by this system are written in a more expressive syntax, with explicit F^ω-style type binders (existential, universal, lambda). The key interest of this system is to provide a description of the mechanisms behind the main features (type abstraction, module identity, applicativity/generativity) via F^ω, while hiding the complexity and artifacts that come from the encoding of module expressions in F^ω.

3.1 Canonical system overview

The syntax for source modules and signatures remains the same. New syntactical categories, described in Figure 5, the canonical language (and declarations) used for inferred signatures. By convention, we use curvy capitals ($\mathcal{C}$, $\mathcal{R}$, $\mathcal{D}$, ...) for canonical objects. We distinguish between three (mutually defined) types of canonical signatures enforcing a key invariant: the places where quantifiers are allowed. *Identity* signatures $\mathcal{C}$ combine an identity type τ with a *concrete* signature $\mathcal{R}$ that contains the structural information (functors, signature), but no quantification. In the judgments, identity signatures are handled with existential or lambda quantifiers in front. Existential quantifiers are marked with a mode, either generative or applicative, that will be used during typing. With this scheme, existential quantification is restricted to the front of a whole signature or to the top-level of the body of *generative* functors in generative mode. Specifically, submodule declarations do not have existential quantification. Universal quantifiers appear in front of applicative functors to capture polymorphism from the abstract types of their parameter. Finally, signatures stored in module type declarations are parameterized by their abstract types with F^ω-style λ-binding.: they can later be existentially or universally quantified, depending on context. Module types variables are inlined in concrete signatures and thus don't appear there.

Finally, environments contain bindings with the identifier extracted and prefixed by a self-reference: $A.I : \mathcal{D}$. To prevent shadowing, we consider a *wellformedness* predicate over environments wf(Γ), defined to check that identifiers appear at most once. As a simplifying convention for the rest of this paper, we add wellformedness of the environment as a precondition to all rules.

System structure and judgments. The canonical system has three judgments:

Signature typing $\Gamma \overset{\text{can}}{\vdash} \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}$ translates a source signature S into its canonical counterpart $\lambda\overline{\alpha}.\mathcal{C}$, making the set of type parameters $\overline{\alpha}$ explicit. It is also defined for declarations.

Subtyping $\Gamma \overset{\text{can}}{\vdash} \mathcal{C} \prec \mathcal{C}'$ checks that a signature $\mathcal{C}$ is *more restrictive* than a signature $\mathcal{C}'$, meaning that the former has more fields and introduces less abstract types. Equivalently, it means that a module with signature $\mathcal{C}$ can be cast into a module of signature $\mathcal{C}'$. This judgment is defined on $\mathcal{C}$, $\mathcal{R}$, and declarations $\mathcal{D}$.

Module typing $\Gamma \overset{\text{can}}{\vdash} \mathsf{M} : \exists^{\diamond}\overline{\alpha}.\mathcal{C}$ infers the canonical signature of a source module M. The signature $\exists^{\diamond}\overline{\alpha}.\mathcal{C}$ features an existential quantification that can be in applicative of generative mode.

In the following subsections, we first explain the signature typing judgment, as it illustrates the key concepts of the canonical system; then we show the subtyping and module typing judgments. We only show the key rules of each judgment, leaving the full set of rules in the appendix (§A).

3.2 Signature typing, the key concepts of the canonical system

We progressively introduce the key concepts of the system and highlight the relevant inference rules. Let us start at the declaration level, inside a structural signature sig $\overline{\mathsf{D}}$ end.

3.2.1 Abstract types and extension of scopes. The syntactic enforcement of the position of quantifiers in the canonical system helps simplify the presentation. The intuition is that an abstract type field `type` t actually introduces a new abstract type α that is accessible not only in the local module, but in all the following ones. Therefore, the binder for that abstract type α must be *lifted* to enclose the whole region where α is accessible. This *existential lifting*, pervasive throughout the declaration typing rules, is a key concept. It is illustrated in the following rules. First, in Rule C-Typ-Decl-TypeAbs, an abstract type declaration `type` $t = A.t$ is converted into a normal type declaration type $t = \alpha$, with a new abstract type quantified with a lambda binder. Then, in Rule C-Typ-Decl-Mod when typing a submodule declaration, a set of abstract types quantified by the submodule signature is extracted and put in front of the declaration. Finally, in Rule C-Typ-Decl-Seq, the types $\overline{\alpha}_1$ introduced by the first declaration D are put in the context for typing the rest of declarations $\overline{\mathsf{D}}$. Then, both sets of abstract types $\overline{\alpha}_1$ and $\overline{\alpha}$ are merged together in front of the list of canonical declarations $\mathcal{D}_1, \overline{\mathcal{D}}$.

C-Typ-Decl-TypeAbs
$$\Gamma \overset{\text{can}}{\vdash}_A (\mathsf{type}\ t = A.t) : \lambda\alpha.\,(\mathsf{type}\ t = \alpha)$$

C-Typ-Decl-Mod
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \overset{\text{can}}{\vdash}_A (\mathsf{module}\ X : \mathsf{S}) : \lambda\overline{\alpha}.\,(\mathsf{module}\ X : \mathcal{C})}$$

C-Typ-Decl-Seq
$$\frac{\Gamma \overset{\text{can}}{\vdash}_A \mathsf{D} : \lambda\overline{\alpha}_1.\mathcal{D} \qquad \Gamma, \overline{\alpha}_1, A.I : \mathcal{D} \overset{\text{can}}{\vdash}_A \overline{\mathsf{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\vdash}_A \mathsf{D}, \overline{\mathsf{D}} : \lambda\overline{\alpha}_1\,\overline{\alpha}.\mathcal{D}, \overline{\mathcal{D}}}$$

3.2.2 At the signature level: module identities. At the signature level, the mechanism for abstract types can be reused to track module identities by adjoining every concrete signature $\mathcal{R}$ with an identity type to compose an *identity signature* $\mathcal{C}$. Fresh identities are introduced when typing a *new* signature, that is, a functor or a structural signature, as in Rule C-Typ-Sig-Str where a fresh identity α_0 is parameterized. By contrast in Rule C-Typ-Sig-TrAscr, the identity type is preserved when typing a transparent ascription signature of some path $(= P < \mathsf{S})$. This can be seen by the fact that no abstract type is quantified in front, preserving both the identity type and all type definitions of $\mathcal{C}$.

C-Typ-Sig-Str
$$\frac{\Gamma \overset{\text{can}}{\vdash}_A \overline{\mathsf{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash} \mathsf{sig}_A\ \overline{\mathsf{D}}\ \mathsf{end} : \lambda\alpha_0, \overline{\alpha}.\,(\alpha_0, \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end})}$$

C-Typ-Sig-TrAscr
$$\frac{\Gamma \overset{\text{can}}{\vdash} P : \mathcal{C} \qquad \Gamma \overset{\text{can}}{\vdash} \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}' \qquad \Gamma \overset{\text{can}}{\vdash} \mathcal{C} \prec \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]}{\Gamma \overset{\text{can}}{\vdash} (= P < \mathsf{S}) : \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]}$$

3.2.3 Functors and higher-order abstract types. Rule C-Typ-Sig-GenFct for generative functors shows how the scope of abstract types is limited: the lifting of abstract type quantifiers, $\overline{\alpha}$ here, stops at the top-level of the functor body. The lambda-abstraction is turned into an existential one, with a generative mode. Hence, every instantiation of the functor will *generate* new (incompatible) types $\overline{\alpha}$. Rule C-Typ-Sig-AppFct for applicative functors shows some key mechanisms. Unlike with generative functors, two applications of the same functor to the same argument should produce the same abstract types, which implies that the scope of those types should (at least) contain both applications. A known solution to this problem is to lift the quantified abstract types $\overline{\beta}$ outside of the functor via a *skolemisation*, turning them into higher-order types $\overline{\beta'}$, as we do here.

C-Typ-Sig-GenFct
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \overset{\text{can}}{\vdash} () \to \mathsf{S} : \lambda\alpha_0.\left(\alpha_0, () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}\right)}$$

C-Typ-Sig-AppFct
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{S}_a : \lambda\overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a \overset{\text{can}}{\vdash} \mathsf{S} : \lambda\overline{\beta}.\mathcal{C}}{\Gamma \overset{\text{can}}{\vdash} (Y : \mathsf{S}_a) \to \mathsf{S} : \lambda\alpha_0, \overline{\beta'}.\left(\alpha_0, \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]\right)}$$

3.3 Subtyping and Typing: a parsimonious system

Subtyping. The subtyping judgment is standard for ML-module systems, we highlight here some key rules. When comparing identity signatures in Rule C-Sub-Sig-Id, the identity types are required to be the same. This is crucial to enforce that a transparent ascription keeps the same identity as the original module. When comparing two structural signatures in Rule C-Sub-Sig-Struct, deletion and reordering of fields is allowed, as a subset of the left-hand side declarations is compared against the full set of right-hand side ones. Finally, Rule C-Sub-Sig-GenFct for generative functors features an instantiation of the abstract types before subtyping the identity signatures $\mathcal{C}$ and $\mathcal{C}'$. Finding such instantiation in an algorithmic way, with higher-order abstract types, is not easy. We believe that the argument used by [15] applies here, as the same notions of *rooted* and *explicit* can be defined with canonical signatures.

C-Sub-Sig-Id
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathcal{R} < \mathcal{R}'}{\Gamma \overset{\text{can}}{\vdash} (\tau, \mathcal{R}) < (\tau, \mathcal{R}')}$$

C-Sub-Sig-Struct
$$\frac{\overline{\mathcal{D}_0} \subseteq \overline{\mathcal{D}} \qquad \Gamma \overset{\text{can}}{\vdash} \overline{\mathcal{D}_0} < \overline{\mathcal{D}}'}{\Gamma \overset{\text{can}}{\vdash} \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} < \mathsf{sig}\ \overline{\mathcal{D}}'\ \mathsf{end}}$$

C-Sub-Sig-GenFct
$$\frac{\Gamma, \overline{\alpha} \overset{\text{can}}{\vdash} \mathcal{C} < \mathcal{C}'\left[\overline{\alpha}' \mapsto \overline{\tau}\right]}{\Gamma \overset{\text{can}}{\vdash} () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} < () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}'}$$

Existential modes. The typing judgment $\Gamma \overset{\text{can}}{\vdash} \mathsf{M} : \exists^{\diamond}\overline{\alpha}.\mathcal{C}$ produces a signature with existential quantification, either in applicative or generative mode. The mode prevents core and module expressions that are inherently generative, such as unpacking a first-class module, calling a generative functor, or computing impure values, from appearing in the body of applicative functors. This discipline is enforced by typing a generative (resp. applicative) functor application in *generative* (resp. applicative) mode in Rule C-Typ-Mod-AppGen (resp. C-Typ-Mod-AppFct).

C-Typ-Mod-AppGen
$$\frac{\Gamma \overset{\text{can}}{\vdash} P : \left(\_, () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}\right)}{\Gamma \overset{\text{can}}{\vdash} P() : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}$$

C-Typ-Mod-AppFct
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{S}_a : \lambda\overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, (Y : \mathcal{C}_a) \overset{\text{can}}{\vdash} \mathsf{M} : \exists^{\triangledown}\overline{\beta}.\mathcal{C}}{\Gamma \overset{\text{can}}{\vdash} (Y : \mathsf{S}_a) \to \mathsf{M} : \exists^{\triangledown}\alpha_0, \overline{\beta'}.\ \left(\alpha_0, (\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C})\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]\right)}$$

In addition, Rule C-Typ-Bind-Seq for bindings forces all the components of a structural signature to have the same mode. We also rely on a core-language expression typing judgment[2] $\Gamma \overset{\text{can}}{\vdash}{}^{\diamond} \mathsf{e} : \tau$ equipped with a mode that tracks the presence of effects[3]. When typing a value field, the mode is propagated via an empty existential (Rule C-Typ-Bind-Let). Finally, the mode of a module can be *downgraded* from applicative to generative via Rule C-Typ-Mod-Mode. All other rules are agnostic

[2]This judgment is trivially extended by rules for accessing module paths; the added rules are given in A.2.
[3]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.

of the typing mode.

C-Typ-Bind-Seq
$$\frac{\Gamma \overset{\text{can}}{\vdash}_A \mathtt{B} : \exists^{\Diamond}\overline{\alpha}_1.\mathcal{D} \qquad \Gamma, \overline{\alpha}_1, A.I : \mathcal{D} \overset{\text{can}}{\vdash}_A \overline{\mathtt{B}} : \exists^{\Diamond}\overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\vdash}_A \mathtt{B}, \overline{\mathtt{B}} : \exists^{\Diamond}\overline{\alpha}_1, \overline{\alpha}.\mathcal{D}, \overline{\mathcal{D}}}$$

C-Typ-Bind-Let
$$\frac{\Gamma \overset{\text{can}\,\Diamond}{\vdash} \mathtt{e} : \tau}{\Gamma \overset{\text{can}}{\vdash}_A (\mathtt{let}\ x = \mathtt{e}) : \exists^{\Diamond}.(\mathsf{val}\ x : \tau)}$$

C-Typ-Mod-Mode
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathtt{M} : \exists^{\nabla}\overline{\alpha}.\mathcal{C}}{\Gamma \overset{\text{can}}{\vdash} \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}$$

Module identities. Similarly to signature typing, fresh identities are introduced when typing a functor (rules C-Typ-Mod-GenFct and C-Typ-Mod-AppFct, above) and a structure (rule C-Typ-Mod-Struct). In addition, applications of a generative functor (rules C-Typ-Mod-AppGen, above) and ascriptions (Rule C-Typ-Mod-Ascr) *can* introduce new identities, depending on whether or not the identity type is a parameterized variable. If the identity type is not parameterized, it means that the functor is actually concrete (does not introduce new types nor new identities) or that the ascription is transparent. If the identity type is parameterized, it means that the functor or the ascription introduces a new, fresh identity. In contrast, when typing an application of an applicative functor (Rule C-Typ-Mod-AppApp), no new identity is introduced: two applications of similar functors to modules with the same identity arguments produce results with the same identity.

C-Typ-Mod-Struct
$$\frac{\Gamma \overset{\text{can}}{\vdash}_A \overline{\mathtt{B}} : \exists^{\Diamond}\overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash} \mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end} : \exists^{\Diamond}\alpha_0, \overline{\alpha}.\ (\alpha_0, \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end})}$$

C-Typ-Mod-Ascr
$$\frac{\Gamma \overset{\text{can}}{\vdash} P : \mathcal{C} \qquad \Gamma \overset{\text{can}}{\vdash} \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}' \qquad \Gamma \overset{\text{can}}{\vdash} \mathcal{C} < \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]}{\Gamma \overset{\text{can}}{\vdash} (P : \mathtt{S}) : \exists^{\nabla}\overline{\alpha}.\mathcal{C}'}$$

C-Typ-Mod-AppApp
$$\frac{\Gamma \overset{\text{can}}{\vdash} P : (\_, \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}) \qquad \Gamma \overset{\text{can}}{\vdash} P' : \mathcal{C}' \qquad \Gamma \overset{\text{can}}{\vdash} \mathcal{C}' < \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}]}{\Gamma \overset{\text{can}}{\vdash} P(P') : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]}$$

C-Typ-Mod-GenFct
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathtt{M} : \exists^{\Diamond}\overline{\alpha}.\mathcal{C}}{\Gamma \overset{\text{can}}{\vdash} () \to \mathtt{M} : \exists^{\nabla}\alpha_0.\ \big(\alpha_0, () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}\big)}$$

Projection and signature avoidance. In the source signatures, dependencies between modules are hard to track as modules can use arbitrary paths to access other modules. Signatures can thus have complex internal dependencies, making the projection of a submodule $\mathtt{M}.X$ delicate: the dependencies of X might become dangling after the other components of signature of $\mathtt{M}$ are lost. In canonical signatures however, dependencies only consist of the use of a common abstract type that has been lifted in front. Thus, canonical signatures do not have internal dependencies and so, do not need a self-reference. The projection rule C-Typ-Mod-Proj simply keeps the set of abstract types $\overline{\alpha}$ to ensure that the submodule signature $\mathcal{C}$ has no dangling dependency.

$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathtt{M} : \exists^{\Diamond}\overline{\alpha}.\ (\_, \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}) \qquad \mathsf{module}\ X : \mathcal{C} \in \overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\vdash} \mathtt{M}.X : \exists^{\Diamond}\overline{\alpha}.\mathcal{C}} \qquad \text{(C-Typ-Mod-Proj)}$$

The canonical system serves as a basis for understanding OCaml modules, as it models key mechanisms with the rigor and simplicity of F^{ω} types, without the encoding artifacts of F^{ω} and without the user-friendly but problematic path-based presentation of the source OCaml.

4 A PATH-BASED APPROACH: THE SOURCE SYSTEM

In this section, we use the insights of the canonical system to build a self-contained *source system* that uses the source signatures $\mathtt{S}$ both as internal representation (in the typing environment) and as result of the inference. The source system models the behavior of OCaml-style modules and could be presented standalone. However, it mimics the mechanisms of the canonical system but without the flexibility of explicit quantifiers, and is thus cumbersome. Hence, the presentation of the source system as derived from the canonical system helps understand it.

The presentation of the canonical system started with the translation of source signatures into canonical ones, which illustrates how the canonical system models the source language. We now follow the reverse approach: to present the judgments and rules of the source system, we actually

Concrete signature

$$
\begin{array}{lll}
\mathbf{R} ::= \mathtt{sig}\ \bar{\mathbf{D}}\ \mathtt{end} & & \text{(Structural signature)} \\
\quad \mid () \to \mathsf{S} & & \text{(Generative functor)} \\
\quad \mid (Y : \mathsf{S}) \to \mathbf{C} & & \text{(Applicative functor)}
\end{array}
$$

Identity signature

$$\mathbf{C} ::= (= P < \mathbf{R})$$

Concrete declarations

$$
\begin{array}{lll}
\mathbf{D} ::= \mathtt{val}\ x : \mathsf{u} & & \text{(Value)} \\
\quad \mid \mathtt{type}\ t = \mathsf{u} & & \text{(Type)} \\
\quad \mid \mathtt{module}\ X : \mathbf{C} & & \text{(Submodule)} \\
\quad \mid \mathtt{module\ type}\ T = \mathsf{S} & & \text{(Module type)}
\end{array}
$$

Fig. 6. Source signature sub-categories. Identity and concrete signatures are subsets of source signatures S, concrete declarations are a subset of source declarations D.

$$
\begin{array}{lll}
\Gamma_{src} ::= \varnothing & & \text{(Empty)} \\
\quad \mid \Gamma_{src}, (Y : \mathbf{C}) & & \text{(Functor Argument)} \\
\quad \mid \Gamma_{src}, A.I : \mathbf{D} & & \text{(Declaration)}
\end{array}
$$

Fig. 7. Source typing environment Γ_{src}

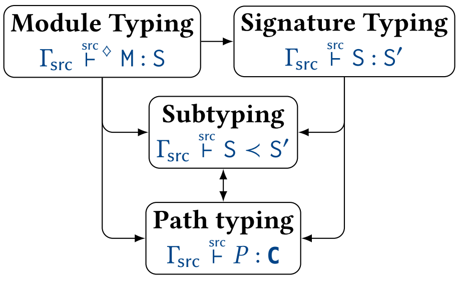

Module Typing
$\Gamma_{src} \overset{src}{\vdash}^{\diamond} \mathsf{M} : \mathsf{S}$
Signature Typing
$\Gamma_{src} \overset{src}{\vdash} \mathsf{S} : \mathsf{S}'$
Subtyping
$\Gamma_{src} \overset{src}{\vdash} \mathsf{S} < \mathsf{S}'$
Path typing
$\Gamma_{src} \overset{src}{\vdash} P : \mathbf{C}$

Fig. 8. Structure of judgments for the source system, all depending on strengthening.

focus on the process of translating canonical signatures *back into the source syntax*, which we call *anchoring*. The anchoring is defined on canonical signatures, which have been obtained either by canonical signature typing or by module type inference. In the former case, anchoring is relatively easy. In the latter case however, the signature avoidance problem arises, making it non-obvious whether a corresponding source signature even exists and how to reconstruct it. In a nutshell, the anchoring aims at *reverting* the existential lifting that happens during the canonical typing. The core of this process is to build an *anchoring map* θ that associates to every abstract type α a corresponding valid path $P.t$ in the source signature. We define the three following judgments:

Signature anchoring $\Gamma \overset{\text{anch}}{\vdash} \mathcal{C} \overset{P?}{\hookrightarrow} \Gamma_{src}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \theta$ takes a canonical signature $\mathcal{C}$ (or *resp.* a declaration $\mathcal{D}$), an environment Γ and its source anchoring Γ_{src}, along with the anchoring map of the environment θ_Γ; it produces a source signature S (or *resp.* a declaration D) and its associated anchoring map θ. The judgment is parameterized by an optional path (written P?): if a path is given, it will prefix all anchorings paths of the resulting map θ.

Type anchoring $\Gamma \overset{\text{anch}}{\vdash} \tau \hookrightarrow \Gamma_{src}; \theta_\Gamma \vdash z$ takes a canonical type τ (and Γ_{src}, Γ, and θ_Γ as above) and returns an anchor z that is either a path $P.t$ when τ is an abstract type or a pair $(P, \mathcal{R})$ when τ is an identity type.

Environment anchoring $\Gamma \hookrightarrow \Gamma_{src} : \theta_\Gamma$ takes a canonical environment Γ and returns a source environment Γ_{src} with its associated anchoring map θ_Γ.

In the rest of this section, we present conjointly the anchoring judgments and the building blocks of a source-only typing system. We first explain the structure of the source typing system. Then, in each following subsection, we explore a *difference of expressivity* between the canonical and source signature syntaxes, regarding, the management of, successively, (1) abstract types, (2) module identities, and (3) higher-order types. From each difference, we extract an *anchoring principle*. We then show how the application of the principle can be seen in the anchoring judgment and how it impacts the design of typing in the source system.

4.1 Source system overview

The source system reuses the grammar of Figure 4. However, to ease the enforcement of some invariants, we introduce syntactical subcategories of the source signature syntax, which are summed up in Figure 6. They mimic the corresponding canonical signatures categories. Specifically, concrete signatures $\mathbf{D}$ do not introduce new abstract types, which can be seen in structural signatures ($\mathtt{sig}\ \overline{\mathbf{D}}\ \mathtt{end}$): they do not feature a self-reference and thus forbids self-referring fields; all paths (for types and modules) must be absolute ones.

The source system is built on four judgments illustrated in Figure 8 and a *strengthening* operation:

Signature typing $\Gamma_{\mathrm{src}} \overset{\mathrm{src}}{\vdash} \mathsf{S} : \mathsf{S}'$ checks the wellformedness of a source signature S and returns the signature with inlined module types and type aliases S'.

Subtyping $\Gamma_{\mathrm{src}} \overset{\mathrm{src}}{\vdash} \mathsf{S} < \mathsf{S}'$ checks that a signature S is *more restrictive* than a signature S', meaning that the former has more fields and introduces less abstract types. The judgment is defined over both identity and concrete signatures. We also introduce a subset of the subtyping relation, called *abstraction-subtyping*, written $\Gamma_{\mathrm{src}} \overset{\mathrm{src}}{\vdash} \mathsf{S} \sqsubset \mathsf{S}'$, where both signatures must have the same fields in the same order, but may differ in their abstract type fields.

Module typing $\Gamma_{\mathrm{src}} \overset{\mathrm{src}}{\vdash}^{\diamond} \mathsf{M} : \mathsf{S}$ infers the signature S of a module M, given a typing mode $\diamond$.

Path typing $\Gamma_{\mathrm{src}} \overset{\mathrm{src}}{\vdash} P : \mathbf{C}$ retrieves a signature $\mathbf{C}$ for a module path P. It is a subset of the more general *module typing*, extracted from the latter to prevent circular dependencies.

Strengthening $\mathsf{S}/P \gg \mathbf{C}$ rewrites fields in S, i.e., abstract type into concrete type and modules into transparent ascriptions pointing to paths prefixed by P. The resulting signature is therefore an identity signature, with a concrete content.

Simplifying assumptions. In order to limit the complexity of the source system, we removed the three sources of *indirections* in the inferred signatures. (1) Type aliases are inlined, making type definitions always use the oldest available alias of a type: it simplifies the equality checking of two types. (2) Transparent ascriptions of transparent ascriptions are inlined, making all modules sharing the same identity transparent ascriptions of a single root module. (3) Module types are inlined to have direct access to their content. In a real-world implementation, the latter would pose a usability problem, as module types can have hundreds of fields and inlining them would make signatures unreadable. We consider this problem orthogonal to the formalization (and thoroughly explored in the OCaml implementation). Finally, our presentation is purposefully not algorithmic.

4.2 The issue of abstract type fields

A key insight is the difference in the source syntax between the declaration of a *concrete* type ($\mathtt{type}\ t = \mathsf{u}$) and that of an *abstract* type ($\mathtt{type}\ t = A.t$). An abstract type declaration states the *introduction* of both a *new abstract type* and a *new field* that may later be used as a handle to refer to this abstract type. For a signature used in a covariant position, an abstract type declaration effectively *creates* a new abstract type (using existential quantification) and *adds* a type field (structural information) to the signature. By contrast, a concrete type definition only introduces structural information—adding a field to refer to an existing type definition.

A canonical signature *separates* the quantification information (existential, universal, or lambda binders) from the structural information (fields). Thus, it makes the scope of abstract types explicit, using the explicit binders. More importantly, it allows referring to types that do not have a handle, while the source system cannot.

1st Anchoring Principle. Source signatures can only express module types whose structure coincide with the quantification, i.e. where the first occurrence of every abstract type α is of the form $\mathtt{type}\ t = \alpha$. Such a declaration is called an *anchoring point* for the type α.

4.2.1 Consequence for the anchoring judgment. The first step in translating signatures is thus to identify the anchoring point of every parameterized abstract type. We do so by going through signatures (and declarations) while building the *anchoring map* θ, which is extended whenever an anchoring point is encountered. A simplified version of the rule for an anchoring point (ignoring high-order types) is:

$$\frac{\alpha \notin \mathrm{dom}(\theta_\Gamma) \qquad \Gamma \hookrightarrow \Gamma_{\mathrm{src}} : \theta_\Gamma}{\Gamma \overset{\mathrm{anch}}{\vdash} \mathsf{type}\ t = \alpha \hookrightarrow \Gamma_{\mathrm{src}}; \theta_\Gamma \vdash \mathtt{type}\ t = A.t : (\alpha \mapsto t)} \quad \text{(A-Decl-Anchor-Simplified)}$$

Here, the trivial anchoring map $\alpha \mapsto t$ is returned. By contrast, the empty map $\varnothing$ is returned when reaching a declaration that is not an anchoring point, i.e., a type declaration (Rule A-Decl-Type) or a value declaration (Rule C-Ach-Decl-Val). Instead, the anchoring map of the environment θ_Γ is used to translate the canonical type τ into a source type u, via the type anchoring judgment.

A-Decl-Type

$$\frac{\Gamma \overset{\mathrm{anch}}{\vdash} \tau \hookrightarrow \Gamma_{\mathrm{src}}; \theta_\Gamma \vdash \mathsf{u}}{\Gamma \overset{\mathrm{anch}}{\vdash} \mathsf{type}\ t = \tau \hookrightarrow \Gamma_{\mathrm{src}}; \theta_\Gamma \vdash \mathtt{type}\ t = \mathsf{u} : \varnothing}$$

A-Decl-Val

$$\frac{\Gamma \overset{\mathrm{anch}}{\vdash} \tau \hookrightarrow \Gamma_{\mathrm{src}}; \theta_\Gamma \vdash \mathsf{u}}{\Gamma \overset{\mathrm{anch}}{\vdash} \mathsf{val}\ x : \tau \hookrightarrow \Gamma_{\mathrm{src}}; \theta_\Gamma \vdash (\mathtt{val}\ x : \tau) : \varnothing}$$

4.2.2 Abstract type fields in the source system. In the source syntax, abstract type fields thus play a special role, as they state the introduction of a new type, and in addition, acts as an implicit quantifier. As consequence, retrieving a signature from the environment naively could duplicate the quantification information and *re-introduce* new (incompatible) abstract types.

Strengthening – $\mathsf{S}/P \gg \mathbf{C}$. To prevent this, a standard solution is to define a *strengthening* operator that rewrites all local links of S (notably, abstract type fields) into absolute links prefixed by P. Similarly, modules fields are rewritten as transparent ascriptions of their counterpart in P to prevent duplication of identities. The returned signature $\mathbf{C}$ is itself a transparent ascription of P. We show here two key rules. Rule S-Str-Sig-Struct deals with structural signatures: the self reference A is replaced by P, which amounts to replacing all local links prefixed by the self-reference A by absolute ones prefixed by the path P. Rule S-Str-Decl-Mod for strengthening declarations by a path P replaces the signature S of a submodule X by the strengthening of S by the path $P.X$. The full set of rules is given in §C.1.

S-Str-Sig-Struct

$$\frac{\overline{\mathsf{D}}[A \mapsto P]\ /P \gg \overline{\mathbf{D}}}{\mathtt{sig}_A\ \overline{\mathsf{D}}\ \mathtt{end}/P \gg (= P < \mathtt{sig}\ \overline{\mathbf{D}}\ \mathtt{end})}$$

S-Str-Decl-Mod

$$\frac{\mathsf{S}/P.X \gg \mathbf{C}}{(\mathtt{module}\ X : \mathsf{S})\ /P \gg \mathtt{module}\ X : \mathbf{C}}$$

Strengthening is used in all judgments to enforce a key invariant: to unify the representation of abstract types and of modules with a fresh identity, we strengthen the signatures and declarations before pushing them in the context. In signatures, we represent abstract types as (local) pointers to themselves (of the form $\mathtt{type}\ t = A.t$). Similarly, in the environment, we represent abstract types as absolute pointers to themselves (of the form $\mathtt{type}\ t = P.t$) and modules declarations as transparent ascription, possibly to themselves (of the form $\mathtt{module}\ X : (= P < \mathbf{R})$).

4.3 The treatment of module identities

The canonical signature syntax can express identity sharing between modules regardless of their signatures, as long as they share the same identity type. In contrast, source signatures, as they rely solely on transparent ascriptions, can only express identity sharing when all modules sharing the same identity have a signature that is a subtype of the (syntactical) first occurrence. As transparent ascription includes a subtyping check, it is a more restricted form of identity sharing than the identity types of the canonical system.

2nd Anchoring Principle. Source signatures can only express identity sharing via transparent ascription. All modules sharing the same identity must have signatures that are suptypes of the first occurrence. The *absence* of transparent ascription expresses the freshness of identity.

4.3.1 Consequence for the anchoring judgment. The only conditions for anchoring an abstract type are checked at its anchoring point, and then all usage points are unrestricted. By contrast, when anchoring a module identity α, the conditions at the anchoring point are not sufficient. After anchoring α at a module definition with a signature $\mathcal{C}$, all usage points must also be at module definitions with a signature that is a subtype of $\mathcal{C}$. To account for this, we extend the anchoring map to store both a module path *and* the signature at the anchoring point for identity types.

A-Sig-Id-TrAsrc

$$\frac{\Gamma \overset{\text{anch}}{\vdash} \tau \hookrightarrow \Gamma_{\text{src}}; \theta_\Gamma \vdash (P, \mathcal{R}') \qquad \Gamma \overset{\text{anch}}{\vdash} \mathcal{R} \overset{P?}{\hookrightarrow} \Gamma_{\text{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \varnothing \qquad \Gamma \overset{\text{can}}{\vdash} \mathcal{R}' < \mathcal{R}}{\Gamma \overset{\text{anch}}{\vdash} (\tau, \mathcal{R}) \overset{P?}{\hookrightarrow} \Gamma_{\text{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} (= P < \mathsf{S}) : \varnothing}$$

A-Sig-Id-Anchor-Simplified

$$\frac{\Gamma \overset{\text{anch}}{\vdash} \mathcal{R} \overset{P}{\hookrightarrow} \Gamma_{\text{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \theta \qquad \alpha \notin \text{dom}(\theta_\Gamma)}{\Gamma \overset{\text{anch}}{\vdash} (\alpha, \mathcal{R}) \overset{P}{\hookrightarrow} \Gamma_{\text{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \theta \uplus (\alpha \mapsto (P, \mathcal{R}))}$$

A-Sig-Id-Ignore-Simplified

$$\frac{\Gamma \overset{\text{anch}}{\vdash} \mathcal{R} \overset{\varnothing}{\hookrightarrow} \Gamma_{\text{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \theta \qquad \alpha \notin \text{dom}(\theta_\Gamma)}{\Gamma \overset{\text{anch}}{\vdash} (\alpha, \mathcal{R}) \overset{\varnothing}{\hookrightarrow} \Gamma_{\text{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \theta \uplus (\alpha \mapsto (\times, \varnothing))}$$

The signature stored in the anchoring map is used in A-Sig-Id-TrAsrc to ensure (via a subtyping check), that the returned transparent ascription $(= P < \mathsf{S})$ is well-typed. Rules A-Sig-Id-Anchor-Simplified and A-Sig-Id-Ignore-Simplified illustrate the new mechanism of *optional paths*. They both handle anchoring of an identity signature (simplified to remove high-order types) with and without a path. Rule A-Sig-Id-Anchor-Simplified requires the optional path argument to contain an actual path P that will be used as anchoring path. On the opposite, in rule A-Sig-Id-Ignore, when the optional path is be empty, which is the case when entering a module type of a generative functor, the identity type α is anchored to an *invalid* path $\times$. This prevents its sharing with any other module.

4.3.2 The special role of paths in the source system.

Path typing – $\Gamma_{src} \overset{src}{\vdash} P : \mathbf{C}$. Paths play a special role in ML module systems, and especially in OCaml. Checking equality between two types or aliasing between two modules boils down to checking that both paths resolve to the same component, hence making paths central to the presentation. The *path-typing* judgment is defined as a restriction of module typing. By handling only paths, it never introduces *new* abstract types and identities: it always returns an identity signature with a concrete signature attached. In all the rules, this relies on the invariant stated above that only strengthened signatures are pushed in the context. For instance, in rule S-Typ-Path-LocalMod, the signatures stored in Γ for $A.X$ is an identity signature, thus featuring a transparent ascription (which can point to $A.X$ itself, if the module had a fresh identity). In rule S-Typ-Path-Proj, the ascription of P is the premise is ignored: the *strengthening* invariant ensures that $\mathbf{C}$, the signature of the submodule X, is an identity signature, pointing to $P.X$ (or to an older module). Finally, in S-Typ-Path-AppFct, a subtyping check is done between the parameter signature S and the argument signature $\mathbf{R}$. Substituting Y with P' in the result corresponds to an *instantiation*.

S-Typ-Path-LocalMod

$$\frac{(A.X : \texttt{module } \mathbf{C}) \in \Gamma}{\Gamma \overset{\text{src}}{\vdash} A.X : \mathbf{C}}$$

S-Typ-Path-Proj

$$\frac{(\texttt{module } X : \mathbf{C}) \in \overline{\mathbf{D}} \qquad \Gamma \overset{\text{src}}{\vdash} P : (= \_ < \texttt{sig } \overline{\mathbf{D}} \text{ end})}{\Gamma \overset{\text{src}}{\vdash} P.X : \mathbf{C}}$$

S-Typ-Path-AppFct

$$\frac{\Gamma \overset{\text{src}}{\vdash} P_0 : (= P_0' < (Y : \mathsf{S}) \to \mathbf{C}) \qquad \Gamma \overset{\text{src}}{\vdash} P : (= P' < \mathbf{R}) \qquad \Gamma \overset{\text{src}}{\vdash} \mathbf{R} < \mathsf{S}}{\Gamma \overset{\text{src}}{\vdash} P_0(P) : (= P_0'(P') < \mathbf{C}[Y \mapsto P'])}$$

Signature typing – $\Gamma_{src} \overset{src}{\vdash} \mathsf{S} : \mathsf{S}'$. In the source presentation, the signature typing judgment is used both as a wellformedness check and to remove the three indirections mentioned above, by inlining module types, type aliases, and transparent ascriptions of transparent ascriptions. Rule S-Typ-Sig-TrAscr in the subtle one. If the path P points to another P', the transparent signature

Source code

```
module M = (struct
  module F(Y:S): sig type t end = ..
  module Proj = struct
    module G (Y:S') = struct
      type t = F(Y).t
    end
    module H (Y:S'') = struct
      type t = F(Y).t
    end
end).Proj
```

Canonical signature

$$\begin{array}{l}
\exists \alpha_t.\mathsf{module}\ M : \mathsf{sig} \\
\quad \mathsf{module}\ G : \forall \overline{\beta}.\mathcal{C}' \longrightarrow \mathsf{sig} \\
\quad\quad \mathsf{type}\ t = \alpha_t(\overline{\beta}) \\
\quad \mathsf{end} \\
\quad \mathsf{module}\ H : \forall \overline{\beta}.\mathcal{C}'' \longrightarrow \mathsf{sig} \\
\quad\quad \mathsf{type}\ t = \alpha_t(\overline{\beta}) \\
\quad \mathsf{end} \\
\mathsf{end}
\end{array}$$

Possible source signature

$$\begin{array}{l}
\mathsf{module}\ M : \mathsf{sig}_A \\
\quad \mathsf{module}\ G : (Y : \mathsf{S}') \longrightarrow \mathsf{sig}_B \\
\quad\quad \mathsf{type}\ t = B.t \\
\quad \mathsf{end} \\
\quad \mathsf{module}\ H : (Y : \mathsf{S}'') \longrightarrow \mathsf{sig}_C \\
\quad\quad \mathsf{type}\ t = A.G(Y).t \\
\quad \mathsf{end} \\
\mathsf{end}
\end{array}$$

Fig. 9. An example of the issue with the anchoring higher-order abstract types. In the canonical signature, identities are hidden for simplicity. In the source signature, the artifacts of self-references are in gray.

$(= P < \mathsf{S})$ must be rewritten to refer to P'; this is achieved by the subtyping check, which rewrites the signature $\mathbf{C}$ as an identity signature $\mathbf{C}'$ pointing to P' (and with a concrete signature attached).

$$\frac{\Gamma \overset{\text{src}}{\vdash} P : \mathbf{C} \qquad \Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \mathsf{S}' \qquad \mathsf{S}'/P \gg \mathbf{C}' \qquad \Gamma \overset{\text{src}}{\vdash} \mathbf{C} < \mathbf{C}'}{\Gamma \overset{\text{src}}{\vdash} (= P < \mathsf{S}) : \mathbf{C}'} \quad \text{(S-Typ-Sig-TrAscr)}$$

Therefore, the signatures inferred by signature typing form a subset of the whole syntactical category S, without module types and types aliases, and where transparent ascriptions have been rewritten to identity signatures (by the rule above).

Subtyping. The subtyping judgment is standard, extended with specific rules do deal with transparent ascription. We show here only the key rules.

$$\text{S-Sub-Sig-TrAscrAbs} \quad \Gamma \overset{\text{src}}{\vdash} (= P < \mathbf{R}) < \mathbf{R}$$

$$\text{S-Sub-Sig-TrAscr} \quad \frac{\Gamma \overset{\text{src}}{\vdash} P < P' \qquad \Gamma \overset{\text{src}}{\vdash} \mathbf{R} < \mathbf{R}'}{\Gamma \overset{\text{src}}{\vdash} (= P < \mathbf{R}) < (= P' < \mathbf{R}')}$$

$$\text{S-Sub-Sig-Struct} \quad \frac{\overline{\mathsf{D}_0} \subseteq \overline{\mathsf{D}} \qquad \Gamma, \overline{A.\mathsf{D}} \overset{\text{src}}{\vdash}_A \overline{\mathsf{D}_0} < \overline{\mathsf{D}'}}{\Gamma \overset{\text{src}}{\vdash} \mathsf{sig}_A\ \overline{\mathsf{D}}\ \mathsf{end} < \mathsf{sig}_A\ \overline{\mathsf{D}'}\ \mathsf{end}}$$

Just as concrete type fields are subtypes of abstract ones, Rule S-Sub-Sig-TrAscrAbs may remove transparent ascriptions by subtyping. In rule S-Sub-Sig-TrAscr. the transparent ascription is kept, but with a different associated signature. Those rules are defined only for identity signatures, as subtyping is checked only between signatures inferred by signature typing. Given our self-referring representation for abstract types, the rule S-Sub-Sig-Struct for structural signatures only needs to push the left-hand side set of fields $\overline{\mathsf{D}}$ prefixed by A before comparing declaration by declaration.

4.4 High-order abstract types and identities

So far, we have not considered abstract types and module identities inside applicative functors. The main consequence is that paths stored in the anchoring map did not contain functor applications. However, applications of higher-order abstract types are anchored as paths with functor applications, which poses further challenges.

Let us consider the example of Figure 9. When typing the applicative functor F, an higher-order abstract type α_t is introduced. A mention of this type in the canonical signature can only appear *via* the typing of a path with an application of this functor: here $F(Y).t$ in the body of G and H. But, as always for signature avoidance, the key issue is that the definition point (the functor F) can be lost by projection, while mentions of the abstract type α_t remain. Even if the first mention is a suitable anchoring point, as it is the case inside the functor G, the signature restriction S' on the parameter of G might be stricter than the original one in F, making some applications *ill-typed* when using G instead of F. Specifically, inside the functor H, the path $G(Y).t$ is either valid or not, depending of the subtyping relationship between S' and S''.

Higher-order abstract types, when presented as fields inside applicative functors, are thus restricted to a certain *domain* that depends on the parameter signature (and the number and order of arguments). Even though a qualified type with an application, such as $F(Y).t$, *corresponds* to a higher-order type application $\alpha_t(\overline{\beta})$, the former is more restricted than the latter, as it requires a subtyping relationship between the signature of Y and the parameter of F. Again, the path-based approach gives a special role to the first occurrence. Here, for a type field that mentions a higher-order abstract type, its domain must cover all uses of that type. By contrast, canonical signatures can express sharing of a higher-order abstract type between functors with any domains.

3rd Anchoring Principle. Source signatures can only express sharing of higher-order abstract types and identities when all uses are within the domain of the anchoring point.

Dependencies in the general case. While sound, we argue that this anchoring principle is too permissive. In the general case, the problem of checking if an arbitrary combination of applications allows obtaining a given type expression corresponds to a higher-order unification problem, which is undecidable. Moreover, some heuristics could invent convoluted paths with *new* functor applications, never mentioned by the user, just for referring to abstract types that have lost their original path. This could be quite surprising, if not misleading. We thus extend the anchoring principle to restrict the positions deemed suitable for anchoring. We consider only positions that *resembles* the original definition point: a type field type $t = \varphi(\overline{\alpha})$ is suitable for anchoring only inside an applicative functor that universally quantifies over *exactly* $\overline{\alpha}$.

4th Anchoring Principle. For decidability and usability, we restrict anchoring of higher-order abstract types (and identities) to functors that abstracts over the same number of arguments and in the same order as their (possibly lost) definition point.

In the example of Figure 9, the functor G universally quantifies over the same set of existentials as the lost definition point, which can be seen in the kind of α_t, making the anchoring point valid.

4.4.1 *Consequence for the anchoring judgment.*

The adaptions for anchoring higher-order types are two-fold. First, the anchoring map is extended to store paths parameterized functor arguments. The signature of the functor parameter is also stored to verify subtyping when reconstructing applications. Second, the list of arguments of the higher-order type at the anchoring point (of the form type $t = \varphi(\overline{\alpha})$), is also stored in the anchoring map. Generalizing to any number of arguments, the anchoring $\theta(\varphi)$ is of the form either $\lambda\overline{\beta}.\lambda(Y : \mathcal{C}).(P.t, \overline{\alpha})$ when φ is an abstract type or $\lambda\overline{\beta}.\lambda(Y : \mathcal{C}).((P, \mathcal{R}), \overline{\alpha})$ when φ is an identity type. This can be seen in two key rules:

$$\frac{\Gamma \hookrightarrow \Gamma_{\mathsf{src}} : \theta_\Gamma \qquad \varphi \notin \mathsf{dom}(\theta_\Gamma) \qquad \Gamma = \Gamma_0, \varphi, \Gamma_1 \qquad \mathsf{args}(\Gamma_1) = \overline{\alpha}}{\Gamma \overset{\mathsf{anch}}{\vdash} \mathsf{type}\ t = \varphi(\overline{\alpha}) \xhookrightarrow{A} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\mathsf{src}}{\vdash} \mathsf{type}\ t = A.t : (\varphi \mapsto (A.t, \overline{\alpha}))} \quad \text{(A-Decl-Anchor)}$$

$$\frac{\begin{array}{c}\Gamma, \overline{\alpha} \overset{\mathsf{anch}}{\vdash} \mathcal{C}_a \xhookrightarrow{Y} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\mathsf{src}}{\vdash} \mathsf{S}_a : \theta_a \qquad \mathsf{dom}(\theta_a) = \overline{\alpha} \\ \Gamma, \overline{\alpha}, (Y : \mathcal{C}_a) \overset{\mathsf{anch}}{\vdash} \mathcal{C} \xhookrightarrow{P(Y)?} (\Gamma_{\mathsf{src}}, Y : \mathsf{S}_a); \theta_\Gamma \uplus Y.\theta_a \overset{\mathsf{src}}{\vdash} \mathsf{S} : \theta\end{array}}{\Gamma \overset{\mathsf{anch}}{\vdash} \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \xhookrightarrow{P?} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\mathsf{src}}{\vdash} (Y : \mathsf{S}_a) \to \mathsf{S} : (\beta \mapsto \lambda\overline{\alpha}.\lambda(Y : \mathcal{C}_a).\theta(\beta))} \quad \text{(A-Sig-FctApp)}$$

In Rule A-Decl-Anchor, a higher-order abstract type is associated with the name t, and the list of type variable arguments $\overline{\alpha}$ is stored. The operator $\mathsf{args}(\Gamma_1)$ is defined to gather all the type variables universally quantified in Γ_1 (distinguished from existentially quantified ones, as they appear in front of functor parameters), between the point where φ was introduced an the current point. In Rule A-Sig-FctApp, the anchoring map θ of the body signature $\mathcal{C}$ is abstracted over the universally quantified type variables $\overline{\alpha}$ and the parameter $(Y : \mathcal{C}_a)$ in the result map.

Type anchoring. The last step is to reconstruct a path (or a pair with a path and a signature) from a canonical type, using the computed maps. The judgment is better understood by its properties:

$$\begin{array}{rcl} \Gamma \overset{\text{anch}}{\vdash} \tau \hookrightarrow \Gamma_{\text{src}}; \theta_\Gamma \vdash (P.t, \_) & \Longrightarrow & \Gamma \overset{\text{can}}{\vdash} P.t : \tau \\ \Gamma \overset{\text{anch}}{\vdash} \tau \hookrightarrow \Gamma_{\text{src}}; \theta_\Gamma \vdash ((P, \mathcal{R}), \_) & \Longrightarrow & \Gamma \overset{\text{can}}{\vdash} P : (\tau, \mathcal{R}) \end{array}$$

For first-order types, the rule A-Type-Star retrieves the anchoring path from the environment. For a higher-order type φ applied to a list of arguments $\overline{\tau}$, the rule A-Type-TypeApp is a bit more involved. First, the list of arguments is split into an ignored prefix, then a list of identity types ρ_i followed by abstract types $\overline{\tau}_i$. The length of this list (of lists) correspond to the number of functor parameters. Each identity type is anchored to some path P_i and signature $\mathcal{R}_i$. Then, the anchoring map is applied to the list of types, paths, and signatures. This application checks the subtyping between the signatures and the stored parameter signatures, and returns a result z (either a type path or a pair of a path and an identity), and, crucially, the list of arguments of the anchored point. This list must be equal to the list of arguments $\overline{\tau}$: it means that the anchoring point can indeed produce the current use point.

$$\textsc{A-Type-Star}\quad \frac{\theta_\Gamma(\alpha) = (z, \varnothing)}{\Gamma \overset{\text{anch}}{\vdash} \alpha \hookrightarrow \Gamma_{\text{src}}; \theta_\Gamma \vdash z}$$

$$\textsc{A-Type-TypeApplication}\quad \frac{\begin{array}{c}\overline{\tau} = \_ :: \overline{\rho_i, \overline{\tau}_i} \qquad \Gamma \overset{\text{anch}}{\vdash} \overline{\rho_i \hookrightarrow \Gamma_{\text{src}}; \theta_\Gamma \vdash (P_i, \mathcal{R}_i)} \\ \Gamma \overset{\text{anch}}{\vdash} \theta_\Gamma(\tau)\overline{(\rho_i, \overline{\tau}_i)(P_i, (\rho_i, \mathcal{R}_i))} = (z, \overline{\tau})\end{array}}{\Gamma \overset{\text{anch}}{\vdash} \varphi\,(\overline{\tau}) \hookrightarrow \Gamma_{\text{src}}; \theta_\Gamma \vdash z}$$

4.4.2 A best-effort *source system.* As explained above, finding a principal signature in the source syntax in presence of applicative functors would require inventing arbitrary complex paths. Instead, our presentation takes a simpler approach. We first introduce a subset of subtyping called *abstraction subtyping*, written $\Gamma_{\text{src}} \overset{\text{src}}{\vdash} \mathsf{S} \sqsubset \mathsf{S}'$, that reuses all the same rules as subtyping, except for the structural signatures one. It is changed to compare the same set of declarations, whereas S-SubGen-Sig-Struct allowed to consider a subset of declarations: no reordering nor deletion of fields. This allows us to define how the source module typing judgment handles the signature avoidance problem. The judgment is written $\Gamma_{\text{src}} \overset{\text{src}\,\diamond}{\vdash} \mathsf{M} : \mathsf{S}$, and depends on an explicit typing mode mechanism to mimic the mode that could be read of the signatures in the canonical system. The key rule is Rule S-Typ-Mod-Proj for a projection. During a projection, a submodule X is extracted out of the module M. However, its signature might have dependencies with other components in S. Here, we allow an abstraction subtyping that can remove any dependencies, by rewriting any type expressions and remove transparent ascriptions. Finally, the signature of the submodule S′ is checked to be wellformed, but crucially, in an environment that *does not contain* the other declarations $\overline{\mathsf{D}}$: the signature must have no dependencies left. This rule is not algorithmic, as the more abstract signature appears out of the blue. It also allows for loosing precision by *over-abstraction*, i.e., turning type fields or identities that could have been kept concrete into abstract types, which might return a non-principal signature. The rest of the rule of module and binding typing are straightforward and given in §C.4.

$$\textsc{S-SubAbs-Sig-Struct}\quad \frac{\Gamma, \overline{A.\mathsf{D}} \overset{\text{src}}{\vdash}_A \overline{\mathsf{D}} \sqsubset \overline{\mathsf{D}}'}{\Gamma \overset{\text{src}}{\vdash} \mathtt{sig}_A\ \overline{\mathsf{D}}\ \mathtt{end} \sqsubset \mathtt{sig}_A\ \overline{\mathsf{D}}'\ \mathtt{end}}$$

$$\textsc{S-Typ-Mod-Proj}\quad \frac{\begin{array}{cc}\Gamma_{\text{src}} \overset{\text{src}\,\diamond}{\vdash} \mathsf{M} : \mathsf{S} & \Gamma_{\text{src}} \overset{\text{src}}{\vdash} \mathsf{S} \sqsubset \mathtt{sig}_A\ \overline{\mathsf{D}}\ \mathtt{end} \\ \mathtt{module}\ X : \mathsf{S}' \in \overline{\mathsf{D}} & \Gamma_{\text{src}} \overset{\text{src}}{\vdash} \mathsf{S}' : \mathsf{S}'\end{array}}{\Gamma_{\text{src}} \overset{\text{src}\,\diamond}{\vdash} \mathsf{M}.X : \mathsf{S}'}$$

The current implementation of OCaml allows an abstraction subtyping to prevent signature avoidance, without providing any guarantees regarding the principality of the inferred signature– and the implementation often does unnecessary and surprising over-abstractions.

4.5 Linking the source and canonical systems

The anchoring judgment allows us to link canonical and source typing. If all signatures used in a canonical typing derivation were anchorable, then the typing derivation can be translated into the source system. More formally, we define the *anchorable canonical typing* judgment $\Gamma \overset{\text{anch}}{\vdash} \mathtt{M} : \exists^{\diamond}\overline{\alpha}.\mathcal{C}$ by only changing the projection rule to add an anchorability condition:

A-Typ-Mod-Proj

$$\frac{\Gamma \overset{\text{anch}}{\vdash} \mathtt{M} : \exists^{\diamond}\overline{\alpha}.\,(\_, \mathtt{sig}\ \overline{\mathcal{D}}\ \mathtt{end}) \qquad \mathtt{module}\ X : \mathcal{C} \in \overline{\mathcal{D}} \qquad \Gamma, \overline{\alpha} \overset{\text{anch}}{\vdash} \mathcal{C} \overset{\varnothing}{\hookrightarrow} \Gamma_{\text{src}}; \theta_{\Gamma} \overset{\text{src}}{\vdash} \mathsf{S} : \theta}{\Gamma \overset{\text{anch}}{\vdash} \mathtt{M}.X : \exists^{\diamond}\overline{\alpha}.\mathcal{C}}$$

Theorem 4.1 (Anchoring of typing). *Anchorable canonical typing derivations can be translated to source typing derivations.*

$$\Gamma \overset{anch}{\vdash} \mathtt{M} : \exists^{\diamond}\overline{\alpha}.\mathcal{C} \implies \exists \Gamma_{src}, \theta_{\Gamma}, \mathsf{S}, \theta, \quad \Gamma, \overline{\alpha} \overset{anch}{\vdash} \mathcal{C} \overset{\varnothing}{\hookrightarrow} \Gamma_{src}; \theta_{\Gamma} \overset{src}{\vdash} \mathsf{S} : \theta \quad \wedge \quad \Gamma_{src} \overset{src\,\diamond}{\vdash} \mathtt{M} : \mathsf{S}$$

The reader might wonder if the other way around is also true, i.e., if source typing derivations can be translated into canonical ones, with inferred signatures linked by the anchoring judgment. The answer is no, as the permissive *abstraction* subtyping of the source system allows it to remove type equalities and transparent ascriptions that would make the anchoring fail. Future work is required to characterize precisely these edge-cases.

5 THE FOUNDATIONS: F$^{\omega}$ ELABORATION

The canonical system is designed to offer a standalone, type-standard and expressive approach to the typing of OCaml modules, while hiding the complexity and artifacts of the encoding in F$^{\omega}$. Still, the encoding served as a basis for the design of the canonical system and is now used as a proof of type soundness. The F$^{\omega}$ encoding of module expressions and signatures shines a new light on the mechanisms of the canonical system. Our encoding is largely based on the work of Rossberg et al. [15], but differs in a key manner in the treatment of skolemization used to encode abstract types of applicative functors. One of our important contribution is the introduction of *transparent* existential types, an intermediate between the standard *opaque* existential types and the absence of abstraction, allowing us to bring the treatment of applicative and generative functors closer and significantly simplifying the elaboration as well as the resulting programs.

5.1 F$^{\omega}$ with kind polymorphism

We use a standard variant of explicitly typed F$^{\omega}$ with primitive records, existential types, and kind polymorphism. Its syntax is given in Figure 10. Its typing rules are standards and available in §D. Type equivalence defined by $\beta\eta$-conversion and reordering of record fields is standard. We use letters τ and e to range over types and expressions to distinguish them from the core language types and expressions, even though these should actually be seen as a subset of τ and e with some syntactic sugar. The syntax is presented with explicit kinds. We write $\varkappa$ for kind variables, α and β for type variables of any kind, and φ and ψ for type variables of higher kinds. We write kinds κ (and kind abstraction $\Lambda\varkappa.$) in pale color so that they are nonintrusive, and we often leave them implicit.

For the sake of conciseness and readability, we introduce some syntactic sugar. The first set of syntax extensions is described on Figure 11—but more will be added later on. As a convention, we use a wildcard when a type annotation unambiguously determined by an immediate subexpression is 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 allow packing and unpacking of a sequence of types altogether, as defined on Figure 11. We also introduce a combined syntactic form repack$^{\blacktriangledown}$ $\langle \bar{\alpha}, x \rangle = e_1$ in e_2, which allows the abstract types of e_1

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

Fig. 10. Syntax of F^ω

$$\begin{array}{lcl}
\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 & \triangleq & (\lambda(x : \_).e_2)\, e_1 \\
\mathsf{pack}\ \langle \tau\bar{\tau}, e \rangle\ \mathsf{as}\ \exists^{\blacktriangledown}(\alpha\bar{\alpha} : \kappa).\sigma & \triangleq & \mathsf{pack}\ \langle \tau, e_0 \rangle\ \mathsf{as}\ \exists^{\blacktriangledown}(\alpha\bar{\alpha} : \kappa).\sigma \\
 & & \text{where } e_0 \text{ is} \\
 & & \quad \mathsf{pack}\ \langle \bar{\tau}, e \rangle\ \mathsf{as}\ \exists^{\blacktriangledown}(\bar{\alpha} : \kappa).\sigma \\
\mathsf{pack}\ \langle \emptyset, e \rangle\ \mathsf{as}\ \sigma & \triangleq & e \\
\mathsf{unpack}\ \langle \alpha\bar{\alpha}, x \rangle = e_1\ \mathsf{in}\ e_2 & \triangleq & \mathsf{unpack}\ \langle \alpha, x \rangle = e_1\ \mathsf{in} \\
 & & \mathsf{unpack}\ \langle \bar{\alpha}, x \rangle = x\ \mathsf{in}\ e_2 \\
\mathsf{unpack}\ \langle \emptyset, x \rangle = e_1\ \mathsf{in}\ e_2 & \triangleq & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
\mathsf{repack}^{\blacktriangledown}\ \langle \bar{\alpha}, x \rangle = e_1\ \mathsf{in}\ e_2 & \triangleq & \mathsf{unpack}\ \langle \bar{\alpha}, x \rangle = e_1\ \mathsf{in} \\
 & & \mathsf{pack}\ \langle \bar{\alpha}, e_2 \rangle\ \mathsf{as}\ \exists^{\blacktriangledown}(\bar{\alpha} : \kappa).\_
\end{array}$$

Fig. 11. Syntactic sugar for F^ω terms

Structural signature

$$\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \quad \triangleq \quad \{\overline{\mathcal{D}}\}$$

Identity signature

$$(\tau, \mathcal{R}) \quad \triangleq \quad \{\mathsf{id} : \tau, \mathsf{Val} : \mathcal{R}\}$$

Declarations

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

Fig. 12. Syntactic sugar for canonical signatures. Functors and abstract signatures correspond directly to F^ω types.

to appear in the type of e_2. Since the type of repacking is here fully determined by the combination of $\bar{\alpha}$ and the type of e_2, we leave it implicit.

5.2 Encoding of signatures

Canonical signatures are actually F^ω types with some syntactic sugar, which is shown in Figure 12. To help with the elaboration, we assume a collection ℓ_I of record labels indexed by identifiers of the canonical (and source) systems. A small trick is needed to represent type fields, which have no computational content, but cannot be erased during elaboration as they carry additional typing constraints. We reuse the solution of *F-ing*, encoding them as identity functions with type annotations. For this, we introduce the following syntactic sugar for the term representing a type field (on the left). We overload the notation to also mean its type (on the right).

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

The type τ is used as argument of a higher-kinded type operator φ to uniformly handle the encoding of types of any kind. The key (and only useful) property is that two types (of the same kind) are equal if and only if their encodings are equal.

5.3 Sharing existential types by repacking

While the correspondence between canonical signatures and F^ω types is straightforward, the actual encoding of module expressions as F^ω terms is slightly more involved. Although structures and functions 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 §3.2.1.

Let us first consider the easier generative case. The only construct for handling a term with an abstract type is the *unpack* operator, 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*. This observation was at the core of the design of *open existential types* [11] and of *recursive type generativity* [2]. Here, in order to stay in plain F^ω, we adapt the trick of *F-ing*: abstract types are explicitly lifted out of the record, component by component, via a *rebinding* pattern where abstract types are unpacked, shared with the rest of the structure, and then repacked.

To capture this lifting of existential out of records, we define two new constructs, defined as:

$$\begin{aligned}
\mathsf{lift}^{\blacktriangledown}\ \{\ell_1 = e_1, \ell_2 = e_2\} &\triangleq \mathsf{let}\ x = \{\ell_1 = e_1; \ell_2 = e_2\}\ \mathsf{in} \\
&\quad\ \mathsf{repack}^{\blacktriangledown}\ \langle \alpha, x_1\rangle = x.\ell_1\ \mathsf{in}\ \mathsf{repack}^{\blacktriangledown}\ \langle \beta, x_2\rangle = x.\ell_2\ \mathsf{in}\ \{\ell_1 = x.\ell_1, \ell_2 = x.\ell_2\} \\
\mathsf{lift}^{\blacktriangledown}\ \langle \bar\alpha, x = e_1 \mathbin{@} e_2\rangle &\triangleq \mathsf{repack}^{\blacktriangledown}\ \langle \bar\alpha, x\rangle = e_1\ \mathsf{in}\ \mathsf{repack}\left\langle \bar\beta, x_2\right\rangle = e_2\ \mathsf{in}\ x_1 \mathbin{@} x_2
\end{aligned}$$

First, we introduce $\mathsf{lift}^{\blacktriangledown}\ \{\ell_1 = e_1, \ell_2 = e_2\}$ for lifting out of a 2-field independent[4] record, which will be used to manipulate identity fields. Second, we define a more general $\mathsf{lift}^{\blacktriangledown}\ \langle \bar\alpha, x = e_1 \mathbin{@} e_2\rangle$ for lifting out the concatenation of two "*dependent*" records, which is more involved, as e_2 may refer to the record e_1 via x and its abstract types $\bar\alpha$, hence $\bar\alpha$ and x act as binders whose scope is e_2. These are better understood by their derived typing rules:

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

5.4 Transparent existential types and their lifting through function types

The repacking pattern allows lifting existential types outside of product types. Unfortunately this is insufficient for the applicative case, which uses skolemization to 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 we have done in the canonical system, is not definable in F^ω. One solution is to avoid skolemization by a-priori abstraction over all possible type variables, i.e., the whole typing context. This is the solution followed the authors of *F-ing* and Shan [17]. While this suffices to prove soundness, this encoding is impractical for manual use of the pattern and does not provide a good intuition of what modules really are. The encoding could be slightly improved by abstracting over fewer variables, but this would not solve the problem, which is a-priori abstraction.

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

We create a transparent existential type with the expression $\mathsf{pack}\ \ e\ \ \mathsf{as}\ \exists^{\triangledown\tau}(\alpha : \kappa).\sigma$, which behaves much as $\mathsf{pack}\ \langle \tau, e\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}(\alpha : \kappa).\sigma$, except that the witness type τ remains visible in the result type. A transparent existential type is thus weaker than a usual abstract type, as we still see the witness type. In particular, 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 $\mathsf{seal}\ e$, which behaves as the identity but turns the expression e of type $\exists^{\triangledown\tau}(\alpha : \kappa).\sigma$ into one of type $\exists^{\blacktriangledown}(\alpha : \kappa).\sigma$. Transparent existential types may also be used abstractly, with the expression $\mathsf{repack}^{\triangledown}\ \langle \alpha, x\rangle = e_1\ \mathsf{in}\ e_2$, which is the pending of the expression $\mathsf{repack}^{\blacktriangledown}\ \langle \alpha, x\rangle = e_1\ \mathsf{in}\ e_2$ but when e_1 is a transparent existential

[4]The use of a let-binding prior to repacking ensures that e_1 and e_2 are independent.

type $\exists^{\triangledown\tau}(\alpha : \kappa).\sigma_1$. We distinguish the two forms using the superscripts $^\blacktriangledown$ and $^\triangledown$ for opaque and transparent existentials. In both cases, e_2 is typed in a context extended with the abstract type $\alpha : \kappa$ and x of type σ_1. That is, e_2 cannot see the witness type τ. However, the abstract type variable α may still appear in the type σ_2 of the expression e_2, and therefore it is made transparent again in the result type of $\mathsf{repack}^\triangledown\ \langle\alpha, x\rangle = e_1$ in e_2, which is $\exists^{\triangledown\tau}(\alpha : \kappa).\sigma_2$. We do not need a transparent version of unpack $\langle\alpha, x\rangle = e_1$ in e_2, since it would be equivalent to unpack $\langle\alpha, x\rangle =$ seal e_1 in e_2.

So far, one may wonder what is the advantage of transparent existentials by comparison with opaque existentials. We provide two key additional constructs for lifting transparent existentials across arrows types and universal types—the only reason to have introduced them in the first place. The lifting across an arrow type, written $\mathsf{lift}^\rightarrow\ e$, turns an expression of type $\sigma_1 \longrightarrow \exists^{\triangledown\tau}(\alpha : \kappa).\sigma_2$ into one of type $\exists^{\triangledown\tau}(\alpha : \kappa).(\sigma_1 \longrightarrow \sigma_2)$ as long as α is fresh for σ_1. While this operation seems easy, it crucially depends on existential types begin transparent—this transformation would be unsound with opaque existentials. Indeed, since we can observe the witness τ, we can ensure that it can be expressed independently of σ_1, allowing us to lift it outside of the function. Similarly, lifting across a universal type variable β of kind κ', written $\mathsf{lift}^\forall\ e$, turns an expression of type $\Lambda(\beta : \kappa').\exists^{\triangledown\tau}(\alpha : \kappa).\sigma$ into one of type $\exists^{\triangledown\lambda(\beta:\kappa').\tau}(\alpha' : \kappa' \longrightarrow \kappa).\forall(\beta : \kappa').\sigma[\alpha \mapsto \alpha'\ \beta]$, provided β is fresh for τ, using skolemization of both the existential variable α and its witness type τ.

To summarize, we have extended the syntax of F^ω as follows:

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

Their typing rules are given in §D. These constructs have no additional computational content, namely $\mathsf{repack}^\triangledown\ \langle\alpha, x\rangle = \tau$ in σ behaves as a let-binding, while the other constructs behave as e.

5.5 Implementation of transparent existential types in F^ω

Interestingly, transparent existential types are completely definable in F^ω, as shown in §D.1. It defines an expression $e_{\mathbb{E}}$ as pack $\langle\tau_0, e_0\rangle$ as $\tau_{\mathbb{E}}$ where e_0 is the concrete implementation, and τ_0 is the interface type that hides the implmentation of the type $\mathbb{E}$. Using this definition, we may see a program e with transparent existential types as a program unpack $\langle\mathbb{E}, x_{\mathbb{E}}\rangle = e_{\mathbb{E}}$ in e in plain F^ω, with the following additional syntactic sugar[5]:

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

We define the lifting operations for records fields $\mathsf{lift}^\triangledown\ \{\ell_1 = e_1, \ell_2 = e_2\}$ and dependent record concatenation $\mathsf{lift}^\triangledown\ \langle\bar\alpha, x_1 = e_1 @ e_2\rangle$ exactly as their opaque versions, but replacing opaque repacking by transparent repacking. We also define a new operation $\mathsf{lift}^*\ e$ that uses a combination of the primitive $\mathsf{lift}^\rightarrow$ and $\mathsf{lift}^\forall$ to turn an expression e of type $\forall\bar\alpha.\sigma_1 \longrightarrow \exists^{\triangledown\bar\tau}(\bar\beta).\sigma_2$ into one of type $\exists^{\triangledown\overline{\lambda\bar\alpha.\tau}}(\bar\beta').\forall\alpha.\sigma_1 \longrightarrow \sigma_2\left[\bar\beta \mapsto \overline{\beta'\ \bar\alpha}\right]$, which is the key transformation for lifting existentials out of applicative functor bodies. This operator is defined from $\mathsf{lift}^{\forall p \rightarrow}_q\ e$ where p and q represent the size of $\bar\alpha$ and $\bar\beta$[6], which is it itself inductively defined as follows:

$$\begin{array}{c} \mathsf{lift}^{\forall p\rightarrow}_{q+1}\ e \triangleq \mathsf{repack}^\triangledown\ \langle\alpha, x\rangle = \mathsf{lift}^{\forall p\rightarrow}\ e\ \mathsf{in}\ \mathsf{lift}^{\forall p\rightarrow}_q\ x \qquad \mathsf{lift}^{\forall^{p+1}}\ e \triangleq \mathsf{lift}^\forall\ (\Lambda\alpha.\mathsf{lift}^{\forall^p}\ (e\ \alpha)) \\ \mathsf{lift}^{\forall p\rightarrow}\ e \triangleq \mathsf{lift}^{\forall p}\ (\mathsf{lift}^\rightarrow\ e) \qquad \mathsf{lift}^{\forall p\rightarrow}_0\ e \triangleq e \qquad \mathsf{lift}^{\forall^0}\ e \triangleq e \qquad \mathsf{lift}^*\ e \triangleq \mathsf{lift}^{\forall p\rightarrow}_q\ e \end{array}$$

[5]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.
[6]p and q are left implicit as they can be determined from the type of the argument e

5.6 Elaboration judgments

As for the canonical system, the elaboration relies on a typing judgment for signatures and modules, and a subtyping judgment. However, as canonical signatures are already F^ω types, we can reuse the canonical typing judgment (easily modified to use implicit kinds). Specifically, canonical signatures do not mention transparent existential types, and neither do typing contexts. 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). We have the two following judgments:

Subtyping $\Gamma \overset{\text{elab}}{\vdash} \mathcal{C} \prec \mathcal{C}' \rightsquigarrow f$ extends canonical subtyping to return an explicit coercion function f. The judgment is also defined for declarations $\Gamma \overset{\text{elab}}{\vdash} \mathcal{D} \prec \mathcal{D}' \rightsquigarrow f$. Interestingly, as signatures do not contain transparent existential types, subtyping between signatures is (a subcase of) standard subtyping in F^ω. As they are similar to canonical subtyping, we left the rules in §E.1. The judgments have the following property regarding F^ω typing:

$$\Gamma \overset{\text{elab}}{\vdash} \mathcal{C} \prec \mathcal{C}' \rightsquigarrow f \implies \Gamma \overset{F\omega}{\vdash} f : \mathcal{C} \to \mathcal{C}' \qquad \Gamma \overset{\text{elab}}{\vdash} \mathcal{D} \prec \mathcal{D}' \rightsquigarrow f \implies \Gamma \overset{F\omega}{\vdash} f : \{\mathcal{D}\} \to \{\mathcal{D}'\}$$

To factor notations for the typing judgment, we introduce the meta-variable ϑ that stands for either the generative mode $\blacktriangledown$ or the applicative mode $\triangledown\tau$ together with a witness type τ. We write $\mathrm{mode}(\vartheta)$ (*resp.* $\mathrm{mode}(\bar{\vartheta})$) for the mode of ϑ (*resp.* the homogeneous sequence $\bar{\vartheta}$), which is either $\triangledown$ or $\blacktriangledown$. When a mode is expected without a witness type, we may leave the projection implicit and just write ϑ instead of $\mathrm{mode}(\vartheta)$.

Typing $\Gamma \overset{\text{elab}}{\vdash} \mathsf{M} : \exists^{\bar{\vartheta}} \bar{\alpha}.\mathcal{C} \rightsquigarrow e$ extends canonical typing with the elaborated module term e. The judgment is also defined for bindings $\Gamma \overset{\text{elab}}{\vdash}_A \mathsf{B} : \exists^{\bar{\vartheta}} \bar{\alpha}.\mathcal{D} \rightsquigarrow e$. The properties of the two judgments are detailed below.

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

$$\begin{array}{llll} \Gamma \overset{elab}{\vdash} \mathsf{M} : \exists^{\bar{\vartheta}} \bar{\alpha}.\mathcal{C} \rightsquigarrow e & \implies \Gamma \overset{F\omega}{\vdash} e : \exists^{\bar{\vartheta}} \bar{\alpha}.\mathcal{C} & \wedge\ \Gamma \overset{can}{\vdash} \mathsf{M} : \exists^{\bar{\vartheta}} \bar{\alpha}.\mathcal{C} \\ \Gamma \overset{elab}{\vdash} \overline{\mathsf{B}} : \exists^{\bar{\vartheta}} \bar{\alpha}.\overline{\mathcal{D}} \rightsquigarrow e & \implies \Gamma \overset{F\omega}{\vdash} e : \exists^{\bar{\vartheta}} \bar{\alpha}.\ \{\overline{\mathcal{D}}\} & \wedge\ \Gamma \overset{can}{\vdash} \mathsf{M} : \exists^{\bar{\vartheta}} \bar{\alpha}.\overline{\mathsf{D}} \end{array} \tag{1}$$

Theorem 5.2 (Completeness). *Well-typed canonical terms can always be elaborated, to either opaque or transparent existentials (and similarly for bindings):*

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

Proof Sketch. Soundness is obtained via induction on the typing rules. Completeness can be easily established as the elaboration rules mimic the canonical typing rules with no additional constraints on the premises, except for transparent existentials. However, these only appear on the type of elaborated modules as a positive information, which is never restrictive. In particular, when a transparent existential type is used, it is always abstractly and its abstract type variable is pushed in the context after dropping the witness type, exactly as an opaque existential type, thus exactly as the canonical system does. □

5.7 Elaborated typing rules

The full set of typing rules for expressions is given in appendix §E.2. Below, we only present an except of the most significant rules. Contrary to canonical typing, elaborated typing does not feature a typing mode, as it can be read directly in the type.

Elaboration of structures. The key rule for structures is the sequence rule that combines bindings. It concisely writes as follows for generative and applicative modes:

E-TypGen-Seq

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

E-TypApp-Seq

$$\frac{\Gamma \overset{\text{elab}}{\vdash}_A \mathsf{B} : \exists^{\triangledown\overline{\tau}_1}(\overline{\alpha}_1).\mathcal{D} \rightsquigarrow e_1 \qquad I_1 = \mathrm{dom}(\mathsf{B}) \qquad \Gamma, \overline{\alpha}_1, A.I_1 : \mathcal{D} \overset{\text{elab}}{\vdash}_A \overline{\mathsf{B}} : \exists^{\triangledown\overline{\tau}_2}(\overline{\alpha}_2).\overline{\mathcal{D}} \rightsquigarrow e_2}{\Gamma \overset{\text{elab}}{\vdash}_A \mathsf{B}; \overline{\mathsf{B}} : \exists^{\triangledown\overline{\tau}_1\overline{\tau}_2}(\overline{\alpha}_1\overline{\alpha}_2).(\mathcal{D}, \overline{\mathcal{D}}) \rightsquigarrow \mathsf{lift}^{\triangledown}\ \langle \overline{\alpha}_1, x_1 = e_1 \mathbin{@} (\mathsf{let}\ A.I_1 = x_1.\ell_{I_1}\ \mathsf{in}\ e_2)\rangle}$$

We concatenates the single field of e_1 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. Notice that the generative and applicative versions of the rules can be factored as follows:

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

Since identity signatures are actually record types of the form $\{\mathsf{id} : \tau, \mathsf{Val} : \mathcal{R}\}$, structures are elaborated to a two-field record containing a type field for the identity $\langle\!\langle \tau \rangle\!\rangle$ and a value of type $\mathcal{R}$. The identity field is there solely for typing purposes and could be erased at runtime. To introduce a fresh identity encoded as a unit type[7], we use a constant $e^{\diamond}_{\mathsf{id}}$ defined as:

$$e^{\triangledown}_{\mathsf{id}} \triangleq \mathsf{pack}\ \langle\!\langle () \rangle\!\rangle\ \mathsf{as}\ \exists^{\triangledown()}(\alpha_0).\langle\!\langle \alpha_0 \rangle\!\rangle \qquad\qquad e^{\blacktriangledown}_{\mathsf{id}} \triangleq \mathsf{pack}\ \langle \alpha_0, \langle\!\langle () \rangle\!\rangle \rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\alpha_0.\langle\!\langle \alpha_0 \rangle\!\rangle$$

Using this constant, we have a unified rule for typing structures in both modes:

$$\frac{\Gamma \overset{\text{elab}}{\vdash}_A \overline{\mathsf{B}} : \exists^{\bar{\vartheta}}\bar{\alpha}.\overline{\mathcal{D}} \rightsquigarrow e \qquad A \notin \Gamma \qquad \diamond = \mathrm{mode}(\bar{\vartheta})}{\Gamma \overset{\text{elab}}{\vdash} \mathtt{struct}_A\ \overline{\mathsf{B}}\ \mathsf{end} : \exists^{\vartheta_0\bar{\vartheta}}\alpha_0\bar{\alpha}.\ (\alpha_0, \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}) \rightsquigarrow \mathsf{lift}^{\diamond}\ \left\{\mathsf{id} = e^{\diamond}_{\mathsf{id}}, \mathsf{Val} = e\right\}} \quad \text{(E-Typ-Mod-Struct)}$$

Modes and sealing. By default, typing is done in applicative mode, hence inferring transparent existentials, but it can be turned into generative mode when required, using Rule E-Typ-Mode. Since it is defined on paths, signature ascription is applicative by default (rule E-Typ-Sig-App).

E-Typ-Mode

$$\frac{\Gamma \overset{\text{elab}}{\vdash} \mathsf{M} : \exists^{\triangledown\overline{\tau}}(\overline{\alpha}).\mathcal{C} \rightsquigarrow e}{\Gamma \overset{\text{elab}}{\vdash} \mathsf{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow \mathsf{seal}\ e}$$

E-Typ-Sig-App

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

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

$$\frac{\Gamma \overset{\text{elab}}{\vdash} \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a \overset{\text{elab}}{\vdash} \mathsf{M} : \exists^{\triangledown\overline{\tau}}(\overline{\beta}).\mathcal{C} \rightsquigarrow e}{\begin{array}{c}\Gamma \overset{\text{elab}}{\vdash} (Y : \mathsf{S}) \to \mathsf{M} : \exists^{\triangledown(), \overline{\lambda\overline{\alpha}.\tau}}(\alpha_0, \overline{\beta'}).\ \left(\alpha_0, \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]\right) \\ \rightsquigarrow \mathsf{lift}^{\triangledown}\ \left\{\mathsf{id} = e^{\triangledown}_{\mathsf{id}}, \mathsf{Val} = \mathsf{lift}^{*}\ (\Lambda\overline{\alpha}.\lambda(Y : \mathcal{C}_a).e)\right\}\end{array}} \quad \text{(E-TypApp-AppFct)}$$

$$\frac{\Gamma \overset{\text{elab}}{\vdash} \mathsf{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow e}{\Gamma \overset{\text{elab}}{\vdash} () \to \mathsf{M} : \exists^{\triangledown()}(\alpha_0).\ (\alpha, () \to \mathcal{C}) \rightsquigarrow \mathsf{lift}^{\triangledown}\ \left\{\mathsf{id} = e^{\triangledown}_{\mathsf{id}}, \mathsf{Val} = \lambda(\_ : ()).e\right\}} \quad \text{(E-TypApp-Mod-GenFct)}$$

The body of an applicative functor is elaborated to transparent existentials which are lifted through λ's, while in the generative case, the existentials are opaque and must not (and cannot) be lifted. However, this difference is largely artificial as a result of using a special argument of type () to enforce generativity. Otherwise, the main difference lies in enforcing the body of the functor to be typed in generative mode, hence to be an opaque existential type. Indeed, since lift^{*} is neutral on

[7]We could introduce a special kind for identities, as they are never used in place of normal types.

terms that do not have transparent existential types, the elaboration of the generative case could also be written $\mathsf{lift}^{\triangledown}\ \{\mathsf{id} = e^{\triangledown}_{\mathsf{id}}, \mathsf{Val} = \mathsf{lift}^{*}\ \lambda(\_ : ()).\,e\}$: the two cases are only differing by the modes of elaboration of their bodies (since the lifting modes just adapts to the type of the argument).

Summary. 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 not 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. Functor applications need not even be aware of existential types and are just a standard application in F^{ω}. This considerably simplifies the treatment of applicative functors.

6 RELATED WORKS

The literature regarding ML-modules is both rich and varied. The link between abstract types in ML-module systems and existential types in F^{ω} was initially explored by Mitchell and Plotkin [10]. This vision was opposed by MacQueen [8] 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. [4] 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. [4]. Two approaches for the formalization and improvement of abstract types in SML were later concomitantly described by Leroy [5] using manifest types and Harper and Lillibridge [3] via an adapted F^{ω} with translucent sums. The genesis of the OCaml module system was specified by Leroy [5, 7] with, later, an extension to applicative functors [6].

The key idea for a simplified link between modules and F^{ω}, developed by Russo [16], was to use existential types to interpret signatures. Following this link, Dreyer [2] proposed to model generativity using stamps instead of existential types, while Montagu and Rémy [12] 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. [15] with the elaboration of a large subset of the SML into F^{ω}, dubbed the *F-ing* approach. *F-ing* gives a *syntactic* translation from SML syntax directly into F^{ω}, thus providing a semantic by elaboration. *F-ing* is safe by construction[8], inheriting the property from F^{ω}, but requires the programmer to think in terms of the elaboration, which is quite involved in some cases, and only sees the elaborated types instead of the usual signatures. This makes direct reasoning on the source program difficult, if at all feasible for the programmer. By contrast, our canonical system gives a specification directly on the source terms, without having to think in terms of encoding, but leveraging the insights provided by the elaboration to F^{ω}. We also give a direct specification using path-based source signatures, as expected by OCaml programmers, and thus provide the currently most complete source-level specification of OCaml modules.

Moving one step further, Rossberg [14] achieved a unification of the core and module languages (thus, unstratified), called 1ML, using F^{ω} as the underlying programming language and seeing module constructs as syntactic sugar. This is appealing, even though the prototype implementation only covered the generative case: the applicative case might have been unusable in practice, due to a priori extrusion of quantifiers over the whole context. Hopefully, this could be fixed by applying our *a posteriori* lifting of transparent existentials types technique to 1ML.

More recently, Crary [1] 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

[8]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.

with *F-ing*. Our work provides complementary information on the understanding of signature avoidance, not on its origin nor how to avoid it, which was already well-understood in *F-ing*, but on the difficulties and the principled way to solve it in the path-based approach of OCaml.

7 CONCLUSION AND FUTURE WORKS

In this article, we have introduced and formalized the *canonical* system, a middle point between the source path-based module system used in OCaml and F^ω. Using this system, we first shone a new light on the ad-hoc techniques of a source-only presentation and provided a detailed description of the solvable and unsolvable cases of signature avoidance. Second, we gave an improved elaboration of modules into F^ω, using the new notion of *transparent existentials* to treat applicative functors in almost the same simple way as generative functors. There remains questions to further explore.

One immediate application of our work is to use canonical signatures as intermediate typing representation for OCaml. For this purpose, we first need to maintain module type names from the source. We have avoided this difficulty by inlining all module types, but a real implementation will need strategies to keep them, which should preferably also be taken into account in the formalization. We also need to support more features of the OCaml module system. First-class modules, *with*-constraints, private types, etc. should not raise any difficulty. However, recursive modules and especially abstract signatures are more challenging and may not quite fit in F^ω.

Canonical signatures are understood as F^ω-types, and are thus considered up to F^ω notion of type-equivalence, which does not take into account type isomorphisms that would hold in F^ω, or even some additional type-isomorphisms that would hold in a restricted subset of F^ω-types that would suffice to encode signatures. Hence, signatures keep irrelevant information distinguishing some signatures that could otherwise be identified, which in turn prevents some signature transformations during anchoring. For instance, $\exists^{\blacktriangledown} \varphi, \alpha. \mathcal{C}[\varphi\ \alpha]$ (where α and φ are free in the one-hole signature context $\mathcal{C}$) is isomorphic, but not equivalent, to $\exists^{\blacktriangledown}\beta. \mathcal{C}[\beta]$. When φ appears in $\mathcal{C}$, this is no more an isomorphism, but we could still wish to see them as isomorphic as we will never be able to observe the difference in the sublanguage of expressions encoding identities. Characterizing all (or a just a subject of) those transformations and how to exploit them is left for future work.

Currently, we have a dilemma: we can present inferred signature to users in the source syntax at the cost of dealing with the signature avoidance problem and confusing explanations on how to rewrite their own code accordingly. Alternatively, canonical signatures eliminate this artificial problem altogether but depart from the path-based source notation that has proved user-friendly in many cases. Giving the user access to canonical signatures would resolve the mismatch between the reachable and expressible spaces of Figure 2. However, without restrictions on the signatures that the user can write, we believe that subtyping becomes undecidable. Finding a set of *good sense* restrictions to maintain decidability, as well as mixing the path-based and canonical signatures constitutes an interesting research and engineering topic.

The introduction of transparent existential types makes the treatment of applicative and generative functors much simpler: transparent existentials are always used by default (applicative mode) when assembling components and only turned into opaque existentials when necessary (generative mode). Functors are then independent of the mode, just leaving opaque existential in their body and moving transparent ones in front of the functors. Applications of functors are also independent of the mode. This considerably simplifies the treatment of applicative functors which should also benefit to the appealing approach of 1ML. While the goal of 1ML is to remove the stratification between core and module layers, and directly program modules in F^ω, we may explore another path, extending F^ω with minimalist constructs, typically for dealing with primitive lifting of existentials, so that we may program *with modules* in this light extension of F^ω.

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

APPENDIX

A CANONICAL SYSTEM

A.1 Subtyping

A.1.1 Signature subtyping

$$\boxed{\Gamma \overset{\text{can}}{\vdash} \mathcal{C} < \mathcal{C}'}$$

$$\textsc{C-Sub-Sig-Id}\quad \frac{\Gamma \overset{\text{can}}{\vdash} \mathcal{R} < \mathcal{R}'}{\Gamma \overset{\text{can}}{\vdash} (\tau, \mathcal{R}) < (\tau, \mathcal{R}')}$$

$$\textsc{C-Sub-Sig-Struct}\quad \frac{\overline{\mathcal{D}_0} \subseteq \overline{\mathcal{D}} \qquad \Gamma \overset{\text{can}}{\vdash} \overline{\mathcal{D}_0} < \overline{\mathcal{D}}'}{\Gamma \overset{\text{can}}{\vdash} \text{sig } \overline{\mathcal{D}} \text{ end} < \text{sig } \overline{\mathcal{D}}' \text{ end}}$$

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

$$\textsc{C-Sub-Sig-AppFct}\quad \frac{\Gamma, \overline{\alpha}' \overset{\text{can}}{\vdash} \mathcal{C}_a' < \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma, \overline{\alpha}' \overset{\text{can}}{\vdash} \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] < \mathcal{C}'}{\Gamma \overset{\text{can}}{\vdash} \forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C} < \forall \overline{\alpha}'.\mathcal{C}_a' \to \mathcal{C}'}$$

A.1.2 Declaration subtyping

$$\boxed{\Gamma \overset{\text{can}}{\vdash} \mathcal{D} < \mathcal{D}'}$$

$$\textsc{C-Sub-Decl-Val}\quad \Gamma \overset{\text{can}}{\vdash} (\text{val } x : \tau) < (\text{val } x : \tau)$$

$$\textsc{C-Sub-Decl-Type}\quad \Gamma \overset{\text{can}}{\vdash} (\text{type } t = \tau) < (\text{type } t = \tau)$$

$$\textsc{C-Sub-Decl-Mod}\quad \frac{\Gamma \overset{\text{can}}{\vdash} \mathcal{C} < \mathcal{C}'}{\Gamma \overset{\text{can}}{\vdash} (\text{module } X : \mathcal{C}) < (\text{module } X : \mathcal{C}')}$$

$$\textsc{C-Sub-Decl-ModType}\quad \frac{\Gamma, \overline{\alpha} \overset{\text{can}}{\vdash} \mathcal{C} < \mathcal{C}' \qquad \Gamma, \overline{\alpha} \overset{\text{can}}{\vdash} \mathcal{C}' < \mathcal{C}}{\Gamma \overset{\text{can}}{\vdash} (\text{module type } T = \lambda\overline{\alpha}.\mathcal{C}) < (\text{module type } T = \lambda\overline{\alpha}.\mathcal{C}')}$$

A.2 Typing

A.2.1 Signature typing

$$\boxed{\Gamma \overset{\text{can}}{\vdash} \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}}.$$

$$\textsc{C-Typ-Sig-ModType}\quad \frac{\Gamma \overset{\text{can}}{\vdash} P : (\_, \text{sig } \overline{\mathcal{D}} \text{ end}) \qquad \text{module type } T = \lambda\overline{\alpha}.\mathcal{C} \in \overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\vdash} P.T : \lambda\overline{\alpha}.\mathcal{C}}$$

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

$$\textsc{C-Typ-Sig-GenFct}\quad \frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \overset{\text{can}}{\vdash} () \to \mathsf{S} : \lambda\alpha_0.\,(\alpha_0, () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C})}$$

$$\textsc{C-Typ-Sig-AppFct}\quad \frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{S}_a : \lambda\overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a \overset{\text{can}}{\vdash} \mathsf{S} : \lambda\overline{\beta}.\mathcal{C}}{\Gamma \overset{\text{can}}{\vdash} (Y : \mathsf{S}_a) \to \mathsf{S} : \lambda\alpha_0, \overline{\beta'}.(\alpha_0, \forall\overline{\alpha}.\mathcal{C} \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right])}$$

$$\textsc{C-Typ-Sig-Str}\quad \frac{\Gamma \overset{\text{can}}{\vdash}_A \overline{\mathsf{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash} \mathsf{sig}_A\ \overline{\mathsf{D}}\ \mathsf{end} : \lambda\alpha_0, \overline{\alpha}.\,(\alpha_0, \text{sig } \overline{\mathcal{D}} \text{ end})}$$

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

A.2.2 *Declaration typing*

$\boxed{\Gamma \overset{\text{can}}{\vdash}_A \mathsf{D} : \lambda\overline{\alpha}.\mathcal{D}}$.

C-Typ-Decl-Val
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{u} : \tau}{\Gamma \overset{\text{can}}{\vdash}_A (\mathsf{val}\ x : \mathsf{u}) : (\mathsf{val}\ x : \tau)}$$

C-Typ-Decl-Type
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{u} : \tau}{\Gamma \overset{\text{can}}{\vdash}_A (\mathsf{type}\ t = \mathsf{u}) : (\mathsf{type}\ t = \tau)}$$

C-Typ-Decl-TypeAbs
$$\Gamma \overset{\text{can}}{\vdash}_A (\mathsf{type}\ t = A.t) : \lambda\alpha.\, (\mathsf{type}\ t = \alpha)$$

C-Typ-Decl-Mod
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \overset{\text{can}}{\vdash}_A (\mathsf{module}\ X : \mathsf{S}) : \lambda\overline{\alpha}.\, (\mathsf{module}\ X : \mathcal{C})}$$

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

C-Typ-Decl-Empty
$$\Gamma \overset{\text{can}}{\vdash}_A \varnothing : \varnothing$$

C-Typ-Decl-Seq
$$\frac{\Gamma \overset{\text{can}}{\vdash}_A \mathsf{D} : \lambda\overline{\alpha}_1.\mathcal{D} \qquad \Gamma, \overline{\alpha}_1, A.I : \mathcal{D} \overset{\text{can}}{\vdash}_A \overline{\mathsf{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\vdash}_A \mathsf{D}, \overline{\mathsf{D}} : \lambda\overline{\alpha}_1\, \overline{\alpha}.\mathcal{D}, \overline{\mathcal{D}}}$$

A.2.3 *Core type checking extension*

$\boxed{\Gamma \overset{\text{can}}{\vdash} \mathsf{u} : \tau}$.

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

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

A.2.4 *Module typing*

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

C-Typ-Mod-Var
$$\frac{(Y : \mathcal{C}) \in \Gamma}{\Gamma \overset{\text{can}}{\vdash} Y : \mathcal{C}}$$

C-Typ-Mod-Local
$$\frac{(A.X :\ \mathsf{module}\ \ \mathcal{C}) \in \Gamma}{\Gamma \overset{\text{can}}{\vdash} A.X : \mathcal{C}}$$

C-Typ-Mod-Struct
$$\frac{\Gamma \overset{\text{can}}{\vdash}_A \overline{\mathsf{B}} : \exists^{\diamond}\overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash} \mathsf{struct}_A\ \overline{\mathsf{B}}\ \mathsf{end} : \exists^{\diamond}\alpha_0, \overline{\alpha}.\ (\alpha_0, \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end})}$$

C-Typ-Mod-Proj
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{M} : \exists^{\diamond}\overline{\alpha}.\ (\_, \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}) \qquad \mathsf{module}\ X : \mathcal{C} \in \overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\vdash} \mathsf{M}.X : \exists^{\diamond}\overline{\alpha}.\mathcal{C}}$$

C-Typ-Mod-GenFct
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{M} : \exists^{\diamond}\overline{\alpha}.\mathcal{C}}{\Gamma \overset{\text{can}}{\vdash} () \to \mathsf{M} : \exists^{\triangledown}\alpha_0.\ (\alpha_0, () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C})}$$

C-Typ-Mod-AppFct
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{S}_a : \lambda\overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, (Y : \mathcal{C}_a) \overset{\text{can}}{\vdash} \mathsf{M} : \exists^{\triangledown}\overline{\beta}.\mathcal{C}}{\Gamma \overset{\text{can}}{\vdash} (Y : \mathsf{S}_a) \to \mathsf{M} : \exists^{\triangledown}\alpha_0, \overline{\beta'}.\ (\alpha_0, (\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C})[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}])}$$

C-Typ-Mod-AppGen
$$\frac{\Gamma \overset{\text{can}}{\vdash} P : (\_, () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C})}{\Gamma \overset{\text{can}}{\vdash} P() : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}$$

C-Typ-Mod-AppApp
$$\frac{\Gamma \overset{\text{can}}{\vdash} P : (\_, \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}) \qquad \Gamma \overset{\text{can}}{\vdash} P' : \mathcal{C}' \qquad \Gamma \overset{\text{can}}{\vdash} \mathcal{C}' < \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}]}{\Gamma \overset{\text{can}}{\vdash} P(P') : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]}$$

C-Typ-Mod-Ascr
$$\frac{\Gamma \overset{\text{can}}{\vdash} P : \mathcal{C} \qquad \Gamma \overset{\text{can}}{\vdash} \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}' \qquad \Gamma \overset{\text{can}}{\vdash} \mathcal{C} < \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]}{\Gamma \overset{\text{can}}{\vdash} (P : \mathsf{S}) : \exists^{\triangledown}\overline{\alpha}.\mathcal{C}'}$$

C-Typ-Mod-Mode
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{M} : \exists^{\triangledown}\overline{\alpha}.\mathcal{C}}{\Gamma \overset{\text{can}}{\vdash} \mathsf{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}$$

A.2.5 *Binding typing* $\boxed{\Gamma \overset{\text{can}}{\vdash} \mathsf{B} : \exists^{\diamond}\overline{\alpha}.\mathcal{D}}$.

C-Typ-Bind-Let
$$\frac{\Gamma \overset{\text{can}}{\vdash}{}^{\diamond}\ \mathsf{e} : \tau}{\Gamma \overset{\text{can}}{\vdash}_A (\mathtt{let}\ x = \mathsf{e}) : \exists^{\diamond}.(\mathsf{val}\ x : \tau)}$$

C-Typ-Bind-Type
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{u} : \tau}{\Gamma \overset{\text{can}}{\vdash}_A (\mathsf{type}\ t = \mathsf{u}) : (\mathsf{type}\ t = \tau)}$$

C-Typ-Bind-Mod
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{M} : \exists^{\diamond}\overline{\alpha}.\mathcal{C}}{\Gamma \overset{\text{can}}{\vdash}_A (\mathtt{module}\ X = \mathsf{M}) : (\exists^{\diamond}\overline{\alpha}.\mathsf{module}\ X : \mathcal{C})}$$

C-Typ-Bind-ModType
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \overset{\text{can}}{\vdash}_A (\mathtt{module\ \ type}\ T = \mathsf{S}) : (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C})}$$

C-Typ-Bind-Empty
$$\Gamma \overset{\text{can}}{\vdash}_A \varnothing : \varnothing$$

C-Typ-Bind-Seq
$$\frac{\Gamma \overset{\text{can}}{\vdash}_A \mathsf{B} : \exists^{\diamond}\overline{\alpha}_1.\mathcal{D} \qquad \Gamma, \overline{\alpha}_1, A.I : \mathcal{D} \overset{\text{can}}{\vdash}_A \overline{\mathsf{B}} : \exists^{\diamond}\overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\vdash}_A \mathsf{B}, \overline{\mathsf{B}} : \exists^{\diamond}\overline{\alpha}_1, \overline{\alpha}.\mathcal{D}, \overline{\mathcal{D}}}$$

A.2.6 *Core expression typing extension* $\boxed{\Gamma \overset{\text{can}}{\vdash}{}^{\diamond}\ \mathsf{e} : \tau}$.

C-Typ-Type-Path
$$\frac{\Gamma \overset{\text{can}}{\vdash} P : \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{val}\ x : \tau \in \overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\vdash}{}^{\diamond}\ P.x : \tau}$$

C-Typ-Type-Local
$$\frac{(A.x :\ \mathsf{val}\ \ \tau) \in \Gamma}{\Gamma \overset{\text{can}}{\vdash}{}^{\diamond}\ A.x : \tau}$$

B ANCHORING

B.1 Anchoring of environments $\boxed{\Gamma \hookrightarrow \Gamma_{\mathsf{src}} : \theta_\Gamma}$

A-Env-Empty
$$\varnothing \hookrightarrow \varnothing : \varnothing$$

A-Env-FctArg
$$\frac{\Gamma, \overline{\alpha} \overset{\text{anch}}{\vdash} \mathcal{C} \overset{Y}{\hookrightarrow} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \theta \qquad \overline{\alpha} = \mathrm{dom}(\theta)}{\Gamma, \overline{\alpha}, (Y : \mathcal{C}) \hookrightarrow \Gamma_{\mathsf{src}}, (Y : \mathsf{S}) : \theta_\Gamma \uplus \theta}$$

A-Env-Decl
$$\frac{\Gamma \overset{\text{anch}}{\vdash} \mathcal{D} \overset{A}{\hookrightarrow} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{D} : \theta \qquad A.I \notin \Gamma}{\Gamma, (A.I : \mathcal{D}) \hookrightarrow \Gamma_{\mathsf{src}}, (A.I : \mathsf{D}) : \theta_\Gamma \uplus \theta}$$

A-Env-Abs
$$\frac{\Gamma \hookrightarrow \Gamma_{\mathsf{src}} : \theta_\Gamma}{\Gamma, \overline{\alpha} \hookrightarrow \Gamma_{\mathsf{src}} : \theta_\Gamma}$$

B.2 Signature anchoring $\boxed{\Gamma \overset{\text{anch}}{\vdash} \mathcal{C} \overset{P?}{\hookrightarrow} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \theta}$

Identity signatures.

A-Sig-Id-Ignore
$$\frac{\Gamma \overset{\text{anch}}{\vdash} \mathcal{R} \overset{\varnothing}{\hookrightarrow} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \theta \qquad \varphi \notin \mathrm{dom}(\theta_\Gamma) \qquad \Gamma = \Gamma_0, \varphi, \Gamma_1 \qquad \mathrm{args}(\Gamma_1) = \overline{\alpha}}{\Gamma \overset{\text{anch}}{\vdash} (\varphi(\overline{\alpha}), \mathcal{R}) \overset{\varnothing}{\hookrightarrow} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \theta \uplus (\varphi \mapsto (\times, \varnothing))}$$

A-Sig-Id-Anchor
$$\frac{\Gamma \overset{\text{anch}}{\vdash} \mathcal{R} \overset{P}{\hookrightarrow} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \theta \qquad \varphi \notin \mathrm{dom}(\theta_\Gamma) \qquad \Gamma = \Gamma_0, \varphi, \Gamma_1 \qquad \mathrm{args}(\Gamma_1) = \overline{\alpha}}{\Gamma \overset{\text{anch}}{\vdash} (\varphi(\overline{\alpha}), \mathcal{R}) \overset{P}{\hookrightarrow} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \theta \uplus (\varphi \mapsto ((P, \mathcal{R}), \overline{\alpha}))}$$

A-Sig-Id-TrAsrc
$$\frac{\Gamma \overset{\text{anch}}{\vdash} \tau \hookrightarrow \Gamma_{\mathsf{src}}; \theta_\Gamma \vdash (P, \mathcal{R}') \qquad \Gamma \overset{\text{anch}}{\vdash} \mathcal{R} \overset{P?}{\longrightarrow} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \varnothing \qquad \Gamma \overset{\text{can}}{\vdash} \mathcal{R}' < \mathcal{R}}{\Gamma \overset{\text{anch}}{\vdash} (\tau, \mathcal{R}) \overset{P?}{\longrightarrow} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} (= P < \mathsf{S}) : \varnothing}$$

Structural signatures.

A-Sig-FctApp

$$\frac{\mathrm{dom}(\theta_a) = \overline{\alpha} \qquad \begin{array}{c}\Gamma, \overline{\alpha} \overset{\text{anch}}{\vdash} \mathcal{C}_a \xhookrightarrow{Y} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{S}_a : \theta_a \\ \Gamma, \overline{\alpha}, (Y : \mathcal{C}_a) \overset{\text{anch}}{\vdash} \mathcal{C} \xhookrightarrow{P(Y)?} (\Gamma_{\mathsf{src}}, Y : \mathsf{S}_a)\,; \theta_\Gamma \uplus Y.\theta_a \overset{\text{src}}{\vdash} \mathsf{S} : \theta\end{array}}{\Gamma \overset{\text{anch}}{\vdash} \forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \xhookrightarrow{P?} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} (Y : \mathsf{S}_a) \to \mathsf{S} : \lambda\overline{\alpha}.\lambda(Y : \mathcal{C}_a).\theta}$$

A-Sig-FctGen

$$\frac{\Gamma, \overline{\alpha} \overset{\text{anch}}{\vdash} \mathcal{C} \xhookrightarrow{\varnothing} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \theta \qquad \mathrm{dom}(\theta) \subseteq \overline{\alpha}}{\Gamma \overset{\text{anch}}{\vdash} () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \xhookrightarrow{P?} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} () \to \mathsf{S} : \varnothing}$$

A-Sig-Str

$$\frac{\Gamma \overset{\text{anch}}{\vdash} \overline{\mathcal{D}} \xhookrightarrow{A} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \overline{\mathsf{D}} : \theta \qquad A \notin \Gamma_{\mathsf{src}}}{\Gamma \overset{\text{anch}}{\vdash} \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \xhookrightarrow{P?} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathtt{sig}_A\ \overline{\mathsf{D}}\ \mathsf{end} : \theta[A \mapsto P?]}$$

B.3 Declaration anchoring

$$\boxed{\Gamma \overset{\text{anch}}{\vdash} \mathcal{D} \xhookrightarrow{A} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{D} : \theta}$$

A-Decl-Val

$$\frac{\Gamma \overset{\text{anch}}{\vdash} \tau \hookrightarrow \Gamma_{\mathsf{src}}; \theta_\Gamma \vdash \mathsf{u}}{\Gamma \overset{\text{anch}}{\vdash} \mathsf{val}\ x : \tau \xhookrightarrow{A} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} (\mathtt{val}\ x : \mathsf{u}) : \varnothing}$$

A-Decl-Type

$$\frac{\Gamma \overset{\text{anch}}{\vdash} \tau \hookrightarrow \Gamma_{\mathsf{src}}; \theta_\Gamma \vdash \mathsf{u}}{\Gamma \overset{\text{anch}}{\vdash} \mathsf{type}\ t = \tau \xhookrightarrow{A} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathtt{type}\ t = \mathsf{u} : \varnothing}$$

A-Decl-Anchor

$$\frac{\Gamma \hookrightarrow \Gamma_{\mathsf{src}} : \theta_\Gamma \qquad \varphi \notin \mathrm{dom}(\theta_\Gamma) \qquad \Gamma = \Gamma_0, \varphi, \Gamma_1 \qquad \mathrm{args}(\Gamma_1) = \overline{\alpha}}{\Gamma \overset{\text{anch}}{\vdash} \mathsf{type}\ t = \varphi(\overline{\alpha}) \xhookrightarrow{A} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathtt{type}\ t = A.t : (\varphi \mapsto (A.t, \overline{\alpha}))}$$

A-Decl-Mod

$$\frac{\Gamma \overset{\text{anch}}{\vdash} \mathcal{C} \xhookrightarrow{X} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \theta}{\Gamma \overset{\text{anch}}{\vdash} \mathsf{module}\ X : \mathcal{C} \xhookrightarrow{A} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} (\mathtt{module}\ X : \mathsf{S}) : A.\theta}$$

A-Decl-ModType

$$\frac{\Gamma, \overline{\alpha} \overset{\text{anch}}{\vdash} \mathcal{C} \xhookrightarrow{\varnothing} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \theta \qquad \mathrm{dom}(\theta) = \overline{\alpha}}{\Gamma \overset{\text{anch}}{\vdash} (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}) \xhookrightarrow{A} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathtt{module\ \ type}\ T = \mathsf{S} : \varnothing}$$

A-Decl-Empty

$$\frac{\Gamma \hookrightarrow \Gamma_{\mathsf{src}} : \theta_\Gamma}{\Gamma \overset{\text{anch}}{\vdash} \varnothing \xhookrightarrow{A} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \varnothing : \varnothing}$$

A-Decl-Seq

$$\frac{\Gamma \overset{\text{anch}}{\vdash} \mathcal{D} \xhookrightarrow{A} \Gamma_{\mathsf{src}}; \theta_\Gamma \overset{\text{src}}{\vdash} \mathsf{D} : \theta_1 \qquad \Gamma, \mathcal{D} \overset{\text{anch}}{\vdash} \overline{\mathcal{D}} \xhookrightarrow{A} \Gamma_{\mathsf{src}}, \mathsf{D}; \theta \uplus \theta_1 \overset{\text{src}}{\vdash} \overline{\mathsf{D}} : \theta_2}{\Gamma \overset{\text{anch}}{\vdash} \mathcal{D}, \overline{\mathcal{D}} \hookrightarrow \Gamma_{\mathsf{src}}; \theta_\Gamma \vdash \mathsf{D}, \overline{\mathsf{D}} : \theta_1 \uplus \theta_2}$$

B.4 Anchoring of abstract types

$$\boxed{\Gamma \overset{\text{anch}}{\vdash} \tau \hookrightarrow \Gamma_{\mathsf{src}}; \theta_\Gamma \vdash \mathsf{u}}$$

A-Type-Star

$$\frac{\theta_\Gamma(\alpha) = (z, \varnothing)}{\Gamma \overset{\text{anch}}{\vdash} \alpha \hookrightarrow \Gamma_{\mathsf{src}}; \theta_\Gamma \vdash z}$$

A-Type-TypeApplication

$$\frac{\overline{\tau} = \_ :: \overline{\rho_i, \overline{\tau}_i} \qquad \Gamma \overset{\text{anch}}{\vdash} \overline{\rho}_i \hookrightarrow \Gamma_{\mathsf{src}}; \theta_\Gamma \vdash \overline{(P_i, \mathcal{R}_i)} \qquad \Gamma \overset{\text{anch}}{\vdash} \theta_\Gamma(\tau)\overline{(\rho_i, \overline{\tau}_i)(P_i, (\rho_i, \mathcal{R}_i))} = (z, \overline{\tau})}{\Gamma \overset{\text{anch}}{\vdash} \varphi\,(\overline{\tau}) \hookrightarrow \Gamma_{\mathsf{src}}; \theta_\Gamma \vdash z}$$

A-Map-apply

$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathcal{C} < \overline{\mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}]} \qquad \Gamma \overset{\text{anch}}{\vdash} \left(\overline{\lambda\overline{\alpha}'.\lambda(Y' : \mathcal{C}'_a)}.(z_0, \overline{\tau_0})\right) \left(\overline{(\overline{\tau}', (P', \mathcal{C}'))}\right)[Y \mapsto P][\overline{\alpha} \mapsto \overline{\tau}] = (z, \overline{\tau_1})}{\Gamma \overset{\text{anch}}{\vdash} \left(\lambda\overline{\alpha}.\lambda(Y : \mathcal{C}_a).\overline{\lambda\overline{\alpha}'.\lambda(Y' : \mathcal{C}'_a)}.(z_0, \overline{\tau_0})\right) \left((\overline{\tau}, (P, \mathcal{C}))\overline{(\overline{\tau}', (P', \mathcal{C}'))}\right) = (z, \overline{\tau_1})}$$

C SOURCE SYSTEM

C.1 Strengthening

C.1.1 Signature strengthening $\boxed{\mathsf{S}/P \gg \mathbf{C}}$.

S-Str-Sig-Alias
$$(= P' < \mathbf{C})\,/P \gg (= P' < \mathbf{C})$$

S-Str-Sig-GenFct
$$(() \to \mathsf{S})\,/P \gg (= P < () \to \mathsf{S})$$

S-Str-Sig-AppFct
$$\frac{\mathsf{S}/P(Y) \gg \mathbf{C}}{((Y : \mathsf{S}_a) \to \mathsf{S})\,/P \gg (= P < (Y : \mathsf{S}_a) \to \mathbf{C})}$$

S-Str-Sig-Struct
$$\frac{\overline{\mathsf{D}}[A \mapsto P]\,/P \gg \overline{\mathbf{D}}}{\mathsf{sig}_A\ \overline{\mathsf{D}}\ \mathsf{end}/P \gg (= P < \mathsf{sig}\ \overline{\mathbf{D}}\ \mathsf{end})}$$

C.1.2 Declaration strengthening – $\mathsf{D}/P \gg \mathsf{D}'$.

S-Str-Decl-Val
$$(\mathtt{val}\ x : \mathsf{u})\,/P \gg \mathtt{val}\ x : \mathsf{u}$$

S-Str-Decl-Type
$$(\mathtt{type}\ t = \mathsf{u})\,/P \gg \mathtt{type}\ t = \mathsf{u}$$

S-Str-Decl-Mod
$$\frac{\mathsf{S}/P.X \gg \mathbf{C}}{(\mathtt{module}\ X : \mathsf{S})\,/P \gg \mathtt{module}\ X : \mathbf{C}}$$

S-Str-Decl-ModType
$$(\mathtt{module\ type}\ T = \mathsf{S})\,/P \gg \mathtt{module\ type}\ T = \mathsf{S}$$

C.2 Path typing $\boxed{\Gamma \overset{\mathsf{src}}{\vdash} P : \mathbf{C}}$

S-Typ-Path-LocalMod
$$\frac{(A.X : \mathtt{module}\ \ \mathbf{C}) \in \Gamma}{\Gamma \overset{\mathsf{src}}{\vdash} A.X : \mathbf{C}}$$

S-Typ-Path-Proj
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash} P : (= P' < \mathsf{sig}\ \overline{\mathbf{D}}\ \mathsf{end}) \qquad (\mathtt{module}\ X : \mathbf{C}) \in \overline{\mathbf{D}}}{\Gamma \overset{\mathsf{src}}{\vdash} P.X : \mathbf{C}}$$

S-Typ-Path-FctArg
$$\frac{(Y : \mathbf{C}) \in \Gamma}{\Gamma \overset{\mathsf{src}}{\vdash} Y : \mathbf{C}}$$

S-Typ-Path-AppFct
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash} P_f : \left(= P'_f < (Y : \mathsf{S}) \to \mathbf{C}\right) \qquad \Gamma \overset{\mathsf{src}}{\vdash} P : (= P' < \mathbf{R}) \qquad \Gamma \overset{\mathsf{src}}{\vdash} \mathbf{R} \prec \mathsf{S}}{\Gamma \overset{\mathsf{src}}{\vdash} P_f(P) : \left(= P'_f(P') < \mathbf{C}\left[Y \mapsto P'\right]\right)}$$

C.3 Subtyping

Rules shared between subtyping and abstraction subtyping are written with the symbol $\prec$.

C.3.1 Signature subtyping $\boxed{\Gamma \overset{\mathsf{src}}{\vdash} \mathsf{S} \prec \mathsf{S}'}$.

Common rules

S-Sub-Sig-TrAscrAbs
$$\Gamma \overset{\mathsf{src}}{\vdash} (= P < \mathbf{R}) \prec \mathbf{R}$$

S-Sub-Sig-TrAscr
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash} P \prec P' \qquad \Gamma \overset{\mathsf{src}}{\vdash} \mathbf{R} \prec \mathbf{R}'}{\Gamma \overset{\mathsf{src}}{\vdash} (= P < \mathbf{R}) \prec (= P' < \mathbf{R}')}$$

S-Sub-Sig-GenFct
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash} \mathsf{S} \prec \mathsf{S}'}{\Gamma \overset{\mathsf{src}}{\vdash} () \to \mathsf{S} \prec () \to \mathsf{S}'}$$

S-Sub-Sig-AppFct
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash} \mathsf{S}_a \prec \mathsf{S}'_a \qquad \mathsf{S}'_a/Y \gg \mathbf{C}'_a \qquad \Gamma, Y : \mathbf{C}'_a \overset{\mathsf{src}}{\vdash} \mathsf{S} \prec \mathsf{S}'}{\Gamma \overset{\mathsf{src}}{\vdash} (Y : \mathsf{S}_a) \to \mathsf{S} \prec (Y : \mathsf{S}'_a) \to \mathsf{S}'}$$

General subtyping rule

S-SubGen-Sig-Struct
$$\frac{\overline{\mathsf{D}_0} \subseteq \overline{\mathsf{D}} \qquad \Gamma, \overline{A.\mathsf{D}} \overset{\mathsf{src}}{\vdash}_A \overline{\mathsf{D}_0} < \overline{\mathsf{D}'}}{\Gamma \overset{\mathsf{src}}{\vdash} \mathsf{sig}_A\ \overline{\mathsf{D}}\ \mathsf{end} < \mathsf{sig}_A\ \overline{\mathsf{D}'}\ \mathsf{end}}$$

Abstraction subtyping rule

$$\textsc{S-SubAbs-Sig-Struct}\quad \frac{\Gamma, \overline{A.\mathsf{D}} \overset{\text{src}}{\vdash}_A \overline{\mathsf{D}} \sqsubset \overline{\mathsf{D}}'}{\Gamma \overset{\text{src}}{\vdash} \mathtt{sig}_A\ \overline{\mathsf{D}}\ \mathtt{end} \sqsubset \mathtt{sig}_A\ \overline{\mathsf{D}}'\ \mathtt{end}}$$

C.3.2 Declaration subtyping

$\boxed{\Gamma \overset{\text{src}}{\vdash}_A \mathsf{D} \prec \mathsf{D}'}$.

$$\textsc{S-Sub-Decl-Val}\quad \frac{\Gamma \overset{\text{src}}{\vdash} \mathsf{u} \prec \mathsf{u}'}{\Gamma \overset{\text{src}}{\vdash}_A (\mathtt{val}\ x : \mathsf{u}) \prec (\mathtt{val}\ x : \mathsf{u}')}$$

$$\textsc{S-Sub-Decl-Type}\quad \frac{\Gamma \overset{\text{src}}{\vdash} \mathsf{u} \prec \mathsf{u}'}{\Gamma \overset{\text{src}}{\vdash}_A (\mathtt{type}\ t = \mathsf{u}) \prec (\mathtt{type}\ t = \mathsf{u}')}$$

$$\textsc{S-Sub-Decl-Mod}\quad \frac{\Gamma \overset{\text{src}}{\vdash} \mathsf{S} \prec \mathsf{S}'}{\Gamma \overset{\text{src}}{\vdash}_A (\mathtt{module}\ X : \mathsf{S}) \prec (\mathtt{module}\ X : \mathsf{S}')}$$

$$\textsc{S-Sub-Decl-ModType}\quad \frac{\Gamma \overset{\text{src}}{\vdash} \mathsf{S} \prec \mathsf{S}' \qquad \Gamma \overset{\text{src}}{\vdash} \mathsf{S}' \prec \mathsf{S}}{\Gamma \overset{\text{src}}{\vdash}_A (\mathtt{module\ type}\ T = \mathsf{S}) \prec (\mathtt{module\ type}\ T = \mathsf{S}')}$$

C.3.3 Type subtyping

$\boxed{\Gamma \overset{\text{src}}{\vdash} \mathsf{u} \prec \mathsf{u}'}$.

$$\textsc{S-Sub-Typ-Equiv}\quad \frac{\Gamma \overset{\text{src}}{\vdash} \mathsf{u} : \mathsf{u}_0 \qquad \Gamma \overset{\text{src}}{\vdash} \mathsf{u}' : \mathsf{u}_0}{\Gamma \overset{\text{src}}{\vdash} \mathsf{u} \prec \mathsf{u}'}$$

C.3.4 Path subtyping

$\boxed{\Gamma \overset{\text{src}}{\vdash} P \prec P'}$.

$$\textsc{S-Sub-Path}\quad \frac{\Gamma \overset{\text{src}}{\vdash} P : (= P_0 < \mathbf{R}) \qquad \Gamma \overset{\text{src}}{\vdash} P' : (= P_0 < \mathbf{R}') \qquad \Gamma \overset{\text{src}}{\vdash} \mathbf{R} \prec \mathbf{R}'}{\Gamma \overset{\text{src}}{\vdash} P \prec P'}$$

C.4 Typing

C.4.1 Signature typing

$\boxed{\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \mathsf{S}'}$.

$$\textsc{S-Typ-Sig-ModType}\quad \frac{\Gamma \overset{\text{src}}{\vdash} P : (= \_ < \mathtt{sig}\ \overline{\mathbf{D}}\ \mathtt{end}) \qquad \mathtt{module\ type}\ T = \mathsf{S} \in \overline{\mathbf{D}}}{\Gamma \overset{\text{src}}{\vdash} P.T : \mathsf{S}}$$

$$\textsc{S-Typ-Sig-LocalModType}\quad \frac{A.T : \mathtt{module\ type}\ \mathsf{S} \in \Gamma}{\Gamma \overset{\text{src}}{\vdash} A.T : \mathsf{S}}$$

$$\textsc{S-Typ-Sig-GenFct}\quad \frac{\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \mathsf{S}'}{\Gamma \overset{\text{src}}{\vdash} () \to \mathsf{S} : () \to \mathsf{S}'}$$

$$\textsc{S-Typ-Sig-AppFct}\quad \frac{\Gamma \overset{\text{src}}{\vdash} \mathsf{S}_a : \mathsf{S}'_a \qquad \mathsf{S}'_a / Y \gg \mathbf{C}_a \qquad \Gamma, (Y : \mathbf{C}_a) \overset{\text{src}}{\vdash} \mathsf{S} : \mathsf{S}'}{\Gamma \overset{\text{src}}{\vdash} (Y : \mathsf{S}_a) \to \mathsf{S} : (Y : \mathsf{S}'_a) \to \mathsf{S}'}$$

$$\textsc{S-Typ-Sig-Str}\quad \frac{\Gamma \overset{\text{src}}{\vdash}_A \overline{\mathsf{D}} : \overline{\mathsf{D}}' \qquad A \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash} \mathtt{sig}_A\ \overline{\mathsf{D}}\ \mathtt{end} : \mathtt{sig}_A\ \overline{\mathsf{D}}'\ \mathtt{end}}$$

$$\textsc{S-Typ-Sig-TrAscr}\quad \frac{\Gamma \overset{\text{src}}{\vdash} P : \mathbf{C} \qquad \Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \mathsf{S}' \qquad \mathsf{S}' / P \gg \mathbf{C}' \qquad \Gamma \overset{\text{src}}{\vdash} \mathbf{C} \prec \mathbf{C}'}{\Gamma \overset{\text{src}}{\vdash} (= P < \mathsf{S}) : \mathbf{C}'}$$

C.4.2 Declaration typing

$\boxed{\Gamma \overset{\text{src}}{\vdash}_A \mathsf{D} : \mathsf{D}'}$.

S-Typ-Decl-Val
$$\frac{\Gamma \overset{\text{src}}{\vdash} \mathsf{u} : \mathsf{u}'}{\Gamma \overset{\text{src}}{\vdash}_A (\mathtt{val}\ x : \mathsf{u}) : (\mathtt{val}\ x : \mathsf{u}')}$$

S-Typ-Decl-Type
$$\frac{\Gamma \overset{\text{src}}{\vdash} \mathsf{u} : \mathsf{u}'}{\Gamma \overset{\text{src}}{\vdash}_A (\mathtt{type}\ t = \mathsf{u}) : (\mathtt{type}\ t = \mathsf{u}')}$$

S-Typ-Decl-TypeAbs
$$\Gamma \overset{\text{src}}{\vdash}_A (\mathtt{type}\ t = A.t) : (\mathtt{type}\ t = A.t)$$

S-Typ-Decl-Mod
$$\frac{\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \mathsf{S}'}{\Gamma \overset{\text{src}}{\vdash}_A (\mathtt{module}\ X : \mathsf{S}) : (\mathtt{module}\ X : \mathsf{S}')}$$

S-Typ-Decl-Empty
$$\Gamma \overset{\text{src}}{\vdash}_A \varnothing : \varnothing$$

S-Typ-Decl-ModType
$$\frac{\Gamma \overset{\text{src}}{\vdash} \mathsf{S} : \mathsf{S}'}{\Gamma \overset{\text{src}}{\vdash}_A (\mathtt{module\ \ type}\ T = \mathsf{S}) : (\mathtt{module\ \ type}\ T = \mathsf{S}')}$$

S-Typ-Decl-Seq
$$\frac{\Gamma \overset{\text{src}}{\vdash}_A \mathsf{D}_1 : \mathsf{D}_1' \qquad \Gamma, A.I_1 : \mathsf{D}_1' \overset{\text{src}}{\vdash}_A \overline{\mathsf{D}} : \overline{\mathsf{D}}'}{\Gamma \overset{\text{src}}{\vdash}_A (\mathsf{D}_1, \overline{\mathsf{D}}) : (\mathsf{D}_1', \overline{\mathsf{D}}')}$$

C.4.3 Core type checking extension

$\boxed{\Gamma \overset{\text{src}}{\vdash} \mathsf{u} : \mathsf{u}'}$.

S-Typ-Type-LocalType
$$\frac{A.t : \mathtt{type}\ \ \mathsf{u} \in \Gamma}{\Gamma \overset{\text{src}}{\vdash} A.t : \mathsf{u}}$$

S-Typ-Type-QualifiedPathType
$$\frac{\Gamma \overset{\text{src}}{\vdash} P : (= P' < \mathtt{sig}\ \overline{\mathbf{D}}\ \mathtt{end}) \qquad \mathtt{type}\ t = \mathsf{u} \in \overline{\mathbf{D}}}{\Gamma \overset{\text{src}}{\vdash} P.t : \mathsf{u}}$$

C.4.4 Module typing

$\boxed{\Gamma \overset{\text{src}}{\vdash}^{\diamond} \mathsf{M} : \mathsf{S}}$.

S-Typ-Mod-Path
$$\frac{\Gamma_{\text{src}} \overset{\text{src}}{\vdash} P : \mathbf{C}}{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}^{\diamond} P : \mathbf{C}}$$

S-Typ-Mod-Mode
$$\frac{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}^{\triangledown} \mathsf{M} : \mathsf{S}}{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}^{\blacktriangledown} \mathsf{M} : \mathsf{S}}$$

S-Typ-Mod-AppFct
$$\frac{\Gamma_{\text{src}} \overset{\text{src}}{\vdash} \mathsf{S}_a : \mathsf{S}_a' \qquad \mathsf{S}_a'/Y \gg \mathbf{C}_a \qquad \Gamma_{\text{src}}; (Y : \mathbf{C}_a) \overset{\text{src}}{\vdash}^{\diamond} \mathsf{M} : \mathsf{S}}{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}^{\diamond} (Y : \mathsf{S}_a) \longrightarrow \mathsf{M} : (Y : \mathsf{S}_a') \longrightarrow \mathsf{S}}$$

S-Typ-Mod-GenFct
$$\frac{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}^{\diamond} \mathsf{M} : \mathsf{S}}{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}^{\diamond} () \longrightarrow \mathsf{M} : () \longrightarrow \mathsf{S}}$$

S-Typ-Mod-AppGen
$$\frac{\Gamma_{\text{src}} \overset{\text{src}}{\vdash} P : () \longrightarrow \mathsf{S}}{\mathtt{gen} \overset{\text{src}}{\vdash}^{\diamond} \Gamma_{\text{src}} : P()\mathsf{S}}$$

S-Typ-Mod-Struct
$$\frac{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}_A^{\diamond} \overline{\mathsf{B}} : \overline{\mathsf{D}} \qquad A \notin \Gamma_{\text{src}}}{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}^{\diamond} \mathtt{struct}_A\ \overline{\mathsf{B}}\ \mathtt{end} : \mathtt{sig}_A\ \overline{\mathsf{D}}\ \mathtt{end}}$$

S-Typ-Mod-Proj
$$\frac{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}^{\diamond} \mathsf{M} : \mathsf{S} \qquad \Gamma_{\text{src}} \overset{\text{src}}{\vdash} \mathsf{S} \sqsubset \mathtt{sig}_A\ \overline{\mathsf{D}}\ \mathtt{end} \qquad \mathtt{module}\ X : \mathsf{S}' \in \overline{\mathsf{D}} \qquad \Gamma_{\text{src}} \overset{\text{src}}{\vdash} \mathsf{S}' : \mathsf{S}'}{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}^{\diamond} \mathsf{M}.X : \mathsf{S}'}$$

S-Typ-Mod-Ascription
$$\frac{\Gamma_{\text{src}} \overset{\text{src}}{\vdash} P : \mathbf{C} \qquad \Gamma_{\text{src}} \overset{\text{src}}{\vdash} \mathsf{S} : \mathsf{S}' \qquad \Gamma_{\text{src}} \overset{\text{src}}{\vdash} \mathbf{C} < \mathsf{S}'}{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}^{\diamond} (P : \mathsf{S}) : \mathsf{S}'}$$

C.4.5 *Binding typing* – $\boxed{\Gamma \overset{\text{src}}{\vdash}{}^{\diamond}_{A} \mathtt{B} : \mathtt{D}}$.

S-Typ-Bind-Let
$$\dfrac{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}{}^{\diamond} \mathtt{e} : \mathtt{u}}{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}{}^{\diamond}_{A} (\mathtt{let}\ x = \mathtt{e}) : (\mathtt{val}\ x : \mathtt{u})}$$

S-Typ-Bind-Typ-Binde
$$\dfrac{\Gamma_{\text{src}} \overset{\text{src}}{\vdash} \mathtt{u} : \mathtt{u}'}{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}{}^{\diamond}_{A} (\mathtt{type}\ t = \mathtt{u}) : (\mathtt{type}\ t = \mathtt{u}')}$$

S-Typ-Bind-AbsType
$$\Gamma_{\text{src}} \overset{\text{src}}{\vdash}{}^{\diamond}_{A} (\mathtt{type}\ t = A.t) : (\mathtt{type}\ t = A.t)$$

S-Typ-Bind-Mod
$$\dfrac{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}{}^{\diamond} \mathtt{M} : \mathtt{S}}{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}{}^{\diamond}_{A} (\mathtt{module}\ X = \mathtt{M}) : (\mathtt{module}\ X : \mathtt{S})}$$

S-Typ-Bind-ModType
$$\dfrac{\Gamma_{\text{src}} \overset{\text{src}}{\vdash} \mathtt{S} : \mathtt{S}'}{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}{}^{\diamond}_{A} (\mathtt{module\ type}\ T = \mathtt{S}) : (\mathtt{module\ type}\ T = \mathtt{S}')}$$

S-Typ-Bind-Empty
$$\Gamma_{\text{src}} \overset{\text{src}}{\vdash}{}^{\diamond}_{A} \varnothing : \varnothing$$

S-Typ-Bind-Seq
$$\dfrac{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}{}^{\diamond}_{A} \mathtt{B} : \mathtt{D} \qquad \Gamma_{\text{src}}, A.I : \mathtt{D} \overset{\text{src}}{\vdash}{}^{\diamond}_{A} \overline{\mathtt{B}} : \overline{\mathtt{D}}}{\Gamma_{\text{src}} \overset{\text{src}}{\vdash}{}^{\diamond}_{A} \mathtt{B}, \overline{\mathtt{B}} : \mathtt{D}, \overline{\mathtt{D}}}$$

C.4.6 *Core expression typing extension* $\boxed{\Gamma \overset{\text{src}}{\vdash}{}^{\diamond} \mathtt{e} : \mathtt{u}}$.

S-Typ-Type-LocalType
$$\dfrac{A.x : \mathtt{val}\ \ \mathtt{u} \in \Gamma}{\Gamma \overset{\text{src}}{\vdash}{}^{\diamond} A.x : \mathtt{u}}$$

S-Typ-Type-QualifiedPathType
$$\dfrac{\Gamma \overset{\text{src}}{\vdash} P : (=\_ \ < \mathtt{sig}\ \overline{\mathbf{D}}\ \mathtt{end}) \qquad \mathtt{val}\ x : \mathtt{u} \in \overline{\mathbf{D}}}{\Gamma \overset{\text{src}}{\vdash}{}^{\diamond} P.x : \mathtt{u}}$$

D F$^{\omega}$

D.1 Implementation of transparent existentials

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

D.2 $\overset{F\omega}{\vdash} \Gamma$ – Environment and kind checking

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

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

D.3 $\Gamma \overset{F\omega}{\vdash} \mathsf{u} : \kappa$ – Type checking

$$\frac{\Gamma \overset{F\omega}{\vdash} \tau_1 : \star \qquad \Gamma \overset{F\omega}{\vdash} \tau_2 : \star}{\Gamma \overset{F\omega}{\vdash} \tau_1 \longrightarrow \tau_2 : \star} \qquad \frac{\Gamma \overset{F\omega}{\vdash} \tau : \star \qquad \overset{F\omega}{\vdash} \Gamma z}{\Gamma \overset{F\omega}{\vdash} \{\overline{\ell_I : \tau}\} : \star} \qquad \frac{\overset{F\omega}{\vdash} \Gamma \qquad \alpha : \kappa \in \Gamma}{\Gamma \overset{F\omega}{\vdash} \alpha : \kappa} \qquad \frac{\Gamma, \alpha : \kappa \overset{F\omega}{\vdash} \tau : \star}{\Gamma \overset{F\omega}{\vdash} \forall (\alpha : \kappa). \tau : \star}$$

$$\frac{\Gamma, \varkappa \overset{F\omega}{\vdash} \tau : \star}{\Gamma \overset{F\omega}{\vdash} \forall \varkappa. \tau : \star} \qquad \frac{\Gamma, \alpha : \kappa \overset{F\omega}{\vdash} \tau : \star}{\Gamma \overset{F\omega}{\vdash} \exists^{\blacktriangledown} (\alpha : \kappa). \tau : \star} \qquad \frac{\Gamma, \alpha : \kappa \overset{F\omega}{\vdash} \tau : \kappa'}{\Gamma \overset{F\omega}{\vdash} \Lambda (\alpha : \kappa). \tau : \kappa \longrightarrow \kappa'} \qquad \frac{\Gamma \overset{F\omega}{\vdash} \tau_1 : \kappa' \longrightarrow \kappa \qquad \Gamma \overset{F\omega}{\vdash} \tau_2 : \kappa'}{\Gamma \overset{F\omega}{\vdash} \tau_1 \, \tau_2 : \kappa}$$

$$\frac{\Gamma, \varkappa \overset{F\omega}{\vdash} \tau : \kappa}{\Gamma \overset{F\omega}{\vdash} \Lambda \varkappa. \tau : \forall \varkappa. \kappa} \qquad \frac{\Gamma \overset{F\omega}{\vdash} \tau : \forall \varkappa. \kappa \qquad \Gamma \overset{F\omega}{\vdash} \kappa'}{\Gamma \overset{F\omega}{\vdash} \tau \, \kappa' : \kappa[\varkappa \mapsto \kappa']}$$

D.4 $\Gamma \overset{F\omega}{\vdash} e : \mathsf{u}$ – Term typing

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

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

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

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

F-Proj
$$\frac{\Gamma \overset{F\omega}{\vdash} e : \{\ell : \tau, \overline{\ell' : \tau'}\}}{\Gamma \overset{F\omega}{\vdash} e.\ell : \tau}$$

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

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

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

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

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

F-Pack
$$\frac{\begin{array}{c}\Gamma \overset{F\omega}{\vdash} \tau : \kappa \\ \Gamma \overset{F\omega}{\vdash} e : \tau'[\tau \mapsto \alpha] \qquad \Gamma \overset{F\omega}{\vdash} \exists^{\blacktriangledown} (\alpha : \kappa). \tau' : \star\end{array}}{\Gamma \overset{F\omega}{\vdash} \mathsf{pack}\ \langle \tau, e \rangle\ \mathsf{as}\ \exists^{\blacktriangledown} \alpha. \tau' : \exists^{\blacktriangledown} (\alpha : \kappa). \tau'}$$

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

D.5 Transparent existentials

F-Hide

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

F-Seal

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

F-Hidden

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

F-LiftArr

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

F-LiftAll

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

E ELABORATION RULES

E.1 Subtyping

E.1.1 *Signature subtyping*

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

E-Sub-Sig-Id

$$\frac{\Gamma \overset{\text{elab}}{\vdash} \mathcal{R} \prec \mathcal{R}' \rightsquigarrow f}{\Gamma \overset{\text{elab}}{\vdash} (\tau, \mathcal{R}) \prec \big(\tau, \mathcal{R}'\big) \rightsquigarrow \lambda x.\ \{\mathsf{id} = x.\mathsf{id}, \mathsf{Val} = f(x.\mathsf{Val})\}}$$

E-Sub-Sig-Struct

$$\frac{\overline{\mathcal{D}_0} \subseteq \overline{\mathcal{D}} \qquad \Gamma \overset{\text{elab}}{\vdash} \overline{\mathcal{D}_0} \prec \overline{\mathcal{D}'} \rightsquigarrow \overline{f} \qquad \overline{I'} = \mathsf{dom}(\overline{\mathcal{D}'})}{\Gamma \overset{\text{elab}}{\vdash} \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \prec \mathsf{sig}\ \overline{\mathcal{D}}'\ \mathsf{end} \rightsquigarrow \lambda x.\ \{\ell_{I'} = f\, x.\ell_{I'}\}}$$

E-Sub-Sig-GenFct

$$\frac{\Gamma, \overline{\alpha} \overset{\text{elab}}{\vdash} \mathcal{C} \prec \mathcal{C}'\big[\overline{\alpha}' \mapsto \overline{\tau}\big] \rightsquigarrow f}{\Gamma \overset{\text{elab}}{\vdash} () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \prec () \to \exists^{\blacktriangledown}\overline{\alpha}'.\mathcal{C}' \rightsquigarrow \lambda x.\lambda\_.\mathsf{unpack}\ \langle\overline{\alpha}, y\rangle = x()\ \mathsf{in}\ \mathsf{pack}\ \langle\overline{\tau}, f\,y\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\overline{\alpha}'.\mathcal{C}'}$$

E-Sub-Sig-AppFct

$$\frac{\Gamma, \overline{\alpha}' \overset{\text{elab}}{\vdash} \mathcal{C}_a' \prec \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow f \qquad \Gamma, \overline{\alpha}' \overset{\text{elab}}{\vdash} \mathcal{R}[\overline{\alpha} \mapsto \overline{\tau}] \prec \mathcal{R}' \rightsquigarrow g}{\Gamma \overset{\text{elab}}{\vdash} \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{R} \prec \forall\overline{\alpha}'.\mathcal{C}_a' \to \mathcal{R}' \rightsquigarrow \lambda x.\Lambda\overline{\alpha}'.\lambda y.\ g(x\,\overline{\tau}\,(f\,y))}$$

E.1.2 *Declaration subtyping*

$$\boxed{\Gamma \overset{\text{elab}}{\vdash} \mathcal{D} \prec \mathcal{D}' \rightsquigarrow f}$$

E-Sub-Decl-Val

$$\Gamma \overset{\text{elab}}{\vdash} (\mathsf{val}\ x : \tau) \prec (\mathsf{val}\ x : \tau) \rightsquigarrow \lambda x.x$$

E-Sub-Decl-Type

$$\Gamma \overset{\text{elab}}{\vdash} (\mathsf{type}\ t = \tau) \prec (\mathsf{type}\ t = \tau) \rightsquigarrow \lambda x.x$$

E-Sub-Decl-Mod

$$\frac{\Gamma \overset{\text{elab}}{\vdash} \mathcal{C} \prec \mathcal{C}' \rightsquigarrow f}{\Gamma \overset{\text{elab}}{\vdash} (\mathsf{module}\ X : \mathcal{C}) \prec (\mathsf{module}\ X : \mathcal{C}') \rightsquigarrow f}$$

E-Sub-Decl-ModType

$$\frac{\Gamma, \overline{\alpha} \overset{\text{elab}}{\vdash} \mathcal{C} \prec \mathcal{C}' \rightsquigarrow f \qquad \Gamma, \overline{\alpha} \overset{\text{elab}}{\vdash} \mathcal{C}' \prec \mathcal{C} \rightsquigarrow g}{\Gamma \overset{\text{elab}}{\vdash} (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}) \prec (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}') \rightsquigarrow \lambda x.x}$$

E.2 Typing

E.2.1 Module typing

$$\boxed{\Gamma \overset{\text{elab}}{\vdash} \mathtt{M} : \exists^{\blacktriangledown}\vartheta.\mathcal{C} \rightsquigarrow e}.$$

E-Typ-Mod-Arg

$$\frac{(Y : \mathcal{C}) \in \Gamma}{\Gamma \overset{\text{elab}}{\vdash} Y : \mathcal{C} \rightsquigarrow Y}$$

E-Typ-Mod-Var

$$\frac{(A.X : \text{ module } \mathcal{C}) \in \Gamma}{\Gamma \overset{\text{elab}}{\vdash} A.X : \mathcal{C} \rightsquigarrow A.X}$$

E-Typ-AppFct

$$\frac{\Gamma \overset{\text{elab}}{\vdash} \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a \overset{\text{elab}}{\vdash} \mathtt{M} : \exists^{\triangledown\overline{\tau}}(\overline{\beta}).\mathcal{C} \rightsquigarrow e}{\begin{array}{l}\Gamma \overset{\text{elab}}{\vdash} (Y : \mathsf{S}) \to \mathtt{M} : \exists^{\triangledown(),\overline{\lambda\overline{\alpha}.\tau}}(\alpha_0, \overline{\beta'}).\ \left(\alpha_0, \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]\right) \\ \qquad \rightsquigarrow \text{lift}^{\triangledown}\ \left\{\text{id} = e^{\triangledown}_{\text{id}}, \text{Val} = \text{lift}^{*}\ (\Lambda\overline{\alpha}.\lambda(Y : \mathcal{C}_a).e)\right\}\end{array}}$$

E-Typ-Mod-AppApp

$$\frac{\Gamma \overset{\text{elab}}{\vdash} P : (\_, \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}) \rightsquigarrow e \qquad \Gamma \overset{\text{elab}}{\vdash} P' : \mathcal{C}' \rightsquigarrow e' \qquad \Gamma \overset{\text{elab}}{\vdash} \mathcal{C}' \prec \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow f}{\Gamma \overset{\text{elab}}{\vdash} P(P') : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow (e.\text{Val})\ \overline{\tau}\ (f\ e')}$$

E-Typ-Mod-GenFct

$$\frac{\Gamma \overset{\text{elab}}{\vdash} \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow e}{\begin{array}{l}\Gamma \overset{\text{elab}}{\vdash} () \to \mathtt{M} : \exists^{\triangledown()}(\alpha_0).\ (\alpha_0, () \to \mathcal{C}) \\ \qquad \rightsquigarrow \text{lift}^{\triangledown}\ \left\{\text{id} = e^{\triangledown}_{\text{id}}, \text{Val} = \lambda(\_ : ()).e\right\}\end{array}}$$

E-Typ-Mod-GenApp

$$\frac{\Gamma \overset{\text{elab}}{\vdash} P : \left(\_, () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}\right) \rightsquigarrow e}{\Gamma \overset{\text{elab}}{\vdash} P() : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow e.\text{Val}()}$$

E-Typ-Mod-Ascr

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

E-Typ-Mod-Mode

$$\frac{\Gamma \overset{\text{elab}}{\vdash} \mathtt{M} : \exists^{\triangledown\overline{\tau}}(\overline{\alpha}).\mathcal{C} \rightsquigarrow e}{\Gamma \overset{\text{elab}}{\vdash} \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow \text{pack } \langle\overline{\tau}, e\rangle \text{ as } \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}$$

E-Typ-Mod-Struct

$$\frac{\Gamma \overset{\text{elab}}{\vdash}_A \bar{\mathsf{B}} : \exists^{\bar{\vartheta}}\bar{\alpha}.\overline{\mathcal{D}} \rightsquigarrow e \qquad A \notin \Gamma \qquad \Diamond = \text{mode}(\bar{\vartheta})}{\Gamma \overset{\text{elab}}{\vdash} \mathtt{struct}_A\ \bar{\mathsf{B}} \text{ end} : \exists^{\vartheta_0\bar{\vartheta}}\alpha_0\bar{\alpha}.\ (\alpha_0, \text{sig } \overline{\mathcal{D}} \text{ end}) \rightsquigarrow \text{lift}^{\Diamond}\ \left\{\text{id} = e^{\Diamond}_{\text{id}}, \text{Val} = e\right\}}$$

E-Typ-Mod-Proj

$$\frac{\Gamma \overset{\text{elab}}{\vdash} \mathtt{M} : \exists^{\bar{\vartheta}}\bar{\alpha}.\ (\_, \text{sig } \overline{\mathcal{D}} \text{ end}) \rightsquigarrow e \qquad \text{module } X : \mathcal{C} \in \overline{\mathcal{D}}}{\Gamma \overset{\text{elab}}{\vdash} \mathtt{M}.X : \exists^{\bar{\vartheta}}\bar{\alpha}.\mathcal{C} \rightsquigarrow \text{repack}^{\Diamond}\ \langle\overline{\alpha}, x\rangle = e \text{ in } e.\text{Val}.\ell_X}$$

E.2.2 Binding typing

$$\boxed{\Gamma \overset{\text{elab}}{\vdash}_A \mathsf{B} : \mathcal{D} \rightsquigarrow e}.$$

E-Typ-Bind-Let

$$\frac{\Gamma \overset{\text{elab}}{\vdash} \mathtt{e} : \tau \rightsquigarrow e}{\Gamma \overset{\text{elab}}{\vdash}_A (\mathtt{let}\ x = \mathtt{e}) : (\mathsf{val}\ x : \tau) \rightsquigarrow \{\ell_x = e\}}$$

E-Typ-Bind-Type

$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{u} : \tau}{\Gamma \overset{\text{elab}}{\vdash}_A (\mathtt{type}\ t = \mathsf{u}) : (\mathsf{type}\ t = \tau) \rightsquigarrow \{\ell_t = \langle\!\langle \tau \rangle\!\rangle\}}$$

E-Typ-Bind-ModType

$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \overset{\text{elab}}{\vdash}_A (\mathtt{module\ \ type}\ T = \mathsf{S}) : (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}) \rightsquigarrow \{\ell_T = \langle\!\langle \lambda\overline{\alpha}.\mathcal{C} \rangle\!\rangle\}}$$

E-Typ-Bind-Empty

$$\Gamma \overset{\text{elab}}{\vdash}_A \varnothing : \varnothing \rightsquigarrow \{\}$$

E-Typ-Bind-Mod

$$\frac{\Gamma \overset{\text{elab}}{\vdash}_A \mathsf{M} : \exists^{\bar{\vartheta}} \bar{\alpha}.\mathcal{C} \rightsquigarrow e}{\Gamma \overset{\text{elab}}{\vdash}_A (\mathtt{module}\ X = \mathsf{M}) : (\exists^{\bar{\vartheta}} \bar{\alpha}.\,\mathsf{module}\ X : \mathcal{C}) \rightsquigarrow \mathsf{repack}^{\diamond}\, \langle \overline{\alpha}, x \rangle = e\ \mathsf{in}\ \{\ell_X = x\}}$$

E-Typ-Seq

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