HAL open science

Retrofitting OCaml modules

Clément Blaudeau, Didier Rémy, Gabriel Radanne

To cite this version:

Clément Blaudeau, Didier Rémy, Gabriel Radanne. Retrofitting OCaml modules. JFLA 2023 - 34èmes Journées Francophones des Langages Applicatifs, Jan 2023, Praz-sur-Arly, France. pp.42-83. hal-03936636v1

HAL Id: hal-03936636

https://inria.hal.science/hal-03936636v1

Submitted on 12 Jan 2023 (v1), last revised 24 Jan 2023 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Retrofitting OCaml modules
Fixing signature avoidance in the generative case

Clément Blaudeau[1], Didier Remy[1], and Gabriel Radanne[2]

[1] Cambium, Inria, France
[2] CASH, Inria, EnsL, UCBL, CNRS, LIP, France

Abstract

ML modules are offer large-scale notions of composition and modularity. Provided as an additional layer on top of the core language, they have proven both vital to the working OCaml and SML programmers, and inspiring to other use-cases and languages. Unfortunately, their meta-theory remanins difficult to comprehend, requiring heavy machinery to prove their soundness. Building on a previous translation from ML modules to F^ω, we propose a new comprehensive description of a generative subset of OCaml modules, embarking on a journey right from the *source* OCaml module system, up to F^ω, and back. We pause in the middle to uncover a system, called *canonical* that combines the best of both worlds. On the way, we obtain type soundness, but also and more importantly, a deeper insight into the *signature avoidance problem*, along with ways to improve both the OCaml language and its typechecking algorithm.

1 A powerful but imperfect module system

Modularity is a key technique to break down a complex program into parts at different levels of abstraction. 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. Controlling the interactions between modules through a cautious choice of interfaces is not only a tool for correctness; using abstraction, it is also a way to enforce invariants and to allow for several teams of developers to work independently on different parts of the same program while enjoying a language-level guarantee that they respect the invariants of other parts.

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 *modules*, which form a 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 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 under development for 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, 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 (which corresponds to the distinction between *public* and *private* fields in Object-Oriented Programming), or by abstracting type components—keeping types accessible while hiding their definitions.

The OCaml module system is renowned for its expressiveness, ease of use, and the properties it can statically enforce. All sizable OCaml projects use modules to access libraries or define

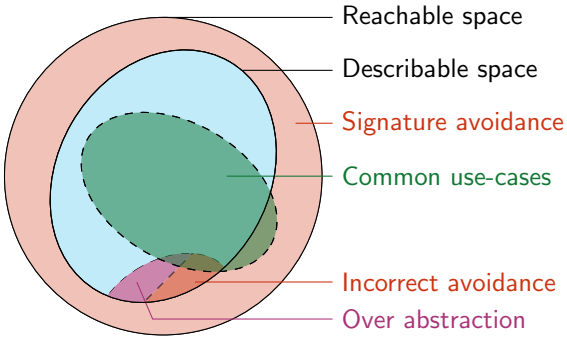


Figure 1: A representation of the mismatch between the *reachable space* of module expressions (outer-most circle) and the *describable space* of signatures (inner ellipse). The common use-cases of OCaml are mainly within the area where the typechecker behaves correctly. In some cases, the current OCaml 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).

parametric instances of data structures (sets, hashtables, streams, etc.). Several successful projects have made heavy use of modules, as in MirageOS [Madhavapeddy et al., 2013] where modules and functors are even assembled on demand using a DSL [Radanne et al., 2019]. Despite the successes and the interest of the community regarding ML modules, giving it 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 [White et al., 2014] where new modules could be built automatically from their signatures by applications of functors to other modules.

1.1 The signature avoidance problem

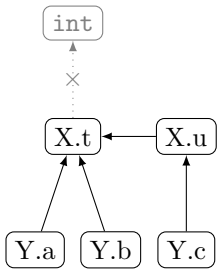
The *signature avoidance* problem is a key issue of ML-module systems. It originates from a mismatch, illustrated in Figure 1, between the expressiveness of the module and signature languages: the *reachable space* of possible module expressions is larger than the *describable space* of signatures: some 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 (or aliases of themselves). Second, sharing abstract types between modules, which is essential for module interactions, produces trees of type-alias equalities. Finally, hiding type components (through either explicit projections or implicit subtyping during functor applications) can remove abstract type aliases from scope, while other components (values or types) kept *in scope* may refer to them (for instance, removing an abstract type `t` while keeping a value of type `t list`). In some situations, there is no possible signature to infer for a module; in other situations, there are several incompatible ones.

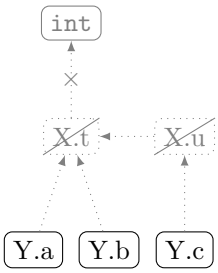
We illustrate this abstraction mechanism in Example 1. Each source code comes with a representation of the type equalities and aliases as a *tree of type equalities*. The connected components of the tree represent equal, interchangeable types that all belong to the same equivalence class. They can be rooted with base types (`int`, `bool`, `string`, etc.) or constructed over other types (like `int list`, `int` $\rightarrow$ `int`, `int` $\times$ `bool`, etc.) or previously defined *abstract types*. Abstracting a type effectively removes a link in the type-sharing tree, which splits a connected component.

Example 1 precisely illustrates this potential loss of type sharing when using module oper-

```
module type S = sig
  type t
  type u = t
end
module M = struct
  module X = (struct
    type t = int
    type u = t
  end : S)
  module Y : struct
    type a = X.t
    type b = X.t
    type c = X.u
  end
end
```




```
module type S = sig
  type t
  type u = t
end
module M = (struct
  module X = (struct
    type t = int
    type u = t
  end : S)
  module Y : struct
    type a = X.t
    type b = X.t
    type c = X.u
  end
end).Y
```



Example 1: Two examples of modules and associated type-sharing trees.

ations. On the left-hand side, restricting the signature of the module `X` to the module type `S` removes the link between `X.t` and `int` (grayed out in the type-sharing tree): this becomes hidden to the typechecker outside of the body of the module `X`. On the right-hand side, the projection on the submodule `Y` removes `X.t` and `X.u`. The types `Y.a`, `Y.b`, and `Y.c` are pointing to *out-of-scope* types: they must be changed or deleted. This shows why expressing membership to an equivalence class through paths is fragile: the removal of type fields can split connected components apart, resulting in a loss of type-sharing or incorrect signatures.

Strategies for solving signature avoidance When a type declaration is referring to an out-of-scope type, there are mainly three strategies to correct the signature: (1) abstracting the type (effectively ignoring the previous link in the tree), (2) rewriting the type equalities using in-scope aliases (effectively rewiring the tree to maintain connected components), or (3) extending the signature syntax with existential types. The first strategy can lead to loss of type sharing, 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 type-checker only tries to follow directed edges of the tree until it finds an accessible type, but does not follow reverse edges and thus does not have a notion of *connected components*. In Example 1, it would fail at finding that `Y.a`, `Y.b` and `Y.c` are aliases. 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 graph. The module syntax of OCaml actually encourages this approach by limiting the places where inlined, unnamed modules and signatures can be used. In particular, projection on an unnamed module (as done in Example 1) 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.

1.2 Related work

Previous work has provided solid formal foundations for ML-modules. The link between abstract types in ML-module systems and existential types in F^{ω} was already explored by Mitchell

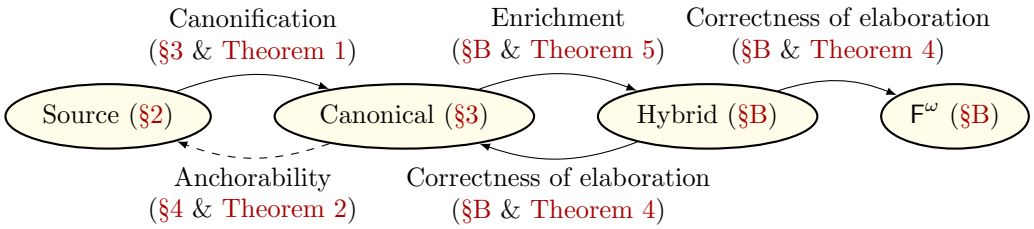


Figure 2: The different systems: their links, with theorems and sections.

and Plotkin [1985]. This vision was opposed by MacQueen [1986], who considered existential types to be too weak, and proposed using a restriction of dependent types (strong sums) to describe module systems. Further work, notably on phase separation by Harper et al. [1989], 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. [1989]. Two approaches for the formalization and improvement of abstract types in SML were later concomitantly described by Leroy [1994] (through manifest types) and Harper and Lillibridge [1994] (via an adapted F^ω with translucent sums). The OCaml module system itself was specified by Leroy [1994, 2000], and later with an extension to applicative functors [Leroy, 1995].

The use of existential types to interpret signatures by Russo [2004] opened the door for a simplified link between modules and F^ω. Type generativity is also explored by DREYER [2007], using stamps in place of existential types. A similar, but logically-based approach was later developed by Montagu and Rémy [2009] introducing the concept of open existential types. Pushing Russo's idea further, a closer correspondence between ML-modules and F^ω was achieved by Rossberg et al. [2014] by elaborating a significant subset of the SML module syntax into F^ω. The type system is safe by construction, inheriting the property from F^ω. A limit of this approach is that, since the definition of modules is only given by translation, the programmer must think in terms of the elaboration and only sees the elaborated types instead of the usual signatures. This makes direct reasoning on the source program more difficult. Their work, called the F-ing *approach*, also covered some difficult points like applicative functors and first-class modules, and was partially mechanized in Coq. Moving one step further, Rossberg [2018] achieved a unification of the core and module languages (thus, un-stratified) using F^ω as the underlying programming language and seeing module constructs as syntactic sugar. More recently, Crary [2020] used involved focusing techniques to solve the signature avoidance problem in the singleton-type approach (for SML modules) in a manner that turns out to have some similarities with that of *F-ing*.

1.3 A journey into OCaml modules

Our approach is strongly inspired by previous works, which we re-explore in the context of OCaml modules. Our goal is to provide a simpler and comprehensive type system for OCaml modules, which could first serve as a specification, then help correct and improve the current implementation, and, finally, serve as a basis for future extensions of the module language. Our contributions are summarized in Figure 2.

We start by presenting a (mostly standard) self-contained specification of the OCaml module system in §2, adapted from previous works. Critically, like the current OCaml implementation, this presentation suffers from the signature avoidance problem.

Then, we build on the insights of the *F-ing* [Rossberg et al., 2014] translation into F^ω to present a new set of typing rules. However, reasoning only *by elaboration* in F^ω is hard, as the elaboration introduces encoding layers and, crucially, uses a very different way of handling

abstract types, making the correspondence between source and F^ω terms and types rather obscure. Instead, as hinted in the work of Russo [2004], we introduce a light extension of the syntax for signatures with F^ω-style quantifiers, which we call *canonical signatures*. They act as a middle point between the path-based approach of the source and the quantified approach of F^ω, effectively splitting the translation effort in two steps. We introduce in §3 a new type system for OCaml modules, called *canonical*, where the source module expressions are typed with canonical signatures and source signatures translated into canonical ones. This system is simpler than the source one and doesn't suffer from the signature avoidance problem. Using our one-way translation, we revisit and explain the ad-hoc techniques (strengthening, equivalence) of the source presentation. We finally give the main result of this section: all source typing derivations can be translated into canonical typing derivations (Theorem 1).

The reverse translation of signatures (from the canonical system to the source one) is not always possible, as the former is more expressive than the latter. In §4 we explore *when* and *how* we can translate canonical signatures back into OCaml signatures, which uncovers precisely why the signature syntax lacks expressivity. This reverse process, called *anchoring*, is one of our main contributions. We then show that we can restrict the canonical system at one specific point to *mimic* the source system and only produce signatures that are valid regarding the source typing (Theorem 2). Some details are also given in the appendix §A.

The link between the canonical system and F^ω is conceptually easier while presenting some technical challenges. For lack of space, it is only detailed in the appendix §B. We introduce an *hybrid* system that produces both canonical and F^ω objects in correspondence. The hybrid system acts a central justification, as it can yield both the canonical system and the *F-ing* style elaboration by projections. The canonical system was actually designed by erasing the terms from the F^ω encoding which can still be seen as implicit *proof terms* justifying the canonical rules.

In this work, we restricted our system to a generative subset of the language. We believe our approach also extends to applicative functors and other features (first-class modules, module aliases, abstract signatures, etc.), which are left for future work. Our long term goal is to extend OCaml signatures following the canonical system, thus completely solving the signature avoidance problem. The canonical system would then be a standalone *source* system which can be used by programmers to reason about OCaml programs, with the F^ω elaboration ensuring its soundness.

2 A generative subset of the OCaml module system

In this section we present a module system that models a generative subset of OCaml. We present its grammar, several auxiliary judgments used, and end with its typing judgment.

2.1 Grammar and syntactic choices

Figure 3 gives the grammar for a generative subset of the module language. The language of modules is built on top of a core language of expressions e and types τ which we leave abstract, except for value identifiers x and type identifiers t, so that we can extend expressions with qualified variables and types with qualified types.

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 *uppercase letters*, M, X, etc. while lowercase letters are used for expressions

Path

$P ::= A.X$		(Direct access)
$\mid P.X$		(External Access)
$\mid Y$		(Functor parameter)
$Q ::= A \mid P$		(Prefix)

Module Expression

$M ::= P$		(Path)
$\mid M.X$		(Projection)
$\mid (P : S)$		(Sealing)
$\mid (Y : S) \to M$		(Functor)
$\mid P(P')$		(Application)
$\mid \texttt{struct}_A\ \overline{B}\ \texttt{end}$		(Structure)

Binding

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

Identifier

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

Signature

$S ::= Q.T$		(Module type)
$\mid (Y : S) \to S$		(Functor signature)
$\mid \texttt{sig}_A\ \overline{D}\ \texttt{end}$		(Structural signature)

Declarations

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

Environment

$\Gamma ::= \varnothing$		(Empty)
$\mid \Gamma, (Y : S)$		(Functor Argument)
$\mid \Gamma, (A.I : D)$		(Declaration)

Core language

$e ::= Q.x$		(Qualified variable)
$\mid \ldots$		(Other expression)
$\tau ::= Q.t$		(Qualified type)
$\mid \ldots$		(Other type)

Figure 3: Syntax of the module language

and types of the core language. Lists are written with an overhead bar: $\overline{D}$ is a list of D. In order to simplify the treatment of scoping and shadowing, we use *self-references*, ranged over by letter A, in both structures and signatures to refer to the current object; their binding occurrence appears as a subscript to the structure or signature they belong to, so that self-references can freely be renamed. *Abstract types* are specified as types pointing to themselves, e.g., $\mathsf{type}\, t = A.t$ where A is the self-reference of the current structure. In typing contexts Γ, identifiers, denoted by I, are prefixed by the self-reference A of the structure or signature they are meant to belong to. As an implicit convention, in entries $A.I : D$, the identifier extracted from D is I. We introduce prefixes, written with the letter Q, to range over either a path P or a self reference A. 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 also play the role of an external name.

Projections and accesses Several restrictions are built into the grammar: field accesses inside a module type are forbidden, since prefixes Q may not originate from a module type identifier T; and paths cannot contain functor applications, which makes the system fully *generative*. 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.

Omitted constructs In addition to restricting the system to be generative, we omit some constructs for the sake of simplicity: the include operator, explicit constraints S with type $t = \tau$

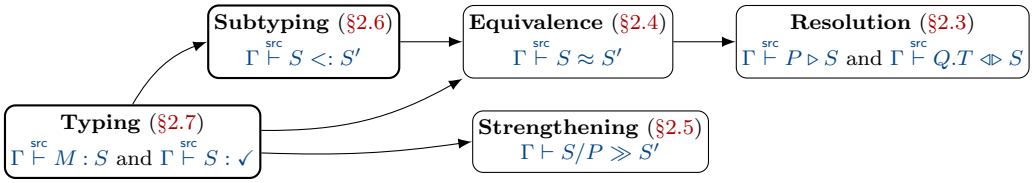


Figure 4: Structure of judgments for the source system

(resp. $\mathsf{module}\ X = P$) and deleting constraints $S\ \mathsf{with\ type}\ t := \tau$ (resp. $\mathsf{module}\ X := P$). We believe that they do not impact the overall structure of the system, only adding more cases in the set of rules. We did not include first-class modules.

2.2 System structure and judgments

The typing judgment uses several auxiliary judgments whose dependencies are presented in Figure 4. The system we present revolves around two main components: *typing* (for module expressions and signatures) and *subtyping*. The *path-based* representation of type sharing in OCaml modules requires us to define two more judgments: *equivalence* and *strengthening*. Finally, as retrieving signatures in the environment is non trivial, we define a pair of *resolution* helper judgments. We present and explain these judgments in reversed order of dependency in the subsequent subsections.

2.3 Resolution

In the OCaml module system, retrieving the signature of a given path (of a module stored in the environment) is not as simple as a lookup. First, objects are stored with multiple levels of indirection (like $X.X'.X''$) which requires to inspect intermediary signatures. Secondly, every signature can be a module type variables, which must also be resolved to its definition before accessing a field. For this purpose, we recursively define a *path resolution* judgment $\Gamma \overset{\text{src}}{\vdash} P \triangleright S$ which gives the signature S of a path P and a *signature resolution* judgment $\Gamma \overset{\text{src}}{\vdash} Q.T \triangleleft\!\triangleright S$, which retrieves the definition S of a module type $Q.T$ (both in an environment Γ).

We show below two examples of resolution rules. Rule S-Res-Modtype links path resolution and signature resolution: if the path resolution of P returns a module type $Q.T$, the definition of $Q.T$ should be resolved to a signature S. Rule S-Res-Proj-Mod accesses a submodule X of a module path P. This rule illustrates the transformation of *local* links in the signature S (that might refer to other components of the module at P via its self-reference A) into *absolute* ones by substitution of the self-reference A for the path P. The full set of rules is given in §C.1.

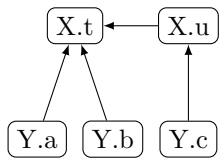
S-Res-Modtype
$$\frac{\Gamma \overset{\text{src}}{\vdash} P \triangleright Q.T \qquad \Gamma \overset{\text{src}}{\vdash} Q.T \triangleleft\!\triangleright S}{\Gamma \overset{\text{src}}{\vdash} P \triangleright S}$$

S-Res-Proj-Mod
$$\frac{\Gamma \overset{\text{src}}{\vdash} P \triangleright \mathtt{sig}_A\ \overline{D}\ \mathtt{end} \qquad (\mathsf{module}\ X : S) \in \overline{D}}{\Gamma \overset{\text{src}}{\vdash} P.X \triangleright S[A \mapsto P]}$$

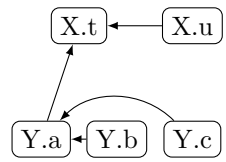
2.4 Equivalence

In the path-based approach of OCaml, several paths can resolve to the same type, as illustrated in §1.1. A type definition can have several *equivalent* expressions, called aliases. This leads us to consider a notion of *type equivalence* based on the existence of a common ancestor when browsing the type sharing tree of aliases. This can be seen in the rules S-Eqv-Type-Res and

```
module type S1 = sig
  module X : sig
    type t
    type u = t
  end
  module Y : sig
    type a = X.t
    type b = X.t
    type c = X.u
  end
end
```

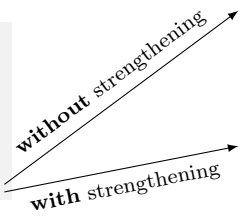



```
module type S2 = sig
  module X : sig
    type t
    type u = t
  end
  module Y : sig
    type a = X.t
    type b = a
    type c = a
  end
end
```

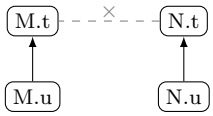


Example 2: Two equivalent signatures S_1 and S_2: they define two *different* type sharing trees that yet have the same connected components. Specifically, two types are aliases in S_1 if and only if they are also aliases in S_2.

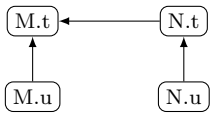
```
1 module M =
2   (struct ... end : sig
3   type t (* abstract *)
4   type u = t
5 end)
6
7 module N = M
```




```
module N : sig
  type t
  type u = t
end
```




```
module N : sig
  type t = M.t
  type u = t
end
```



Example 3: Strengthening and its graphical interpretation

S-Eqv-Type-Local: a type is equivalent to its definition (and only to itself for abstract types). With transitivity, symmetry, and reflexivity, we get an equivalence relationship on types.

S-Eqv-Type-Res

$$\frac{\Gamma \overset{\text{src}}{\vdash} P \triangleright \texttt{sig}_A\ \overline{D}\ \texttt{end} \qquad (\mathsf{type}\, t = \tau) \in \overline{D}}{\Gamma \overset{\text{src}}{\vdash} P.t \approx \tau[A \mapsto P]}$$

S-Eqv-Type-Local

$$\frac{(A.t : \mathsf{type}\, t = \tau) \in \Gamma}{\Gamma \overset{\text{src}}{\vdash} A.t \approx \tau}$$

Building on this type equivalence, we define *signature equivalence* : two signatures are equivalent if we can go from the first one to the second one by only substituting type aliases. This is illustrated by the two signatures in Example 2. The full set of rules is given in §C.2.

2.5 Strengthening

The need for strengthening is illustrated in Example 3. When aliasing the module `M` on line 7, the inferred signature for `N` cannot be straightforwardly obtained by copying the signature of `M` without loss of type-sharing. Indeed, `M` defines an abstract type which is a root of a connected component of the underlying type-sharing tree. Duplicating the signature duplicates the root, effectively creating a separated connected component and thus, a new abstract type. The known solution to this problem is to use a *strengthening* operation $\Gamma \vdash P/S \gg S'$ of a signature S by a path P, producing S', a *strengthened* version of S where each abstract type field has been rewritten into a concrete type field pointing to its original definition in P.

Rule S-Str-Sig-Sig shows the case for declarations inside a signature. Our technical choice of having abstract types represented as pointers to themselves ($\mathsf{type}\, t = A.t$) makes the definition of strengthening easy, as substituting the self-reference for the path suffices to transform local links into absolute ones ($\mathsf{type}\, t = P.t$). Thus, strengthening does not change declarations, except

for submodules (rule S-Str-Decl-Mod) where the signature is strengthened with the extended path $P.X$. Functor signatures are not strengthened, as they do not introduce new abstract types. The full set of rules is given in §C.3.

S-Str-Sig-Sig
$$\frac{\Gamma \vdash \overline{D}[A \mapsto P] \,/ P \gg \overline{D'}}{\Gamma \vdash \mathtt{sig}_A\ \overline{D}\ \mathtt{end} / P \gg \mathtt{sig}_A\ \overline{D'}\ \mathtt{end}}$$

S-Str-Decl-Mod
$$\frac{\Gamma \vdash S / (P.X) \gg S'}{\Gamma \vdash (\mathsf{module}\, X : S) \,/ P \gg \mathsf{module}\, X : S'}$$

2.6 Subtyping

The subtyping judgment has a fundamental role in the OCaml module system, as it allows the user to define an abstract (polymorphic) interface and assign it to a module that has a richer signature than this interface. In OCaml, some (but not all) subtyping operations have computational content: the removal and reordering of fields implies a runtime copy of the memory representation of the module. We thus distinguish two operations: *Subtyping by abstraction*, written $\Gamma \overset{\mathsf{src}}{\vdash} S_1 \lhd: S_2$, which has no computational content and *Subtyping*, written $\Gamma \overset{\mathsf{src}}{\vdash} S_1 <: S_2$, which includes the former and also allows for removal and reordering of fields and therefore has some computational content. Both judgments have the same rules, except for the comparison of structural signatures.

Subtyping on signatures enforces a structural match (up to name resolution): functor signature against functor signature and structural signature against structural signature. The latter case is where the only distinction between the two modes of subtyping appears: in subtyping by abstraction, the two lists of declarations $\overline{D}$ and $\overline{D'}$ are compared directly as shown in rule S-Sub-Sig-Sig. In the general form of subtyping, a subset $\overline{D}_0$ of the declaration list $\overline{D}$ is compared with $\overline{D'}$, allowing for reordering and deletion, as shown in Rule S-Sub-Sig. Using a subset instead of a subsequence allows for reordering.

S-SubEq-Sig-Sig
$$\frac{\Gamma, \overline{A.D} \overset{\mathsf{src}}{\vdash} \overline{D} \lhd: \overline{D'}}{\Gamma \overset{\mathsf{src}}{\vdash} \mathtt{sig}_A\ \overline{D}\ \mathtt{end} \lhd: \mathtt{sig}_A\ \overline{D'}\ \mathtt{end}}$$

S-Sub-Sig-Sig
$$\frac{\overline{D}_0 \subseteq \overline{D} \qquad \Gamma, \overline{A.D} \overset{\mathsf{src}}{\vdash}_A \overline{D_0} <: \overline{D'}}{\Gamma \overset{\mathsf{src}}{\vdash} \mathtt{sig}_A\ \overline{D}\ \mathtt{end} <: \mathtt{sig}_A\ \overline{D'}\ \mathtt{end}}$$

In both rules, the subtyping is done *declaration by declaration* (independently), in an environment that contains all the declarations of the richer (left-hand-side) signature. This approach matches well with our representation choice of abstract types, allowing us to have only one rule for comparing both abstract types and concrete types declarations (rule S-Sub-Decl-Type). The other subtyping rules for declarations are straightforward, and depends on signature subtyping. As a module type field can be used in both covariant and contravariant positions (as functor arguments for example), the rule for subtyping a module type declaration checks subtyping in both directions. In §C.4 and in the following, the rules that are similar between abstraction and general subtyping are given with the symbol $\prec:$ being either $<:$ or $\lhd:$

S-Sub-Decl-Type
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash} \tau \prec: \tau'}{\Gamma \overset{\mathsf{src}}{\vdash}_A (\mathsf{type}\, t = \tau) \prec: (\mathsf{type}\, t = \tau')}$$

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

2.7 Typing

Once all the auxiliary judgments have been laid out, the typing judgment is mostly straightforward. All rules are given in §C.5. The judgments for signature typing $\Gamma \overset{\mathsf{src}}{\vdash} S : \checkmark$ and declarations

typing $\Gamma \overset{\text{src}}{\vdash}_A D : \checkmark$ ensure that no shadowing occurs and no free variables are used. We omit the rules here. The judgment for bindings $\Gamma \overset{\text{src}}{\vdash} B : D$ uses a core-language typing judgment for expressions and typing of signatures, and is mutually recursive with the typing of modules (for submodules). The rules are straightforward and omitted.

Module typing The judgment for modules contains both syntax-directed and free-floating rules, making the presentation purposefully logical rather than algorithmic. There is an equivalent algorithmic presentation, but it is more involved and hides the key insights of the separation of operations between resolution, equivalence, strengthening, subtyping, and typing. Besides, the equivalent, canonical model that we propose in the next section is better suited for deriving algorithmic presentations for the source system. Free-floating typing rules must have no computational content. Therefore, general subtyping (which has some computation content), is not free floating: it should only be used at specific program points, namely functor application and signature ascription, where the target signature of the runtime coercion is explicit in the source. While having no computational content, subtyping by abstraction is still not made free-floating to prevent unnecessary abstraction of types (which would lead to loss of type-sharing). The syntax-directed rules are as follows:

S-Typ-Mod-Res
$$\frac{\Gamma \overset{\text{src}}{\vdash} P \triangleright S}{\Gamma \overset{\text{src}}{\vdash} P : S}$$

S-Typ-Mod-Sealing
$$\frac{\Gamma \overset{\text{src}}{\vdash} S : \checkmark \qquad \Gamma \overset{\text{src}}{\vdash} P : S' \qquad \Gamma \overset{\text{src}}{\vdash} S' <: S}{\Gamma \overset{\text{src}}{\vdash} (P : S) : S}$$

S-Typ-Mod-Struct
$$\frac{\Gamma \overset{\text{src}}{\vdash}_A \overline{B} : \overline{D} \qquad A \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash} \texttt{struct}_A\ \overline{B}\ \texttt{end} : \texttt{sig}_A\ \overline{D}\ \texttt{end}}$$

S-Typ-Mod-Functor
$$\frac{\Gamma \overset{\text{src}}{\vdash} S_a : \checkmark \qquad \Gamma; (Y : S_a) \overset{\text{src}}{\vdash} M : S \qquad Y \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash} (Y : S_a) \to M : (Y : S_a) \to S}$$

S-Typ-Mod-App
$$\frac{\Gamma \overset{\text{src}}{\vdash} P : (Y : S_a) \to S \qquad \Gamma \overset{\text{src}}{\vdash} P' : S'_a \qquad \Gamma \overset{\text{src}}{\vdash} S'_a <: S_a}{\Gamma \overset{\text{src}}{\vdash} P(P') : S\big[Y \mapsto P'\big]}$$

S-Typ-Mod-Proj
$$\frac{\Gamma \overset{\text{src}}{\vdash} M : \texttt{sig}_A\ \big(\overline{D_1}, \mathsf{module}\ X : S, \overline{D_2}\big)\ \texttt{end} \qquad \Gamma, \overline{D_1} \overset{\text{src}}{\vdash} S \lhd: S' \qquad \Gamma \overset{\text{src}}{\vdash} S' : \checkmark}{\Gamma \overset{\text{src}}{\vdash} M.X : S'}$$

Functor application (Rule S-Typ-Mod-App) is normally a place where *signature-avoidance* may occur. In our presentation however, the restriction of functor application to paths prevents this issue. Indeed, as all the abstract types of the argument are defined in the context, no type field is hidden by the application, and thus, no escaping of scope can occur. In contrast, as the grammar allows projection on any module expression (which can hide type fields and could trigger signature avoidance), the projection rule S-Typ-Mod-Proj is designed to tackle the issue. The abstraction subtyping $\Gamma, \overline{D_1} \overset{\text{src}}{\vdash} S \lhd: S'$ allows for cutting the links to types in S that will become inaccessible *outside* of the signature of M, i.e., without the fields declared in $\overline{D}_1$, so that the resulting signature S' is wellformed in Γ. This rule is however too permissive, as we allow to *implicitly* abstract *more than necessary* and lose some type sharing. Additional conditions to prevent *unnecessary* loss of type sharing could be added to this rule, but they would be complex to write down in this setting. Fortunately, they will be easy to express using the canonical system.

Free-Floating rules Before doing a projection $M.X$, we must allow to *rewire* the type sharing tree so that types used in X point to other aliases (either external ones, i.e., accessible outside

of M, or local ones, accessible inside of X). Similarly, we must allow strengthening to keep sharing information between aliases. This leads to two additional *free-floating* rules:

S-Typ-Strengthen
$$\frac{\Gamma \overset{\text{src}}{\vdash} P : S \qquad \Gamma \vdash S/P \gg S'}{\Gamma \overset{\text{src}}{\vdash} P : S'}$$

S-Typ-Equiv
$$\frac{\Gamma \overset{\text{src}}{\vdash} M : S \qquad \Gamma \overset{\text{src}}{\vdash} S \approx S'}{\Gamma \overset{\text{src}}{\vdash} M : S'}$$

Remarks on the source system While this declarative presentation of the source system may appear relatively simple, it actually suffers from three main issues. First, it relies on the equivalence judgment to *guess* correct rewritings in signatures. Second, type sharing can be lost when projecting on a submodule via over-abstraction. Thirdly, the strengthening judgment may seem ad-hoc, while being necessary to prevent loss of type-sharing. In current OCaml, these problems are partially solved via *heuristics* which are sufficient in simple cases, but fail for more advanced uses of modules. We now study a canonical presentation of the module system that solves all three issues.

3 Canonical signatures and existential types

Some issues of the source presentation come from the ambiguity and weaknesses of the path-based type-sharing. In this section, we present an alternative set of typing rules with an extended syntax for signatures inspired by previous work on representation of modules in F^ω. This system, called *canonical*, has simpler typing judgments and does not suffer from the signature avoidance problem. We show that the canonical system is more expressive than the source one, as every module expression that can be typed in the source system can also be typed in the canonical system with a signature that has at least the same amount of type sharing.

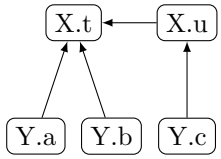
3.1 The need for existential types

In the source presentation, maintaining the connected components of the type sharing tree throughout projection —where fields can be deleted— sometimes require rewriting type equalities (through equivalence) or abstracting type fields (through abstraction subtyping). The core issue is that source signatures store type equalities via a type sharing tree that is cumbersome to handle. However, the information we are actually interested in is not the type sharing tree itself, only the connected components. To handle those directly, we introduce an *existential type* that acts as a canonical representative of the equivalence class. In the type sharing tree, it can be seen as an additional point with a special status, and the whole tree is flattened to one level of depth, as illustrated in Example 4. Rooted with an existential type that cannot be deleted, the connected components become insensible to the loss of some type fields. Building on this intuition, we present a new system, called *canonical*, where signatures use quantifiers to handle the abstract types.

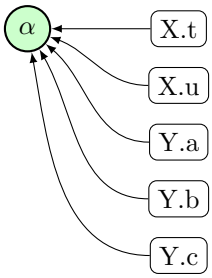
3.2 System structure and judgments

The grammar of the language, given in Figure 5, extends the OCaml grammar of Figure 3. Crucially, the source syntax (for modules and signatures) remains the same, including source signatures, while new syntactical categories are introduced to represent canonical signatures (and canonical types). By convention, we use curvy capitals ($\mathcal{R}$, $\mathcal{S}$, $\mathcal{D}$, ...) for canonical objects.

```
module M : sig
  module X : sig
    type t
    type u = t
  end
  module Y : struct
    type a = X.t
    type b = X.t
    type c = X.u
  end
end
```




```
module M : ∃α, sig
  module X : sig
    type t = α
    type u = α
  end
  module Y : struct
    type a = α
    type b = α
    type c = α
  end
end
```



Example 4: Comparison between the source and canonical signatures and their associated type-sharing trees. Canonical signatures are extended with an existential binder that acts as a special anchor point for the tree.

Canonical Types

$\tau ::= \alpha$ (Existential identifier)
$\quad \mid \ldots$ (Other types)

Environments

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

Canonical abstract signatures

$\mathcal{S} ::= \exists \overline{\alpha}.\mathcal{R}$ (Abstract signature)

Canonical manifest signatures

$\mathcal{R} ::= \forall \overline{\alpha}.\ (Y : \mathcal{R}) \to \mathcal{S}$ (Functor)
$\quad \mid \mathtt{sig}_A\ \overline{\mathcal{D}}\ \mathtt{end}$ (Signature)

Canonical declarations

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

Figure 5: Syntax extensions of the *canonical* module language.

We distinguish between *manifest* canonical signatures $\mathcal{R}$ that only refer to abstract types bound in the context and *abstract* canonical signatures $\exists \overline{\alpha}.\mathcal{R}$ (written $\mathcal{S}$) that also specify the existential types $\overline{\alpha}$ created by the module. Notice that canonical manifest signatures cannot refer to other signatures (e.g., using paths), as these are always inlined. Signatures of functors are polymorphic in the abstract types provided by the argument: indeed, these should be treated abstractly, while they can also be returned and thus shared in the result. Signatures stored in module type declarations are just parameterized by their abstract types: these will later be existentially or universally quantified, depending on context. We use an F^ω style λ-binding for these. As we will see in the typing rules, the grammar restricts the positions where abstract types are bounded (existentially or universally). For instance, inside a structural signature, all submodules have manifest signatures, while functor bodies may bound new abstract types.

Canonical signatures remove the need for resolution, equivalence, and strengthening, which significantly simplifies the overall presentation. The judgments are thus reduced to typing of modules (and bindings), typing of source signatures, which we enrich to produce a canonical signature, and subtyping.

3.3 Signature typing

We start with signature typing $\Gamma \overset{\text{can}}{\vdash} S : \lambda \overline{\alpha}.\mathcal{R}$ and declaration typing $\Gamma \overset{\text{can}}{\vdash}_A D : \lambda \overline{\alpha}.\mathcal{D}$, as they illustrate the key mechanism of *abstract type lifting*. The positions where abstract types are

bound are crucial to ensure the correct sharing of types. First, abstract types are introduced only by an abstract type declaration (Rule C-Typ-Decl-TypeAbs). Then, they are gathered and lifted from abstract type definitions and submodules to their parent module to ensure sharing between declarations in the same structure (rules C-Typ-Decl-Mod and C-Typ-Decl-Seq). The lifting does not go through a functor definition (Rule C-Typ-Sig-Functor): indeed, in the generative case, each application of the functor should create new unrelated abstract types. The full set of rules are in §D.2.

C-Typ-Decl-TypeAbs

$$\frac{A.t \notin \Gamma}{\Gamma \overset{\mathsf{can}}{\vdash}_A \mathsf{type}\, t = A.t : \lambda\alpha.\, \mathsf{type}\, t = \alpha}$$

C-Typ-Decl-Mod

$$\frac{\Gamma \overset{\mathsf{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{R} \qquad A.X \notin \Gamma}{\Gamma \overset{\mathsf{can}}{\vdash}_A (\mathsf{module}\, X : S) : \lambda\overline{\alpha}.\, (\mathsf{module}\, X : \mathcal{R})}$$

C-Typ-Decl-Seq

$$\frac{\Gamma \overset{\mathsf{can}}{\vdash}_A D_1 : \lambda\overline{\alpha}_1.\mathcal{D}_1 \qquad \Gamma, \overline{\alpha}_1, A.I_1 : \mathcal{D}_1 \overset{\mathsf{can}}{\vdash}_A \overline{\mathcal{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \overset{\mathsf{can}}{\vdash}_A (D_1, \overline{D}) : \lambda\overline{\alpha}_1\, \overline{\alpha}.\, (\mathcal{D}_1, \overline{\mathcal{D}})}$$

C-Typ-Sig-Functor

$$\frac{\Gamma \overset{\mathsf{can}}{\vdash} S_a : \lambda\overline{\alpha}.\mathcal{R}_a \qquad \Gamma, \overline{\alpha}, Y : \mathcal{R}_a \overset{\mathsf{can}}{\vdash} S : \lambda\overline{\beta}.\mathcal{R} \qquad Y \notin \Gamma}{\Gamma \overset{\mathsf{can}}{\vdash} (Y : S_a) \to S : \forall\overline{\alpha}.\, (Y : \mathcal{R}_a) \to \exists\overline{\beta}.\mathcal{R}}$$

3.4 Subtyping

The subtyping between two canonical signatures $\exists\overline{\alpha}.\mathcal{R}$ and $\exists\overline{\alpha}'.\mathcal{R}'$ is similar to the source subtyping, with the addition of the mechanism to deal with quantifiers.

Rule C-Sub-Sig-Match allows us to compare abstract signatures, while the other subtyping rules deal with concrete signatures. The types $\overline{\alpha}$ created by the left-hand-side signature are made available (i.e., pushed in the context), while the right-hand-side ones $\overline{\alpha}'$ are instantiated with some types $\overline{\tau}$. If both signatures define the same abstract types (up to α-conversion), a simple renaming instantiation $\overline{\alpha \mapsto \alpha'}$ is sufficient. Otherwise, the left-hand-side signature $\mathcal{R}$ can be *less abstract* (i.e., binding fewer abstract types) than the right-hand-side one $\mathcal{R}'$, in which case some right-hand side abstract types are instantiated with concrete types.

For functor signatures (Rule C-Sub-Sig-Functor), the parameter is in a contravariant position, hence we check subtyping of the argument signatures in reverse order. As with the source presentation, we distinguish between two kinds of subtyping: abstraction-only subtyping $\Gamma \overset{\mathsf{can}}{\vdash} \mathcal{S}_1 \lhd: \mathcal{S}_2$ and general subtyping $\Gamma \overset{\mathsf{can}}{\vdash} \mathcal{S}_1 <: \mathcal{S}_2$, which allows both abstraction and deletion of fields. They differ only by the rules C-Sub-Sig-Sig *vs.* C-SubEq-Sig-Sig. In §D.1 and in the following, the rules that are similar between abstraction and general subtyping are given with the symbol $\prec:$ being either $<:$ or $\lhd:$.

C-Sub-Sig-Match

$$\frac{\Gamma, \overline{\alpha} \overset{\mathsf{can}}{\vdash} \mathcal{R} \prec: \mathcal{R}'\left[\overline{\alpha}' \mapsto \overline{\tau}\right]}{\Gamma \overset{\mathsf{can}}{\vdash} \exists\overline{\alpha}.\mathcal{R} \prec: \exists\overline{\alpha}'.\mathcal{R}'}$$

C-Sub-Sig-Functor

$$\frac{\Gamma, \overline{\alpha}' \overset{\mathsf{can}}{\vdash} \mathcal{R}' \prec: \mathcal{R}[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma, \overline{\alpha}', \left(Y : \mathcal{R}'\right) \overset{\mathsf{can}}{\vdash} \mathcal{S}[\overline{\alpha} \mapsto \overline{\tau}] \prec: \mathcal{S}' \qquad Y \notin \Gamma}{\Gamma \overset{\mathsf{can}}{\vdash} \forall\overline{\alpha}.\, (Y : \mathcal{R}) \to \mathcal{S} \prec: \forall\overline{\alpha}'.\, \left(Y : \mathcal{R}'\right) \to \mathcal{S}'}$$

C-Sub-Sig-Sig

$$\frac{\overline{\mathcal{D}_0} \subseteq \overline{\mathcal{D}} \qquad \Gamma \overset{\mathsf{can}}{\vdash} \overline{\mathcal{D}_0} \prec: \overline{\mathcal{D}'}}{\Gamma \overset{\mathsf{can}}{\vdash} \mathtt{sig}_A\ \overline{\mathcal{D}}\ \mathtt{end} \prec: \mathtt{sig}_A\ \overline{\mathcal{D}'}\ \mathtt{end}}$$

C-SubEq-Sig-Sig

$$\frac{\Gamma \overset{\mathsf{can}}{\vdash} \overline{\mathcal{D}} \lhd: \overline{\mathcal{D}'}}{\Gamma \overset{\mathsf{can}}{\vdash} \mathtt{sig}_A\ \overline{\mathcal{D}}\ \mathtt{end} \lhd: \mathtt{sig}_A\ \overline{\mathcal{D}'}\ \mathtt{end}}$$

3.5 Typing

As with subtyping, the use of canonical signatures and existential types makes the typing simpler and syntax directed. The key technical point is the lifting of the existential quantification that

happens during typing, as already explained for the signature typing in §3.3.

Bindings typing The rules presented below for typing a binder give a list of canonical declarations *alongside* a list of (created) existential types. The signature typing judgment is used to convert source types and signatures (as in Rule C-Typ-Decl-ModType for instance) into canonical types and signatures.

C-Typ-Decl-Mod

$$\frac{\Gamma \overset{\text{can}}{\vdash} M : \exists\overline{\alpha}.\mathcal{R} \qquad A.X \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_A (\mathsf{module}\, X = M) : \exists\overline{\alpha}.(\mathsf{module}\, X : \mathcal{R})}$$

C-Typ-Decl-ModType

$$\frac{\Gamma \overset{\text{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{R} \qquad A.T \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_A (\mathsf{module\ type}\ T = S) : (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{R})}$$

C-Typ-Decl-Seq

$$\frac{\Gamma \overset{\text{can}}{\vdash}_A B_1 : \exists\overline{\alpha}_1.\mathcal{D}_1 \qquad \Gamma, \overline{\alpha}_1, A.I_1 : \mathcal{D}_1 \overset{\text{can}}{\vdash}_A \overline{B} : \exists\overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\vdash}_A B_1, \overline{B} : \exists\overline{\alpha}_1\overline{\alpha}.\mathcal{D}_1, \overline{\mathcal{D}}}$$

The existential types—which are lifted from submodules with Rule C-Typ-Decl-Mod—are merged by Rule C-Typ-Decl-Seq, to extend their scope to the local enclosing module. For example, if a module M has a submodule X which defines an abstract type α, the existential quantification is lifted from the submodule X to the whole module M.

Module expressions typing Existential types are introduced only by explicit sealing (Rule C-Typ-Mod-Sealing). Applying a functor to its argument (Rule C-Typ-Mod-App) can share the types of the argument via the substitution $[\overline{\alpha} \mapsto \overline{\tau}]$.

C-Typ-Struct

$$\frac{\Gamma \overset{\text{can}}{\vdash}_A B : \exists\overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash} \mathtt{struct}_A\ B\ \mathtt{end} : \exists\overline{\alpha}.\mathtt{sig}_A\ \overline{\mathcal{D}}\ \mathtt{end}}$$

C-Typ-Mod-Sealing

$$\frac{\Gamma \overset{\text{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{R} \qquad \Gamma \overset{\text{can}}{\vdash} P : \mathcal{R}' \qquad \Gamma \overset{\text{can}}{\vdash} \mathcal{R}' <: \mathcal{R}[\overline{\alpha} \mapsto \overline{\tau}]}{\Gamma \overset{\text{can}}{\vdash} (P : S) : \exists\overline{\alpha}.\mathcal{R}}$$

C-Typ-Mod-Functor

$$\frac{\Gamma \overset{\text{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{R} \qquad \Gamma, \overline{\alpha}, (Y : \mathcal{R}) \overset{\text{can}}{\vdash} M : \mathcal{S}}{\Gamma \overset{\text{can}}{\vdash} (Y : S) \to M : \forall\overline{\alpha}.\ (Y : \mathcal{R}) \to \mathcal{S}}$$

C-Typ-Mod-App

$$\frac{\Gamma \overset{\text{can}}{\vdash} P : \forall\overline{\alpha}.\ (Y : \mathcal{R}) \to \mathcal{S} \qquad \Gamma \overset{\text{can}}{\vdash} P' : \mathcal{R}' \qquad \Gamma \overset{\text{can}}{\vdash} \mathcal{R}' <: \mathcal{R}[\overline{\alpha} \mapsto \overline{\tau}]}{\Gamma \overset{\text{can}}{\vdash} P(P') : \mathcal{S}[\overline{\alpha} \mapsto \overline{\tau}]}$$

C-Typ-Mod-Proj

$$\frac{\Gamma \overset{\text{can}}{\vdash} M : \exists\overline{\alpha}.\mathtt{sig}_A\ \overline{\mathcal{D}}_1, \mathsf{module}\, X : \mathcal{R}, \overline{\mathcal{D}}_2\ \mathtt{end}}{\Gamma \overset{\text{can}}{\vdash} M.X : \exists\overline{\alpha}.\mathcal{R}}$$

In the source presentation, the projection rule S-Typ-Mod-Proj is the only place where signature avoidance can arise, by cutting local links in unsolvable ways. We needed to allow for abstraction subtyping. In the canonical presentation (Rule C-Typ-Mod-Proj), all the local links go through existentially quantified abstract types in front of the signature. By keeping all the existentially quantified types, projecting a component cannot cause signature avoidance: all the abstract types (that could go out of scope and cause signature avoidance issues) that $M.X$ could use are in the list $\overline{\alpha}$ and remain bound in the signature of $M.X$. The full set of typing rules is given in §D.2.

3.6 From the source to the canonical system

Both presentations share the same grammar for module expressions and signatures, and thus have the same input for typing. By extending the typing predicate to environments (to check wellformedness), we can translate source judgments (resolution, equivalence, strengthening) into the canonical presentation. This translation—called canonification—gives insight on how the source mechanisms can be understood in the canonical framework.

Typing of source environments We translate environments with source signatures into their canonical counterparts, using signature typing, declaration typing, and the following rules:

C-Cf-Empty
$$\varnothing \overset{\mathsf{can}}{\vdash} \varnothing$$

C-Cf-FctArg
$$\frac{\Gamma \overset{\mathsf{can}}{\vdash} \Gamma_{\mathsf{s}} \qquad \Gamma \overset{\mathsf{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{R}}{\Gamma, \overline{\alpha}, (Y : \mathcal{R}) \overset{\mathsf{can}}{\vdash} \Gamma_{\mathsf{s}}, (Y : S)}$$

C-Cf-Decl
$$\frac{\Gamma \overset{\mathsf{can}}{\vdash} \Gamma_{\mathsf{s}} \qquad \Gamma \overset{\mathsf{can}}{\vdash}_{A} D : \lambda\overline{\alpha}.\mathcal{D}}{\Gamma, \overline{\alpha}, (A.I : \mathcal{D}) \overset{\mathsf{can}}{\vdash} \Gamma_{\mathsf{s}}, (A.I : D)}$$

Canonification of resolution We now state how the resolution behaves regarding canonification. The main difference between the two systems comes from the fact that abstract types are quantified outside of the signature in the canonical system (the signature is always concrete), while they are introduced anywhere inside source signatures (which may contain abstract type definitions). Thus, the source signature obtained via resolution of P and the canonical signature of P only differ by the presence of quantifiers.

$$\Gamma_{\mathsf{s}} \overset{\mathsf{src}}{\vdash} P \triangleright S \;\wedge\; \Gamma \overset{\mathsf{can}}{\vdash} \Gamma_{\mathsf{s}} \;\wedge\; \Gamma \overset{\mathsf{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{R} \implies \Gamma \overset{\mathsf{can}}{\vdash} P : \mathcal{R} \quad (1)$$

$$\Gamma_{\mathsf{s}} \overset{\mathsf{src}}{\vdash} Q.T \triangleleft\!\triangleright S \;\wedge\; \Gamma \overset{\mathsf{can}}{\vdash} \Gamma_{\mathsf{s}} \;\wedge\; \Gamma \overset{\mathsf{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{R} \implies \Gamma \overset{\mathsf{can}}{\vdash} Q.T : \lambda\overline{\alpha}.\mathcal{R} \quad (2)$$

Canonification of strengthening Via resolution, the introduction of abstract types is re-exposed by the signature. To prevent an actual duplication of abstract types, the source presentation relies on strengthening. While the resolution of a path P corresponds to a canonical signature with existential types bound in front, a strengthened signature can be made fully concrete (no quantified types): all types refer to their original definition through the path P. In the canonical presentation, this is a *no-op*, as all type definitions already go through the proper quantified existential types in the context:

$$\Gamma_{\mathsf{s}} \overset{\mathsf{src}}{\vdash} P \triangleright S \;\wedge\; \Gamma_{\mathsf{s}} \vdash S/P \gg S' \;\wedge\; \Gamma \overset{\mathsf{can}}{\vdash} \Gamma_{\mathsf{s}} \;\wedge\; \Gamma \overset{\mathsf{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{R} \implies \Gamma \overset{\mathsf{can}}{\vdash} S' : \mathcal{R} \quad (3)$$

Canonification of equivalence As we described in the previous section, two equivalent signatures have the same connected components of their type sharing tree. In the canonical model, all connected components are flattened into a tree of depth 1, with an existentially quantified variable at the root. This means that two equivalent signatures actually have the same canonification, which matches the intuition (and naming) that canonical signatures are indeed *canonical*. We get the following result:

$$\Gamma_{\mathsf{s}} \overset{\mathsf{src}}{\vdash} S_1 \approx S_2 \;\wedge\; \Gamma \overset{\mathsf{can}}{\vdash} \Gamma_{\mathsf{s}} \;\wedge\; \Gamma \overset{\mathsf{can}}{\vdash} S_1 : \mathcal{S}_1 \;\wedge\; \Gamma \overset{\mathsf{can}}{\vdash} S_2 : \mathcal{S}_2 \implies \mathcal{S}_1 = \mathcal{S}_2 \quad (4)$$

Canonification of subtyping As the rules of subtyping in the source and canonical systems are very similar, the two judgments are also very similar. The only difference is that source subtyping can lose type equalities by over-abstraction. However, in the canonical system, with

Rule C-Sub-Sig-Match, the correspondence between the abstract types is dealt with at the signature level, with no abstraction done at the declaration level. We get the following result:

$$\Gamma_{\mathsf{s}} \overset{\mathsf{src}}{\vdash} S_1 <: S_2 \;\wedge\; \Gamma \overset{\mathsf{can}}{\vdash} \Gamma_{\mathsf{s}} \;\wedge\; \Gamma \overset{\mathsf{can}}{\vdash} S_1 : \mathcal{S}_1 \;\wedge\; \Gamma \overset{\mathsf{can}}{\vdash} S_2 : \mathcal{S}_2 \implies \Gamma \overset{\mathsf{can}}{\vdash} \mathcal{S}_1 <: \mathcal{S}_2 \tag{5}$$

Canonification of typing Once all canonification results have been laid out, the typing result becomes easy to write. All typing derivations can be translated into canonical ones, with the only difference being that some over-abstraction during projections and resolutions can lead to a source signature having lost some type sharing. This type-sharing loss, can be delayed to the last step of the derivation (combining the losses of all the intermediary signatures), which gives the following result:

Theorem 1 (Canonification of typing)**.** Considering any module expression M that admits a signature S in the source presentation, it also admits a corresponding canonical signature $\mathcal{S}$ with more type equalities:

$$\Gamma_{\mathsf{s}} \overset{\mathsf{src}}{\vdash} M : S \;\wedge\; \Gamma \overset{\mathsf{can}}{\vdash} \Gamma_{\mathsf{s}} \;\wedge\; \Gamma \overset{\mathsf{can}}{\vdash} S : \mathcal{S} \implies \Gamma \overset{\mathsf{can}}{\vdash} M : \mathcal{S}' \;\wedge\; \Gamma \overset{\mathsf{can}}{\vdash} \mathcal{S}' \lhd: \mathcal{S}$$

This link between the source and canonical system shows the expressiveness of the latter one, but is only half of the story. Exploring at which conditions the canonical derivations can be translated back into the source will uncover a new concept: anchorability.

4 Anchoring: back to the source system

In this section we study the process of translating typing derivations from the canonical system back into the source system, which we call *anchoring*. As the canonical system is more expressive than the source one, it is not always possible to translate signatures back into the source syntax. Still, we can precisely exhibit the conditions under which anchoring is possible. This gives a new specification of the source system, which could also be used to improve the way signature avoidance is actually resolved in OCaml.

The canonical system thus provides the intuition, formal guarantees, and algorithmic insights for the design of OCaml's modules mechanisms.

4.1 Quantification vs structural information

The key insight is the semantic difference in the source syntax between the declaration of a *concrete* type ($\mathsf{type}\, t = \tau$) and that of an *abstract* type ($\mathsf{type}\, t$). An abstract type declaration states the *introduction* of both a *new type* and of a *new field* that may later be used as a handle to refer to this abstract type, directly or indirectly via a path. For a signature used in a covariant position, an abstract type declaration effectively *creates* a new type (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 but without introducing a new type.

Canonical signatures *separate* the quantification information (existential, universal, or lambda binders) from the structural information (fields). Thus, it makes the scope of abstract types explicit, using the logical notion of binder, which is perfectly suited for that purpose. More importantly, it allows to refer to types that do not yet have a handle, while the source system cannot. This is illustrated in the following code :

```
module X = (struct
  module X0 = (... : sig type t end)
  module X1 = struct
    type u = X0.t * int
    type v = X0.t * bool
  end
end).X1
```

```
(* Canonical signature of X *)
module X : ∃α. sig
  type u = α * int
  type v = α * string
end
```

Here, a type definition (`t`) is lost when projecting only on the submodule `X1`. The source syntax *cannot* express the existence of a type that is the first component of both `u` and `v` without already having a handle to that type[1]: being forced to merge quantification and structural information limits the expressiveness. However, the module `X` can be given a canonical signature, where the two kinds of information are clearly separated.

This allows us to better describe the solvable and unsolvable cases of signature avoidance for the source presentation. In the *solvable* cases of signature avoidance, each use of an abstract type has an *in-scope* alias (that can serve as a handle) defined before the type is used. In such cases, the quantification and structural information coincide in a way expressible by the source syntax. In other cases, signature avoidance is unsolvable.

4.2 Anchoring of canonical signatures

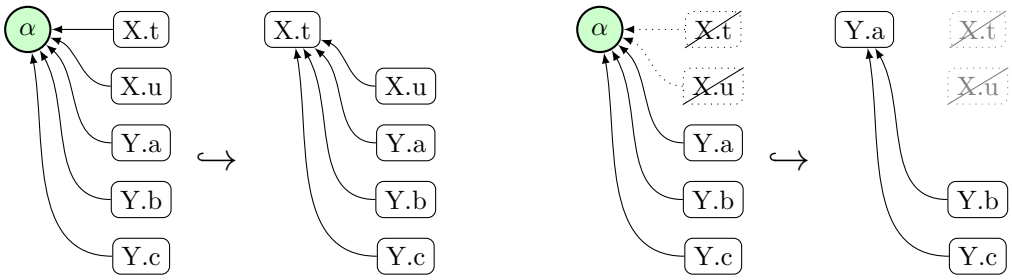
Building on the insights of the previous subsection, we can describe the *anchoring* process that translates a canonical signature back into the source system when the quantification and structural information coincide. The main mechanism boils down to identifying the first place where an abstract type α is used. If it is at the top level of a type declaration $\mathsf{type}\, t = \alpha$, we say that the existential type α is *anchorable*. In the corresponding source signature, we can make t an abstract type definition, creating a handle to t that can then be used instead of α in the canonical signature. The type t must be the *first* reference to the abstract type α, regardless of whether or not it is the original definition point. If the type α is used earlier, we raise a signature avoidance failure.

Example 5 illustrates the process of finding the first available position to rewire the type-sharing tree of Example 1. We see the change of the defining instance of α, from `X.t` to `Y.a` after a projection that makes `X.t` and `X.u` unreachable.

The anchoring process consists of associating abstract types with their first available aliases, which we gather using an *anchoring map* $\theta : \alpha \mapsto Q.t$. We define a signature anchoring judgment $\Gamma \overset{\text{anc}}{\vdash} \mathcal{S} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash S : \theta$ that, given an environment Γ and a canonical signature $\mathcal{S}$, produces a source environment Γ_{s} associated with an anchoring map θ_Γ and a source signature S associated with an anchoring map θ. In a nutshell, the judgment produces the source counterpart of the canonical signature $\mathcal{S}$ by gathering the handles of the abstract types, and replacing the uses of abstract types by their handles. The full set of rules is given in §D.3.

Using the anchoring judgment, we define *anchorable typing*, written $\Gamma \overset{\text{anc}}{\vdash} M : \mathcal{S}$. The judgment is defined by a copy of the typing rules for the judgment $\Gamma \overset{\text{can}}{\vdash} M : \mathcal{S}$ presented in §3.5, except for the projection rule which takes an additional premise checking for anchorability of the resulting signature. All anchorable typing derivations can be translated into source typing derivations.

[1] Adding a type field for `t` would not work, as it would not be an equivalent signature.



Canonical signature $\mathcal{S}_1$ Source signature S_1 Canonical signature $\mathcal{S}_2$ Source signature S_2

Example 5: Graphical representation of the anchoring process on the type sharing trees. In the two canonical signatures $\mathcal{S}_1$ and $\mathcal{S}_2$, the existential type α is anchorable. For $\mathcal{S}_1$, we can use `X.t` for the handle to α, as done in S_1. For $\mathcal{S}_2$, a projection removed `X.t` and `X.u`. The handle becomes `Y.a` as shown in S_2.

Theorem 2 (Anchorable typing)**.** Anchorable typing produces a signature that is valid regarding the source typing rules:

$$\Gamma \overset{\text{anc}}{\vdash} M : \mathcal{S} \quad \wedge \quad \Gamma \overset{\text{anc}}{\vdash} \mathcal{S} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash S : \theta \quad \Longrightarrow \quad \Gamma_{\mathsf{s}} \overset{\text{src}}{\vdash} M : S \tag{6}$$

More explanations and the formal definition of the anchoring process are given in §A. This completes the link between the source and canonical systems: canonical derivations can be translated back into the source when we restrict their expressiveness. The last link, namely the actual encoding in F^ω that inspired the design of the canonical system, is detailed in §B.

Conclusion

ML-Module systems are known for being a well-studied but complex topic. The path-based, OCaml approach has proven to be successful in practice, but has some structural issues. While being restricted to a generative subset, our study of the signature avoidance problem in the source presentation exposes the limitations of the current signature syntax. These limitations are at the heart of the need for *ad-hoc* and complex fixes (strengthening, equivalence). Building on previous works, we introduced the canonical system as a more expressive yet simpler language, equipped with the right construction (existential types) to separate quantification and structural information, and solve the main issues of the source system. Using the canonical system, we shined a new light on the ad-hoc techniques of the source presentation and provided a detailed description of the solvable and unsolvable cases of signature avoidance. While still being close to the OCaml source, the canonical presentation can easily be elaborated into F^ω, which provides formal guarantees. As a middle-point between usability and formalism, the canonical system is both a comprehensive description and a framework for building new features and improving the algorithms (specifically for the *solvable* cases of signature avoidance) of the current OCaml typechecker.

References

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.

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

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.

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.

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.

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.

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

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.

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.

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.

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.

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.

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

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.

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.

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.

A Anchoring

A.1 Anchoring of signatures

In this section we present in more details the mechanism that, under certain conditions, allows to build one of the source signatures corresponding to a given canonical signature. As mentioned in §4.2, the anchoring judgment, written

$$\Gamma \overset{\text{anc}}{\vdash} \mathcal{S} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash S : \theta$$

states that, given a canonical signature $\mathcal{S}$ and environment Γ, we can build a corresponding source signature S and source environment Γ_{s}. Both are returned with an *anchoring map* (θ for the signature and θ_Γ for the environment) that gives a path $Q.t$ for the handle of every abstract type α.

The anchoring map is extended when a type declaration with an *un-mapped* existential is encountered, which can be seen in rule C-Ach-Decl-AnchorPoint: the anchoring of the type declaration returns a non-empty map $\alpha \mapsto t$. In contrast, if the type field is concrete, an empty anchoring map is produced, as in Rule C-Ach-Decl-Type. Maps are merged together with the sequence rule C-Ach-Decl-Seq, via the merging operator $\uplus$. The map associated with the left-hand-side declaration is *prefixed* with the self-reference. More generally, we write $Q.\theta$ as a shortcut for the map $\alpha \mapsto Q.\theta(\alpha)$.

C-Ach-Decl-AnchorPoint

$$\frac{\Gamma \hookrightarrow \Gamma_{\mathsf{s}} : \theta_\Gamma \qquad \alpha \notin \mathrm{dom}(\theta_\Gamma)}{\Gamma \overset{\text{anc}}{\vdash}_A \mathsf{type}\, t = \alpha \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash \mathsf{type}\, t = A.t : (\alpha \mapsto t)}$$

C-Ach-Decl-Type

$$\frac{\Gamma \overset{\text{anc}}{\vdash} \tau \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash \tau' : \varnothing}{\Gamma \overset{\text{anc}}{\vdash}_A \mathsf{type}\, t = \tau \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash \mathsf{type}\, t = \tau' : \varnothing}$$

C-Ach-Decl-Seq

$$\frac{\Gamma \overset{\text{anc}}{\vdash} \mathcal{D} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash D : \theta \qquad \Gamma \overset{\text{anc}}{\vdash} \overline{\mathcal{D}} \hookrightarrow (\Gamma_{\mathsf{s}}, A.I : D)\,; (\theta_\Gamma \uplus A.\theta) \vdash \overline{D} : \theta'}{\Gamma \overset{\text{anc}}{\vdash}_A \mathcal{D}, \overline{\mathcal{D}} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash D, \overline{D} : \theta_\Gamma \uplus \theta'}$$

The judgment is easily extended to signatures and environments. Rule C-Ach-Env-Decl shows how the anchoring map associated with a single declaration is merged with the environment map.

C-Ach-Env-Decl

$$\frac{\Gamma \hookrightarrow \Gamma_{\mathsf{s}} : \theta_\Gamma \qquad \Gamma, \overline{\alpha} \overset{\text{anc}}{\vdash}_A \mathcal{D} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash D : \theta \qquad \mathrm{dom}(\theta) = \overline{\alpha}}{\Gamma, \overline{\alpha}, A.I : \mathcal{D} \hookrightarrow (\Gamma_{\mathsf{s}}, A.I : D) : \theta_\Gamma \uplus A.\theta}$$

A.2 Anchorable typing

As mentioned in §4.2, we can use the anchoring judgment to restrict the canonical typing to anchorable signatures. In our approach, the only place where a *non-anchorable* signature can be introduced is during projection on a submodule. Thus, we reuse all the rules of §3.5, except for the projection rule, which is modified to force the resulting signature to be anchorable:

C-TypA-Mod-Proj

$$\frac{\Gamma \overset{\text{anc}}{\vdash} M : \exists\overline{\alpha}.\mathtt{sig}_A\ \overline{\mathcal{D}}_1, \mathsf{module}\, X : \mathcal{R}, \overline{\mathcal{D}}_2\ \mathsf{end} \qquad \Gamma \overset{\text{anc}}{\vdash} \exists\overline{\alpha}.\mathcal{R} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash S : \theta}{\Gamma \overset{\text{anc}}{\vdash} M.X : \exists\overline{\alpha}.\mathcal{R}}$$

As expected, the anchorable typing, as a restriction of the canonical typing to *source-only* mechanisms, respects the source typing. More formally:

Theorem 3 (Anchorable typing)**.** Given any environment Γ, module M and (canonical) signature $\mathcal{S}$, the anchored signature S is a valid signature for the source typing:

$$\left.\begin{array}{l}\Gamma \overset{\text{anc}}{\vdash} M : \mathcal{S} \\ \Gamma \overset{\text{anc}}{\vdash} \mathcal{S} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash S : \theta\end{array}\right\} \Longrightarrow \Gamma_{\mathsf{s}} \overset{\text{src}}{\vdash} M : S \tag{7}$$

The proof of this theorem uses the following lemma :

Lemma 1 (Anchorability of subtyping)**.** Given Γ, Γ_{s}, σ_Γ, two canonical signatures $\mathcal{S}_1$ and $\mathcal{S}_2$, two source signatures S_1 and S_2, and two substitutions σ_1 and σ_2, we have:

$$\left.\begin{array}{l}\Gamma \overset{\text{anc}}{\vdash} \mathcal{S} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash S : \theta \\ \Gamma \overset{\text{anc}}{\vdash} \mathcal{S}' \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash S' : \theta' \\ \Gamma \overset{\text{can}}{\vdash} \mathcal{S} <: \mathcal{S}'\end{array}\right\} \Longrightarrow \Gamma_{\mathsf{s}} \overset{\text{src}}{\vdash} S <: S' \tag{8}$$

This completes the presentation of the link between the source and the canonical systems. As mentioned, the canonical system was designed as a simpler presentation of the mechanisms found in the F^ω encoding of modules. To complete this presentation, the next section gives in more details the encoding and the main techniques used, especially regarding the sharing of abstract types.

B F^ω and elaboration

To properly finish the transition from source to F^ω via the canonical system, we are missing one final link from the canonical system to F^ω. In this section we give formal foundations and guarantees for the canonical system through *elaboration* into F^ω: the elaboration is defined by *hybrid* judgments that relate module expressions and canonical signatures to terms and types of F^ω. This gives a precise correspondence between the key mechanisms of the canonical system, in particular existential types, and standard mechanisms of F^ω. The canonical system can actually be understood as a *particular mode of use* of F^ω. Thanks to F^ω, elaborated terms enjoy the properties of soundness, progress, and type preservation. In fact, the canonical system has been designed with the elaboration in mind, mainly by erasing proof terms and only retaining F^ω types. Hence, the elaboration is mostly straightforward.

Although, we use F^ω as the target language, we do not actually need the power of F^ω to model generative functors. The reason to pursue the development in F^ω is to also allow parameterized types in the source language and to be able to extend the encoding to the case of applicative functors in the future.

We start by giving the syntax and rules of the version of F^ω (extended with existential types and records) that we use for the elaboration and present the F^ω terms and types that encode modules and signatures. Then, we introduce the hybrid judgments. Finally, we state the correctness theorem for the elaboration and the link between the canonical system and F^ω.

This section is largely inspired by Rossberg et al. [2014]. We only made minor changes to the encoding of module expressions and signatures: the encoding of module types use λ-abstraction rather than existential quantification; declarations are encoded differently. The main difference is the extension of the judgments to produce *both* the F^ω type and the canonical signature, whereas in Rossberg et al. [2014], typing and subtyping are only defined by elaboration.

$$
\begin{array}{rll}
\kappa := & \Omega \mid \kappa \to \kappa & \text{(kinds)} \\
\rho := & \alpha \mid \rho \to \rho \mid \{\overline{\ell : \rho}\} \mid \forall \alpha : \kappa.\rho \mid \exists \alpha : \kappa.\rho \mid \lambda \alpha : \kappa.\rho \mid \rho\,\rho & \text{(types)} \\
E := & x \mid \lambda x : \rho.E \mid E\,E \mid \{\overline{\ell = E}\} \mid E.\ell \mid \lambda \alpha : \kappa.E \mid E\,\rho & \\
& \mid \mathsf{pack}\,\langle \rho, E\rangle_{\rho} \mid \mathsf{unpack}\,\langle \alpha, x\rangle = E \,\mathsf{in}\, E & \text{(terms)} \\
W := & \lambda x : \rho.E \mid \{\overline{\ell = W}\} \mid \lambda \alpha : \kappa.E \mid \mathsf{pack}\,\langle \rho, W\rangle_{\rho} & \text{(values)} \\
\Gamma := & \cdot \mid \Gamma, \alpha : \kappa \mid \Gamma, x : \rho & \text{(environments)}
\end{array}
$$

Figure 6: Syntax of F^{ω}

Abstract signature

$\Pi := \exists \overline{\alpha}.\Sigma$

Concrete signature

$\Sigma := \forall \overline{\alpha}.\Sigma \to \Pi$ (Functor)

$\mid \{\overline{\zeta}\}$ (Signature)

Declaration (with syntactic sugar)

$$
\begin{array}{rlcll}
\zeta := & \mathsf{val}\, x : \rho & \triangleq & \ell_x : \rho & \text{(Value)} \\
\mid & \mathsf{type}\, t = \rho & \triangleq & \ell_t : \langle\!\langle \rho \rangle\!\rangle & \text{(Type)} \\
\mid & \mathsf{module}\, X : \Sigma & \triangleq & \ell_X : \Sigma & \text{(Module)} \\
\mid & \mathsf{module\ type}\ T = \lambda \overline{\alpha}.\Sigma & \triangleq & \ell_T : \langle\!\langle \lambda \overline{\alpha}.\Sigma \rangle\!\rangle & \text{(Module type)}
\end{array}
$$

Figure 7: Syntactic categories and encoding

B.1 F-omega and encoded signatures

We use a standard variant of explicitly typed F^{ω} with primitive records and existential types; We only give its syntax in Figure 6; its typing rules are available in §E. We use letters ρ and E to range over types and expressions so as to distinguish them from the core language types and expressions, τ and e, although these should be seen as a subset of ρ and E. While kinds are part of terms, we usually omit them in F^{ω} terms for the sake of conciseness and readability.

To help with the elaboration, we assume a collection ℓ_I of record labels indexed by identifiers of the canonical (and source) system. The encoding of the canonical system into F^{ω} is described in Figure 7. Functors are encoded as functions and structural signatures as records. Declarations are encoded as record entries: we use the category of the identifier to distinguish between the encoding of values, types, modules and module types. We also introduce some syntactic sugar to have the encoding of declarations look similar to the usual canonical and source syntax. We use the following (overloaded) operator to shorten the encoding of type and module-type fields, with the type encoding on the left and the corresponding term on the right:

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

This allows to represent *type* fields as F^{ω} *terms*, specifically record fields: the term is always the identity and only its type encoding in the annotation matters, namely to enforce typing constraints. Indeed, erasing type fields during elaboration would have been possible, but this would prevent the elaboration from distinguishing canonical terms that would only differ from their type fields. This will ensure that two modules that differ only from their type definitions (i.e., their type fields) will also differ in their signatures. The type is generalized with a type operator β to allow the encoding of higher-kinded types.

A representative example of the encoding of a simple module is given in Example 6. The encoded signature is a type of F^{ω} that can be read as a canonical signature. Using the syntactic sugar makes the similarity with the source type even more striking. In the evidence term, we can see that the sealing of `int` corresponds to a $\mathsf{pack}\langle \mathsf{int}, \ldots \rangle$ term. To share the abstract type between the two fields (`X` and `u`), the abstract type is unpacked and then repacked, using a

pattern of the form $\mathsf{unpack}\langle\overline{\alpha},\cdot\rangle = \cdot \text{ in } \mathsf{pack}\langle\overline{\alpha},\cdot\rangle$ called a *repacking* pattern.

```
module M = struct
  module X = (struct
    type t = int
    let v = 42
  end : sig
    type t
    val v : t
  end)
  type u = X.t*bool
end
```

Encoded signature

$$\Pi \;=\; \exists\alpha \left\{ \begin{array}{l} \ell_X : \left\{ \begin{array}{l} \ell_t : \langle\!\langle \alpha \rangle\!\rangle \\ \ell_v : \alpha \end{array} \right\} \\ \ell_u : \langle\!\langle \alpha \times \mathtt{bool} \rangle\!\rangle \end{array} \right\} \triangleq \exists\alpha \left\{ \begin{array}{l} \mathsf{module}\, X : \left\{ \begin{array}{l} \mathsf{type}\, t = \alpha \\ \mathsf{val}\, v : \alpha \end{array} \right\} \\ \mathsf{type}\, u = \alpha \times \mathtt{bool} \end{array} \right\}$$

Encoded module

$$\begin{array}{c} E = \mathsf{unpack}\langle\alpha_1, y_1\rangle = \mathsf{pack}\,\langle \mathtt{int}, \{\ell_X = \{\ell_t = \langle\!\langle \mathtt{int} \rangle\!\rangle, \ell_v = 42\}\}\rangle \,\mathsf{in} \\ \mathsf{unpack}\langle\varnothing, y_2\rangle = \{\ell_u = \langle\!\langle (\alpha_1 \times \mathtt{bool}) \rangle\!\rangle\} \,\mathsf{in} \\ \mathsf{pack}\,\langle \alpha_1, \{\ell_X = (y_1.\ell_X), \ell_u = (y_2.\ell_u)\}\rangle \end{array}$$

Example 6: A simple module with two components (a submodule X and a type definition u). Its signature is encoded into a type Π of F^ω. A term E (called an *evidence term*) represents the module and has type Π.

B.2 An extension of the canonical system

The *hybrid* system can be seen as an *extension* of the canonical one: the judgments are extended to actually present both the canonical and F^ω parts. We define all judgments over an extended, hybrid environment Δ that contains both canonical and F^ω terms, defined as follows:

$$\Delta ::= \varnothing \mid \Delta, \overline{\alpha} \mid \Delta, (Y : (\mathcal{R}, \Sigma)) \mid \Delta, (A.I : (\mathcal{D}, \zeta))$$

The elaborated judgments rely on a core-language translation into F^ω for terms and types, that is omitted here.

B.2.1 Subtyping

We define the subtyping judgment $\Delta \overset{\text{elab}}{\vdash} \mathcal{S}_1 <: \mathcal{S}_2 \rightsquigarrow \Pi_1 <: \Pi_2 \Rightarrow f$, which follows closely the canonical one, as the handling of canonical signatures and F^ω types are similar. We use a version of F^ω with explicit subtyping, so the judgment is extended to produce a *subtyping function* $f : \Pi_1 \to \Pi_2$. The nature of this subtyping function gives interesting insights on the compilation of modules. For abstraction-only subtyping, the subtyping function is $\beta\eta$-convertible to the identity as the memory representation of modules (the terms) are the same when stripped of types. For general subtyping (with reordering or deletion of fields), the subtyping function is not *code-free*; correspondingly, it requires a specific action from the compiler (copy, access table, etc.).

The rule for functor subtyping E-Sub-Sig-Functor illustrates the main mechanisms: building on the subtyping functions (and type matching $\overline{\tau}$) for the parameter and the body, we craft a subtyping function from the left-hand-side functor signature to the right-hand-side one. All

the rules are given in §F.1.

E-Sub-Sig-Functor

$$\frac{\Delta, \overline{\alpha}' \overset{\text{elab}}{\vdash} \mathcal{R}'_a <: \mathcal{R}_a[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow \Sigma'_a <: \Sigma_a[\overline{\tau} \mapsto \overline{\alpha}] \Rightarrow f_a \qquad \Delta, \overline{\alpha}', (Y : (\mathcal{R}'_a, \Sigma'_a)) \overset{\text{elab}}{\vdash} \mathcal{S}[\overline{\alpha} \mapsto \overline{\tau}] <: \mathcal{S}' \rightsquigarrow \Pi[\overline{\alpha} \mapsto \overline{\tau}] <: \Pi' \Rightarrow f \qquad Y \notin \Delta}{\Delta \overset{\text{elab}}{\vdash} \forall\overline{\alpha}.\ (Y : \mathcal{R}_a) \to \mathcal{S} <: \forall\overline{\alpha}'.\ (Y : \mathcal{R}'_a) \to \mathcal{S}' \rightsquigarrow \forall\overline{\alpha}.\ \Sigma_a \to \Pi <: \forall\overline{\alpha}'.\ \Sigma'_a \to \Pi' \quad [\lambda\,(g : (\forall\overline{\alpha}.\ \Sigma_a \to \Pi))\,.\,\Lambda\overline{\alpha}'.\lambda\,(x : \Sigma'_a)\,.\,f\ (g\ \overline{\tau}\ (f_a\ x))]}$$

B.2.2 Typing

As for subtyping, the typing rules extend the canonical ones and produce an F^ω type (and term for module and bindings). When handling signatures and types, the lifting of existentials is just a rewriting. Producing the actual proof terms shows how the extension of scope is made using the repacking pattern. All the rules are given in §F.2.

Binding typing rules A binder is encoded with a single-field declaration record. The extrusion of existential types is illustrated in Rule E-Typ-Decl-Mod. In the sequence rule E-Typ-Decl-Seq, two bindings are merged using a let-binding for renaming. The extension of scope needed to make the abstract types of the first declaration $\overline{\alpha}_1$ available to the rest also uses the repacking pattern.

E-Typ-Decl-Mod

$$\frac{\Delta \overset{\text{elab}}{\vdash} M : \exists\overline{\alpha}.\mathcal{R} \rightsquigarrow e : \exists\overline{\alpha}.\Sigma \qquad A.X \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_A (\mathsf{module}\, X = M) : (\exists\overline{\alpha}.\, \mathsf{module}\, X : \mathcal{R}) \rightsquigarrow \mathsf{unpack}\langle\overline{\alpha}, y\rangle = e \text{ in } \mathsf{pack}\langle\overline{\alpha}, \{\ell_X = y\}\rangle : (\exists\overline{\alpha}.\, \mathsf{module}\, X : \Sigma)}$$

E-Typ-Decl-Seq

$$\frac{\Delta \overset{\text{elab}}{\vdash}_A B_1 : \exists\overline{\alpha}_1.\mathcal{D}_1 \rightsquigarrow e_1 : \exists\overline{\alpha}_1.\,\{\zeta_1\} \qquad \Delta, \overline{\alpha}_1, A.I_1 : (\mathcal{D}_1, \zeta_1) \overset{\text{elab}}{\vdash}_A \overline{B} : \exists\overline{\alpha}.\overline{\mathcal{D}} \rightsquigarrow e : \exists\overline{\alpha}.\,\{\overline{\zeta}\}}{\Delta \overset{\text{elab}}{\vdash}_A B_1, \overline{B} : \exists\overline{\alpha}_1\overline{\alpha}.\overline{\mathcal{D}_1}, \overline{\mathcal{D}} \rightsquigarrow \begin{array}{l} \mathsf{unpack}\langle\overline{\alpha}_1, x_1\rangle = e_1 \text{ in} \\ \mathsf{unpack}\langle\overline{\alpha}, x\rangle = (\mathsf{let}\, A.I_1 = x_1.\ell_{I_1} \text{ in } e) \text{ in} \\ \mathsf{pack}\langle\overline{\alpha}_1\overline{\alpha}, \{\ell_I = x_1.\ell_I, \overline{\ell_I = x.\ell_I}\}\rangle \\ : \exists\overline{\alpha}_1\overline{\alpha}.\,\{\zeta_1, \overline{\zeta}\} \end{array}}$$

Module expressions typing We only present the key typing rules for module expressions. The introduction of abstract types is done by sealing (E-Typ-Mod-Sealing), and features an explicit isolated *packing* (as opposed to a repacking). The application of functor (E-Typ-Mod-App) shows how the polymorphic interface (universally quantified) is instantiated with types (obtained from the matching of the argument's and parameter's signatures). Finally, the projection rule (E-Typ-Mod-Proj) illustrates another usage of the repacking pattern to collect

abstract types, project on the X submodule, and put back the abstract types.

E-Typ-Mod-Sealing

$$\frac{\Delta \overset{\text{elab}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{R} \rightsquigarrow \lambda\overline{\alpha}.\Sigma \qquad \Delta \overset{\text{elab}}{\vdash} P : \mathcal{R}' \rightsquigarrow e : \Sigma' \qquad \Delta \overset{\text{elab}}{\vdash} \mathcal{R}' <: \mathcal{R}[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow \Sigma' <: \Sigma[\overline{\alpha} \mapsto \overline{\tau}] \Rightarrow f}{\Delta \overset{\text{elab}}{\vdash} (P : S) : \exists\overline{\alpha}.\mathcal{R} \rightsquigarrow \mathsf{pack}\langle\overline{\tau}, (f\ e)\rangle : \lambda\overline{\alpha}.\Sigma}$$

E-Typ-Mod-App

$$\frac{\Delta \overset{\text{elab}}{\vdash} P' : \mathcal{R}' \rightsquigarrow e' : \Sigma' \qquad \Delta \overset{\text{elab}}{\vdash} P : \forall\overline{\alpha}.\ (Y : \mathcal{R}) \to \mathcal{S} \rightsquigarrow e : \forall\overline{\alpha}.\Sigma \to \mathcal{S} \qquad \Delta \overset{\text{elab}}{\vdash} \mathcal{R}' <: \mathcal{R}[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow \Sigma' <: \Sigma[\overline{\alpha} \mapsto \overline{\tau}] \Rightarrow f}{\Delta \overset{\text{elab}}{\vdash} P(P') : \mathcal{R}[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow e\ \overline{\tau}\ (f\ e') : \Pi[\overline{\alpha} \mapsto \overline{\tau}]}$$

E-Typ-Mod-Proj

$$\frac{\Delta \overset{\text{elab}}{\vdash} M : \exists\overline{\alpha}.\mathtt{sig}_A\ \overline{\mathcal{D}}_1, \mathsf{module}\ X : \mathcal{R}, \overline{\mathcal{D}}_2\ \mathtt{end} \rightsquigarrow e : \{\overline{\zeta_1}, \ell_X : \Sigma, \overline{\zeta_2}\}}{\Delta \overset{\text{elab}}{\vdash} M.X : \exists\overline{\alpha}.\mathcal{R} \rightsquigarrow \mathsf{unpack}\langle\overline{\alpha}, y\rangle = e\ \mathsf{in}\ \mathsf{pack}\langle\overline{\alpha}, e.\ell_X\rangle : \exists\overline{\alpha}.\Sigma}$$

B.3 Correctness and link with the canonical system

We write Δ^{can} and Δ^{F} for the left and right projections of the hybrid context Δ, defined in the obvious way. We present implications as rules.

Theorem 4 (Correctness of elaboration)**.** The elaboration judgment ensures well-typedness on both sides. For subtyping, we have the following result:

$$\Delta \overset{\text{elab}}{\vdash} \mathcal{S} <: \mathcal{S}' \rightsquigarrow \Pi <: \Pi' \Rightarrow f \implies \begin{cases} \Delta^{\mathsf{F}} \overset{\text{can}}{\vdash} \mathcal{S} <: \mathcal{S}' \\ \Delta^{\mathsf{F}} \vdash f : \Pi \to \Pi' \end{cases} \tag{9}$$

For typing, we have the following result:

$$\Delta \overset{\text{elab}}{\vdash} M : \mathcal{S} \rightsquigarrow e : \Pi \implies \begin{cases} \Delta^{\mathsf{can}} \overset{\text{can}}{\vdash} M : \mathcal{S} \\ \Delta^{\mathsf{F}} \vdash e : \Pi \end{cases} \tag{10}$$

Conversely, we show that well-typedness in the canonical system ensures that elaboration is well-defined. We first observe that there is an isomorphism between canonical and F^ω-encoded signatures. This allows us, for a given signature $\mathcal{S}$ (resp. $\mathcal{R}$) to consider its encoding in F^ω, noted $\Pi_{\mathcal{S}}$ (resp. $\Sigma_{\mathcal{R}}$). Similarly, we may consider the extension of an environment Γ into a hybrid enriched environment Δ_Γ (we omit the definition.)

Theorem 5 (Enrichment of canonical judgments)**.** The F^ω terms (e, f, $\Pi_{\mathcal{S}}$) corresponding to canonical derivation always exist. More formally, we have:

$$\Gamma \overset{\text{can}}{\vdash} S : \mathcal{S} \implies \Delta_\Gamma \overset{\text{elab}}{\vdash} S : \mathcal{S} \rightsquigarrow \Pi_{\mathcal{S}} \tag{11}$$

$$\Gamma \overset{\text{can}}{\vdash} \mathcal{S}_1 <: \mathcal{S}_2 \implies \exists f,\ \Delta_\Gamma \overset{\text{elab}}{\vdash} \mathcal{S}_1 <: \mathcal{S}_2 \rightsquigarrow \Pi_{\mathcal{S}_1} <: \Pi_{\mathcal{S}_2} \Rightarrow f \tag{12}$$

$$\Gamma \overset{\text{can}}{\vdash} M : \mathcal{S} \implies \exists e,\ \Delta_\Gamma \overset{\text{elab}}{\vdash} M : \mathcal{S} \rightsquigarrow e : \Pi_{\mathcal{S}} \tag{13}$$

C Source system – Complete rules

C.1 Resolution

C.1.1 $\Gamma \overset{\text{src}}{\vdash} P \triangleright S$ – Path resolution

S-Res-LocalMod
$$\frac{(A.X : S) \in \Gamma}{\Gamma \overset{\text{src}}{\vdash} A.X \triangleright S}$$

S-Res-FctArg
$$\frac{(Y : S) \in \Gamma}{\Gamma \overset{\text{src}}{\vdash} Y \triangleright S}$$

S-Res-Proj-Mod
$$\frac{\Gamma \overset{\text{src}}{\vdash} P \triangleright \texttt{sig}_A\ \overline{D}\ \texttt{end} \qquad (\mathsf{module}\, X : S) \in \overline{D}}{\Gamma \overset{\text{src}}{\vdash} P.X \triangleright S[A \mapsto P]}$$

S-Res-Modtype
$$\frac{\Gamma \overset{\text{src}}{\vdash} P \triangleright Q.T \qquad \Gamma \overset{\text{src}}{\vdash} Q.T \lhd\!\rhd S}{\Gamma \overset{\text{src}}{\vdash} P \triangleright S}$$

C.1.2 $\Gamma \overset{\text{src}}{\vdash} S \lhd\!\rhd S'$ – Signature resolution

S-Res-LocalSig
$$\frac{(A.T : S) \in \Gamma}{\Gamma \overset{\text{src}}{\vdash} A.T \lhd\!\rhd S}$$

S-Res-Proj-ModType
$$\frac{\Gamma \overset{\text{src}}{\vdash} P \triangleright \texttt{sig}_A\ \overline{D}\ \texttt{end} \qquad (\mathsf{module\ type}\ T = S) \in \overline{D}}{\Gamma \overset{\text{src}}{\vdash} P.T \lhd\!\rhd S[A \mapsto P]}$$

S-Res-ModType-Rec
$$\frac{\Gamma \overset{\text{src}}{\vdash} Q.T \lhd\!\rhd Q'.T' \qquad \Gamma \overset{\text{src}}{\vdash} Q'.T' \lhd\!\rhd S}{\Gamma \overset{\text{src}}{\vdash} Q.T \lhd\!\rhd S}$$

C.2 Equivalence

C.2.1 $\Gamma \overset{\text{src}}{\vdash} \tau \approx \tau'$ – Type equivalence

S-Eqv-Type-Local
$$\frac{(A.t : \mathsf{type}\, t = \tau) \in \Gamma}{\Gamma \overset{\text{src}}{\vdash} A.t \approx \tau}$$

S-Eqv-Type-Res
$$\frac{\Gamma \overset{\text{src}}{\vdash} P \triangleright \texttt{sig}_A\ \overline{D}\ \texttt{end} \qquad (\mathsf{type}\, t = \tau) \in \overline{D}}{\Gamma \overset{\text{src}}{\vdash} P.t \approx \tau[A \mapsto P]}$$

S-Eqv-Type-Trans
$$\frac{\Gamma \overset{\text{src}}{\vdash} \tau_1 \approx \tau_2 \qquad \Gamma \overset{\text{src}}{\vdash} \tau_2 \approx \tau_3}{\Gamma \overset{\text{src}}{\vdash} \tau_1 \approx \tau_3}$$

S-Eqv-Type-Sym
$$\frac{\Gamma \overset{\text{src}}{\vdash} \tau' \approx \tau}{\Gamma \overset{\text{src}}{\vdash} \tau \approx \tau'}$$

S-Eqv-Type-Refl
$$\Gamma \overset{\text{src}}{\vdash} \tau \approx \tau$$

C.2.2 $\Gamma \overset{\text{src}}{\vdash}_A D \approx D'$ – Declaration equivalence

S-Eqv-Decl-Val

$$\frac{\Gamma \overset{\text{src}}{\vdash} \tau \approx \tau'}{\Gamma \overset{\text{src}}{\vdash}_A (\mathsf{val}\, x : \tau) \approx (\mathsf{val}\, x : \tau')}$$

S-Eqv-Decl-Type

$$\frac{\Gamma \overset{\text{src}}{\vdash} \tau \approx \tau'}{\Gamma \overset{\text{src}}{\vdash}_A (\mathsf{type}\, t = \tau) \approx (\mathsf{type}\, t = \tau')}$$

S-Eqv-Decl-Mod

$$\frac{\Gamma \overset{\text{src}}{\vdash} S \approx S'}{\Gamma \overset{\text{src}}{\vdash}_A (\mathsf{module}\, X : S) \approx (\mathsf{module}\, X : S')}$$

S-Eqv-Decl-ModType

$$\frac{\Gamma \overset{\text{src}}{\vdash} S \approx S'}{\Gamma \overset{\text{src}}{\vdash}_A (\mathsf{module\ type}\ T = S) \approx (\mathsf{module\ type}\ T = S')}$$

S-Eqv-Decl-Empty

$$\Gamma \overset{\text{src}}{\vdash} \varnothing \approx \varnothing$$

S-Eqv-Decl-Seq

$$\frac{\Gamma \overset{\text{src}}{\vdash}_A D_1 \approx D_1' \qquad \Gamma, A.I_1 : D_1 \overset{\text{src}}{\vdash}_A \overline{D} \approx \overline{D'}}{\Gamma \overset{\text{src}}{\vdash}_A D_1, \overline{D} \approx D_1', \overline{D'}}$$

C.2.3 $\Gamma \overset{\text{src}}{\vdash} S \approx S'$ – Signature equivalence

S-Eqv-Sig-Modtype

$$\frac{\Gamma \overset{\text{src}}{\vdash} Q.T \lhd\rhd S}{\Gamma \overset{\text{src}}{\vdash} Q.T \approx S}$$

S-Eqv-Sig-Functor

$$\frac{\Gamma \overset{\text{src}}{\vdash} S_a \approx S_a' \qquad \Gamma, Y : S_a \overset{\text{src}}{\vdash} S \approx S'}{\Gamma \overset{\text{src}}{\vdash} (Y : S_a) \to S \approx (Y : S_a') \to S'}$$

S-Eqv-Sig-Sig

$$\frac{\Gamma \overset{\text{src}}{\vdash}_A \overline{D} \approx \overline{D'}}{\Gamma \overset{\text{src}}{\vdash} \mathtt{sig}_A\ \overline{D}\ \mathtt{end} \approx \mathtt{sig}_A\ \overline{D'}\ \mathtt{end}}$$

S-Eqv-Sig-Trans

$$\frac{\Gamma \overset{\text{src}}{\vdash} S \approx S' \qquad \Gamma \overset{\text{src}}{\vdash} S' \approx S''}{\Gamma \overset{\text{src}}{\vdash} S \approx S''}$$

S-Eqv-Sig-Refl

$$\Gamma \overset{\text{src}}{\vdash} S \approx S$$

S-Eqv-Sig-Sym

$$\frac{\Gamma \overset{\text{src}}{\vdash} S' \approx S}{\Gamma \overset{\text{src}}{\vdash} S \approx S'}$$

C.3 Strengthening

C.3.1 $\Gamma \vdash S/P \gg S'$ – Signature strengthening

S-Str-Sig-Functor

$$\Gamma \vdash (Y : S_a) \to S/P \gg (Y : S_a) \to S$$

S-Str-Sig-Sig

$$\frac{\Gamma \vdash \overline{D}[A \mapsto P]\,/P \gg \overline{D'}}{\Gamma \vdash \mathtt{sig}_A\ \overline{D}\ \mathtt{end}/P \gg \mathtt{sig}_A\ \overline{D'}\ \mathtt{end}}$$

C.3.2 $\Gamma \vdash D/P \gg D'$ – Declaration strengthening

S-Str-Decl-Val
$$\Gamma \vdash \mathsf{val}\, x : \tau / P \gg \mathsf{val}\, x : \tau$$

S-Str-Decl-Type
$$\Gamma \vdash \mathsf{type}\, t = \tau / P \gg \mathsf{type}\, t = \tau$$

S-Str-Decl-Mod
$$\frac{\Gamma \vdash S / (P.X) \gg S'}{\Gamma \vdash (\mathsf{module}\, X : S) / P \gg \mathsf{module}\, X : S'}$$

S-Str-Decl-ModType
$$\Gamma \vdash \mathsf{module\ type}\ T = S / P \gg \mathsf{module\ type}\ T = S$$

C.4 Subtyping

Rules for both general and abstraction subtyping are shown, $\prec:$ being either $<:$ or $\lhd:$.

C.4.1 $\Gamma \overset{\mathsf{src}}{\vdash} S <: S'$ – Signature subtyping

S-Sub-Sig-Equiv
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash} S \approx S'}{\Gamma \overset{\mathsf{src}}{\vdash} S \prec: S'}$$

S-Sub-Sig-Trans
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash} S \prec: S' \qquad \Gamma \overset{\mathsf{src}}{\vdash} S' \prec: S''}{\Gamma \overset{\mathsf{src}}{\vdash} S \prec: S''}$$

S-Sub-Sig-Functor
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash} S'_a \prec: S_a \qquad \Gamma, Y : S'_a \overset{\mathsf{src}}{\vdash} S \prec: S' \qquad Y \notin \Gamma}{\Gamma \overset{\mathsf{src}}{\vdash} (Y : S_a) \to S \prec: (Y : S'_a) \to S'}$$

S-Sub-Sig-Sig
$$\frac{\overline{D}_0 \subseteq \overline{D} \qquad \Gamma, \overline{A.D} \overset{\mathsf{src}}{\vdash}_A \overline{D_0} <: \overline{D'}}{\Gamma \overset{\mathsf{src}}{\vdash} \mathtt{sig}_A\ \overline{D}\ \mathtt{end} <: \mathtt{sig}_A\ \overline{D'}\ \mathtt{end}}$$

S-SubEq-Sig-Sig
$$\frac{\Gamma, \overline{A.D} \overset{\mathsf{src}}{\vdash} \overline{D} \lhd: \overline{D'}}{\Gamma \overset{\mathsf{src}}{\vdash} \mathtt{sig}_A\ \overline{D}\ \mathtt{end} \lhd: \mathtt{sig}_A\ \overline{D'}\ \mathtt{end}}$$

C.4.2 $\Gamma \overset{\mathsf{src}}{\vdash}_A D <: D'$ – Declaration subtyping

S-Sub-Decl-Val
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash} \tau \approx \tau'}{\Gamma \overset{\mathsf{src}}{\vdash}_A (\mathsf{val}\, x : \tau) \prec: (\mathsf{val}\, x : \tau')}$$

S-Sub-Decl-Type
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash} \tau \approx \tau'}{\Gamma \overset{\mathsf{src}}{\vdash}_A (\mathsf{type}\, t = \tau) \prec: (\mathsf{type}\, t = \tau')}$$

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

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

C.5 Typing

C.5.1 $\Gamma \overset{\text{src}}{\vdash} S : \checkmark$ – Signature typing

S-Typ-Sig-LocalModType

$$\frac{\Gamma \overset{\text{src}}{\vdash} P.T : \texttt{sig}_A\ \overline{D}\ \texttt{end} \qquad \text{module type } T = S \in \overline{D}}{\Gamma \overset{\text{src}}{\vdash} P.T : \checkmark}$$

S-Typ-Sig-ModType

$$\frac{A.T : \text{module type } T = S \in \Gamma}{\Gamma \overset{\text{src}}{\vdash} A.T : \checkmark}$$

S-Typ-Sig-Functor

$$\frac{\Gamma \overset{\text{src}}{\vdash} S_a : \checkmark \qquad \Gamma, Y : S_a \overset{\text{src}}{\vdash} S : \checkmark \qquad Y \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash} (Y : S_a) \to S : \checkmark}$$

S-Typ-Sig-Sig

$$\frac{\Gamma \overset{\text{src}}{\vdash}_A \overline{D} : \checkmark \qquad A \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash} \texttt{sig}_A\ \overline{D}\ \texttt{end} : \checkmark}$$

C.5.2 $\Gamma \overset{\text{src}}{\vdash}_A D : \checkmark$ – Declarations typing

S-Typ-Decl-Val

$$\frac{\Gamma \overset{\text{src}}{\vdash} \tau : \checkmark \qquad A.x \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_A (\text{val}\, x : \tau) : \checkmark}$$

S-Typ-Decl-Type

$$\frac{\Gamma \overset{\text{src}}{\vdash} \tau : \checkmark \qquad A.t \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_A (\text{type}\, t = \tau) : \checkmark}$$

S-Typ-Decl-TypeAbs

$$\frac{A.t \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_A (\text{type}\, t = A.t) : \checkmark}$$

S-Typ-Decl-Mod

$$\frac{\Gamma \overset{\text{src}}{\vdash} S : \checkmark \qquad A.X \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_A (\text{module}\, X : S) : \checkmark}$$

S-Typ-Decl-ModType

$$\frac{\Gamma \overset{\text{src}}{\vdash} S : \checkmark \qquad A.T \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_A (\text{module type } T = S) : \checkmark}$$

S-Typ-Decl-Empty

$$\Gamma \overset{\text{src}}{\vdash}_A \varnothing : \checkmark$$

S-Typ-Decl-Seq

$$\frac{\Gamma \overset{\text{src}}{\vdash}_A D_1 : \checkmark \qquad \Gamma, A.I_1 : D_1 \overset{\text{src}}{\vdash}_A \overline{D} : \checkmark}{\Gamma \overset{\text{src}}{\vdash}_A (D_1, \overline{D}) : \checkmark}$$

C.5.3 $\Gamma \overset{\text{src}}{\vdash} \tau : \checkmark$ – Type checking

S-Typ-Type-LocalType

$$\frac{A.t : \text{type}\, t = \tau \in \Gamma}{\Gamma \overset{\text{src}}{\vdash} A.t : \checkmark}$$

S-Typ-Type-QualifiedPathType

$$\frac{\Gamma \overset{\text{src}}{\vdash} P : \texttt{sig}_A\ \overline{D}\ \texttt{end} \qquad \text{type}\, t = \tau \in \overline{D}}{\Gamma \overset{\text{src}}{\vdash} P.t : \checkmark}$$

C.5.4 $\Gamma \overset{\text{src}}{\vdash} M : S$ – Module typing

S-Typ-Mod-Res
$$\frac{\Gamma \overset{\text{src}}{\vdash} P \triangleright S}{\Gamma \overset{\text{src}}{\vdash} P : S}$$

S-Typ-Mod-Strengthen
$$\frac{\Gamma \overset{\text{src}}{\vdash} P : S \qquad \Gamma \vdash S/P \gg S'}{\Gamma \overset{\text{src}}{\vdash} P : S'}$$

S-Typ-Mod-Equiv
$$\frac{\Gamma \overset{\text{src}}{\vdash} M : S \qquad \Gamma \overset{\text{src}}{\vdash} S \approx S'}{\Gamma \overset{\text{src}}{\vdash} M : S'}$$

S-Typ-Mod-Sealing
$$\frac{\Gamma \overset{\text{src}}{\vdash} S : \checkmark \qquad \Gamma \overset{\text{src}}{\vdash} P : S' \qquad \Gamma \overset{\text{src}}{\vdash} S' <: S}{\Gamma \overset{\text{src}}{\vdash} (P : S) : S}$$

S-Typ-Mod-Functor
$$\frac{\Gamma \overset{\text{src}}{\vdash} S_a : \checkmark \qquad \Gamma; (Y : S_a) \overset{\text{src}}{\vdash} M : S \qquad Y \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash} (Y : S_a) \to M : (Y : S_a) \to S'}$$

S-Typ-Mod-FctApp
$$\frac{\Gamma \overset{\text{src}}{\vdash} P : (Y : S_a) \to S \qquad \Gamma \overset{\text{src}}{\vdash} P' : S'_a \qquad \Gamma \overset{\text{src}}{\vdash} S'_a <: S_a}{\Gamma \overset{\text{src}}{\vdash} P(P') : S[Y \mapsto P']}$$

S-Typ-Mod-Struct
$$\frac{\Gamma \overset{\text{src}}{\vdash}_A \overline{B} : \overline{D} \qquad A \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash} \texttt{struct}_A\ \overline{B}\ \texttt{end} : \texttt{sig}_A\ \overline{D}\ \texttt{end}}$$

S-Typ-Proj
$$\frac{\Gamma \overset{\text{src}}{\vdash} M : \texttt{sig}_A\ \left(\overline{D_1}, \mathsf{module}\, X : S, \overline{D_2}\right)\ \texttt{end} \qquad \Gamma, \overline{D_1} \overset{\text{src}}{\vdash} S \lhd: S' \qquad \Gamma \overset{\text{src}}{\vdash} S' : \checkmark}{\Gamma \overset{\text{src}}{\vdash} M.X : S'}$$

C.5.5 $\Gamma \overset{\text{src}}{\vdash}_A B : D$ – Bindings typing

S-Typ-Bind-Let
$$\frac{\Gamma \overset{\text{src}}{\vdash} e : \tau \qquad A.x \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_A (\mathsf{let}\, x = e) : (\mathsf{val}\, x : \tau)}$$

S-Typ-Bind-Type
$$\frac{\Gamma \overset{\text{src}}{\vdash} \tau : \checkmark \qquad A.t \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_A (\mathsf{type}\, t = \tau) : (\mathsf{type}\, t = \tau)}$$

S-Typ-Bind-AbsType
$$\frac{A.t \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_A (\mathsf{type}\, t = A.t) : (\mathsf{type}\, t = A.t)}$$

S-Typ-Bind-Mod
$$\frac{\Gamma \overset{\text{src}}{\vdash} M : S \qquad A.X \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_A (\mathsf{module}\, X = M) : (\mathsf{module}\, X : S)}$$

S-Typ-Bind-ModType
$$\frac{\Gamma \overset{\text{src}}{\vdash} S : \checkmark \qquad A.T \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_A (\mathsf{module\ type}\ T = S) : (\mathsf{module\ type}\ T = S)}$$

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

S-Typ-Bind-Seq
$$\frac{\Gamma \overset{\text{src}}{\vdash}_A B_1 : D_1 \qquad \Gamma, A.I_1 : D_1 \overset{\text{src}}{\vdash}_A B_2 : D_2}{\Gamma \overset{\text{src}}{\vdash}_A B_1, B_2 : D_1 +\!\!+ D_2}$$

D Canonical system – Complete rules

D.1 Subtyping

Rules for both general and abstraction subtyping are shown, $\prec:$ being either $<:$ or $\lhd:$.

D.1.1 $\Gamma \overset{\text{can}}{\vdash} \mathcal{S} <: \mathcal{S}'$ – Signatures subtyping

C-Sub-Sig-Functor

$$\frac{\Gamma, \overline{\alpha}' \overset{\text{can}}{\vdash} \mathcal{R}' \prec: \mathcal{R}[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma, \overline{\alpha}', (Y : \mathcal{R}') \overset{\text{can}}{\vdash} \mathcal{S}[\overline{\alpha} \mapsto \overline{\tau}] <: \mathcal{S}' \qquad Y \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash} \forall \overline{\alpha}.\ (Y : \mathcal{R}) \to \mathcal{S} \prec: \forall \overline{\alpha}'.\ (Y : \mathcal{R}') \to \mathcal{S}'}$$

C-Sub-Sig-Match

$$\frac{\Gamma, \overline{\alpha} \overset{\text{can}}{\vdash} \mathcal{R} \prec: \mathcal{R}'[\overline{\alpha}' \mapsto \overline{\tau}]}{\Gamma \overset{\text{can}}{\vdash} \exists \overline{\alpha}.\mathcal{R} \prec: \exists \overline{\alpha}'.\mathcal{R}'}$$

C-Sub-Sig-Sig

$$\frac{\overline{\mathcal{D}_0} \subseteq \overline{\mathcal{D}} \qquad \Gamma \overset{\text{can}}{\vdash} \overline{\mathcal{D}_0} <: \overline{\mathcal{D}'}}{\Gamma \overset{\text{can}}{\vdash} \texttt{sig}_A\ \overline{\mathcal{D}}\ \texttt{end} <: \texttt{sig}_A\ \overline{\mathcal{D}'}\ \texttt{end}}$$

C-SubEq-Sig-Sig

$$\frac{\Gamma \overset{\text{can}}{\vdash} \overline{\mathcal{D}} \lhd: \overline{\mathcal{D}'}}{\Gamma \overset{\text{can}}{\vdash} \texttt{sig}_A\ \overline{\mathcal{D}}\ \texttt{end} \lhd: \texttt{sig}_A\ \overline{\mathcal{D}'}\ \texttt{end}}$$

D.1.2 $\Gamma \overset{\text{can}}{\vdash} \mathcal{D} <: \mathcal{D}'$ – Declarations subtyping

C-Sub-Decl-Val

$$\frac{\Gamma \overset{\text{can}}{\vdash} \tau \prec: \tau'}{\Gamma \overset{\text{can}}{\vdash} (\mathsf{val}\, x : \tau) \prec: (\mathsf{val}\, x : \tau')}$$

C-Sub-Decl-Type

$$\frac{\Gamma \overset{\text{can}}{\vdash} \tau \prec: \tau'}{\Gamma \overset{\text{can}}{\vdash} (\mathsf{type}\, t = \tau) \prec: (\mathsf{type}\, t = \tau')}$$

C-Sub-Decl-Mod

$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathcal{R} \prec: \mathcal{R}'}{\Gamma \overset{\text{can}}{\vdash} (\mathsf{module}\, X : \mathcal{R}) \prec: (\mathsf{module}\, X : \mathcal{R}')}$$

C-Sub-Decl-ModType

$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathcal{R} \prec: \mathcal{R}' \qquad \Gamma \overset{\text{can}}{\vdash} \mathcal{R}' \prec: \mathcal{R}}{\Gamma \overset{\text{can}}{\vdash} (\mathsf{module\ type}\ T = \lambda \overline{\alpha}.\mathcal{R}) \prec: (\mathsf{module\ type}\ T = \lambda \overline{\alpha}.\mathcal{R}')}$$

D.2 Typing

D.2.1 $\Gamma \overset{\text{can}}{\vdash} Q.t : \tau$ – Type checking

C-Typ-Type-LocalType

$$\frac{A.t : \mathsf{type}\, t = \tau \in \Gamma}{\Gamma \overset{\text{can}}{\vdash} A.t : \tau}$$

C-Typ-Type-PathType

$$\frac{\Gamma \overset{\text{can}}{\vdash} P : \texttt{sig}_A\ \overline{\mathcal{D}}\ \texttt{end} \qquad (\mathsf{type}\, t = \tau) \in \overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\vdash} P.t : \tau}$$

D.2.2 $\Gamma \overset{\text{can}}{\vdash_A} D : \lambda\overline{\alpha}.\mathcal{D}$ – Declaration typing

C-Typ-Decl-Val
$$\frac{\Gamma \overset{\text{can}}{\vdash} \tau : \tau' \qquad A.x \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash_A} \mathsf{val}\, x : \tau : \mathsf{val}\, x : \tau'}$$

C-Typ-Decl-Type
$$\frac{\Gamma \overset{\text{can}}{\vdash} \tau : \tau' \qquad A.t \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash_A} \mathsf{type}\, t = \tau : \mathsf{type}\, t = \tau'}$$

C-Typ-Decl-TypeAbs
$$\frac{A.t \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash_A} \mathsf{type}\, t = A.t : \lambda\alpha.\, \mathsf{type}\, t = \alpha}$$

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

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

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

C-Typ-Decl-Seq
$$\frac{\Gamma \overset{\text{can}}{\vdash_A} D_1 : \lambda\overline{\alpha}_1.\mathcal{D}_1 \qquad \Gamma, \overline{\alpha}_1, A.I_1 : \mathcal{D}_1 \overset{\text{can}}{\vdash_A} \overline{\mathcal{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\vdash_A} \left(D_1, \overline{D}\right) : \lambda\overline{\alpha}_1\, \overline{\alpha}.\, \left(\mathcal{D}_1, \overline{\mathcal{D}}\right)}$$

D.2.3 $\Gamma \overset{\text{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{R}$ – Signature typing

C-Typ-Sig-Sig
$$\frac{\Gamma \overset{\text{can}}{\vdash_A} \overline{D} : \lambda\overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash} \mathtt{sig}_A\ \overline{D}\ \mathtt{end} : \lambda\overline{\alpha}.\mathtt{sig}_A\ \overline{\mathcal{D}}\ \mathtt{end}}$$

C-Cf-ModType
$$\frac{\Gamma \overset{\text{can}}{\vdash} P : \mathtt{sig}_A\ \overline{\mathcal{D}}\ \mathtt{end} \qquad \mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{R} \in \overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\vdash} A.T : \lambda\overline{\alpha}.\mathcal{R}}$$

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

C-Typ-Sig-Functor
$$\frac{\Gamma \overset{\text{can}}{\vdash} S_a : \lambda\overline{\alpha}.\mathcal{R}_a \qquad \Gamma, \overline{\alpha}, Y : \mathcal{R}_a \overset{\text{can}}{\vdash} S : \lambda\overline{\beta}.\mathcal{R} \qquad Y \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash} (Y : S_a) \to S : \forall\overline{\alpha}.\ (Y : \mathcal{R}_a) \to \exists\overline{\beta}.\mathcal{R}}$$

D.2.4 $\Gamma \overset{\text{can}}{\vdash} M : \mathcal{S}$ – Module typing

C-Typ-Mod-FctArg
$$\frac{(Y : \mathcal{R}) \in \Gamma}{\Gamma \overset{\mathsf{can}}{\vdash} Y : \mathcal{R}}$$

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

C-Typ-Struct
$$\frac{\Gamma \overset{\mathsf{can}}{\vdash}_A B : \exists\overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \overset{\mathsf{can}}{\vdash} \texttt{struct}_A\ B\ \texttt{end} : \exists\overline{\alpha}.\texttt{sig}_A\ \overline{\mathcal{D}}\ \texttt{end}}$$

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

C-Typ-Mod-Functor
$$\frac{\Gamma \overset{\mathsf{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{R} \qquad \Gamma, \overline{\alpha}, (Y : \mathcal{R}) \overset{\mathsf{can}}{\vdash} M : \mathcal{S}}{\Gamma \overset{\mathsf{can}}{\vdash} (Y : S) \to M : \forall\overline{\alpha}.\ (Y : \mathcal{R}) \to \mathcal{S}}$$

C-Typ-Mod-App
$$\frac{\Gamma \overset{\mathsf{can}}{\vdash} P : \forall\overline{\alpha}.\ (Y : \mathcal{R}) \to \mathcal{S} \qquad \Gamma \overset{\mathsf{can}}{\vdash} P' : \mathcal{R}' \qquad \Gamma \overset{\mathsf{can}}{\vdash} \mathcal{R}' <: \mathcal{R}[\overline{\alpha} \mapsto \overline{\tau}]}{\Gamma \overset{\mathsf{can}}{\vdash} P(P') : \mathcal{S}[\overline{\alpha} \mapsto \overline{\tau}]}$$

C-Typ-Mod-Proj
$$\frac{\Gamma \overset{\mathsf{can}}{\vdash} M : \exists\overline{\alpha}.\texttt{sig}_A\ \overline{\mathcal{D}}_1, \mathsf{module}\ X : \mathcal{R}, \overline{\mathcal{D}}_2\ \texttt{end}}{\Gamma \overset{\mathsf{can}}{\vdash} M.X : \exists\overline{\alpha}.\mathcal{R}}$$

D.2.5 $\Gamma \overset{\mathsf{can}}{\vdash}_A B : \exists\overline{\alpha}.\mathcal{D}$ – Bindings typing

C-Typ-Decl-Let
$$\frac{\Gamma \overset{\mathsf{can}}{\vdash} e : \tau \qquad \Gamma \overset{\mathsf{can}}{\vdash} \tau : \tau' \qquad A.x \notin \Gamma}{\Gamma \overset{\mathsf{can}}{\vdash}_A (\mathsf{let}\, x = e) : (\mathsf{val}\, x : \tau')}$$

C-Typ-Decl-Type
$$\frac{\Gamma \overset{\mathsf{can}}{\vdash} \tau : \tau' \qquad A.t \notin \Gamma}{\Gamma \overset{\mathsf{can}}{\vdash}_A (\mathsf{type}\, t = \tau) : (\mathsf{type}\, t = \tau')}$$

C-Typ-Decl-Mod
$$\frac{\Gamma \overset{\mathsf{can}}{\vdash} M : \exists\overline{\alpha}.\mathcal{R} \qquad A.X \notin \Gamma}{\Gamma \overset{\mathsf{can}}{\vdash}_A (\mathsf{module}\, X = M) : (\exists\overline{\alpha}.\ \mathsf{module}\, X : \mathcal{R})}$$

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

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

C-Typ-Decl-Seq
$$\frac{\Gamma \overset{\mathsf{can}}{\vdash}_A B_1 : \exists\overline{\alpha}_1.\mathcal{D}_1 \qquad \Gamma, \overline{\alpha}_1, A.I_1 : \mathcal{D}_1 \overset{\mathsf{can}}{\vdash}_A \overline{B} : \exists\overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \overset{\mathsf{can}}{\vdash}_A B_1, \overline{B} : \exists\overline{\alpha}_1\overline{\alpha}.\overline{\mathcal{D}_1}, \overline{\mathcal{D}}}$$

D.3 Anchoring

D.3.1 $\Gamma \overset{\mathsf{anc}}{\vdash} \tau \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash \tau'$ – Type anchoring

C-Ach-Type-Anchor

$$\frac{\Gamma \hookrightarrow \Gamma_{\mathsf{s}} : \theta_\Gamma \qquad \theta_\Gamma(\alpha) = Q.t}{\Gamma \overset{\mathsf{anc}}{\vdash} \alpha \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash Q.t}$$

C-Ach-Decl-Type

$$\frac{\Gamma \overset{\mathsf{anc}}{\vdash} \tau \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash \tau' : \varnothing}{\Gamma \overset{\mathsf{anc}}{\vdash}_A \mathsf{type}\, t = \tau \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash \mathsf{type}\, t = \tau' : \varnothing}$$

D.3.2 $\Gamma \overset{\mathsf{anc}}{\vdash}_A \mathcal{D} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash D : \theta$ – Declaration anchoring

C-Ach-Decl-Val

$$\frac{\Gamma \overset{\mathsf{anc}}{\vdash} \tau \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash \tau' : \varnothing}{\Gamma \overset{\mathsf{anc}}{\vdash}_A \mathsf{val}\, x : \tau \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash \mathsf{val}\, x : \tau' : \varnothing}$$

C-Ach-Decl-AnchorPoint

$$\frac{\Gamma \hookrightarrow \Gamma_{\mathsf{s}} : \theta_\Gamma \qquad \alpha \notin \mathrm{dom}(\theta_\Gamma)}{\Gamma \overset{\mathsf{anc}}{\vdash}_A \mathsf{type}\, t = \alpha \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash \mathsf{type}\, t = A.t : (\alpha \mapsto t)}$$

C-Ach-Decl-Type

$$\frac{\Gamma \overset{\mathsf{anc}}{\vdash} \tau \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash \tau' : \varnothing}{\Gamma \overset{\mathsf{anc}}{\vdash}_A \mathsf{type}\, t = \tau \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash \mathsf{type}\, t = \tau' : \varnothing}$$

C-Ach-Decl-Mod

$$\frac{\Gamma \overset{\mathsf{anc}}{\vdash} \mathcal{R} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash S : \theta}{\Gamma \overset{\mathsf{anc}}{\vdash}_A \mathsf{module}\, X : \mathcal{R} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash \mathsf{module}\, X : S : X.\theta}$$

C-Ach-Decl-Sig

$$\frac{\Gamma, \overline{\alpha} \overset{\mathsf{anc}}{\vdash} \mathcal{R} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash S : \theta \qquad \mathrm{dom}(\theta) = \overline{\alpha}}{\Gamma \overset{\mathsf{anc}}{\vdash}_A \mathsf{module}\, T : \lambda\overline{\alpha}.\mathcal{R} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash \mathsf{module}\, T : S : \varnothing}$$

C-Ach-Decl-Seq

$$\frac{\Gamma \overset{\mathsf{anc}}{\vdash} \mathcal{D} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash D : \theta \qquad \Gamma \overset{\mathsf{anc}}{\vdash} \overline{\mathcal{D}} \hookrightarrow (\Gamma_{\mathsf{s}}, A.I : D)\,; (\theta_\Gamma \uplus A.\theta) \vdash \overline{D} : \theta'}{\Gamma \overset{\mathsf{anc}}{\vdash}_A \mathcal{D}, \overline{\mathcal{D}} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash D, \overline{D} : \theta_\Gamma \uplus \theta'}$$

D.3.3 $\Gamma \overset{\mathsf{can}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{R}$ – Signature anchoring

C-Ach-Sig-Functor

$$\frac{\begin{array}{c}\Gamma, \overline{\alpha} \overset{\mathsf{anc}}{\vdash} \mathcal{R} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash S_a : \theta_a \\ \mathrm{dom}(\theta_a) = \overline{\alpha} \qquad \Gamma, \overline{\alpha}, Y : \mathcal{R} \overset{\mathsf{anc}}{\vdash} \mathcal{S} \hookrightarrow (\Gamma_{\mathsf{s}}, Y : S_a)\,; (\theta_\Gamma \uplus Y.\theta) \vdash S : \theta\end{array}}{\Gamma \overset{\mathsf{anc}}{\vdash} \forall\overline{\alpha}.\ (Y : \mathcal{R}) \to \mathcal{S} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash (Y : S_a) \to S : \varnothing}$$

C-Ach-Sig-Sig

$$\frac{\Gamma \overset{\mathsf{anc}}{\vdash}_A \overline{\mathcal{D}} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash \overline{D} : \theta}{\Gamma \overset{\mathsf{anc}}{\vdash} \mathtt{sig}_A\ \overline{\mathcal{D}}\ \mathtt{end} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash \mathtt{sig}_A\ \overline{D}\ \mathtt{end} : \theta}$$

D.3.4 $\Gamma \hookrightarrow \Gamma_{\mathsf{s}} : \theta_\Gamma$ – Environment anchoring

C-Ach-Env-Empty

$$\varnothing \hookrightarrow \varnothing : \varnothing$$

C-Ach-Env-FctArg

$$\frac{\Gamma \hookrightarrow \Gamma_{\mathsf{s}} : \theta_\Gamma \qquad \Gamma, \overline{\alpha} \overset{\mathsf{anc}}{\vdash} \mathcal{R} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash S : \theta \qquad \mathrm{dom}(\theta) = \overline{\alpha}}{\Gamma, \overline{\alpha}, Y : \mathcal{R} \hookrightarrow (\Gamma_{\mathsf{s}}, Y : S) : \theta_\Gamma \uplus Y.\theta}$$

C-Ach-Env-Decl

$$\frac{\Gamma \hookrightarrow \Gamma_{\mathsf{s}} : \theta_\Gamma \qquad \Gamma, \overline{\alpha} \overset{\mathsf{anc}}{\vdash}_A \mathcal{D} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash D : \theta \qquad \mathrm{dom}(\theta) = \overline{\alpha}}{\Gamma, \overline{\alpha}, A.I : \mathcal{D} \hookrightarrow (\Gamma_{\mathsf{s}}, A.I : D) : \theta_\Gamma \uplus A.\theta}$$

D.3.5 $\Gamma \overset{\mathsf{anc}}{\vdash} M : S$ – Anchorable typing

C-TypA-Mod-Proj

$$\frac{\Gamma \overset{\mathsf{anc}}{\vdash} M : \exists \overline{\alpha}.\mathtt{sig}_A\ \overline{\mathcal{D}}_1, \mathsf{module}\ X : \mathcal{R}, \overline{\mathcal{D}}_2\ \mathsf{end} \qquad \Gamma \overset{\mathsf{anc}}{\vdash} \exists \overline{\alpha}.\mathcal{R} \hookrightarrow \Gamma_{\mathsf{s}}; \theta_\Gamma \vdash S : \theta}{\Gamma \overset{\mathsf{anc}}{\vdash} M.X : \exists \overline{\alpha}.\mathcal{R}}$$

E F^{ω}– Complete rules

E.1 $\Gamma : \mathsf{wf}$ – Environment checking

$$\cdot : \mathsf{wf} \qquad \frac{\Gamma : \mathsf{wf} \qquad \alpha \notin \Gamma}{\Gamma, \alpha : \kappa : \mathsf{wf}} \qquad \frac{\Gamma \vdash \tau : \star \qquad x \notin \Gamma}{\Gamma, (x : \tau) : \mathsf{wf}}$$

E.2 $\Gamma \vdash \tau : \kappa$ – Type checking

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

$$\frac{\Gamma, \alpha : \kappa \vdash \tau : \star}{\Gamma \vdash \exists \alpha : \kappa.\tau : \star} \qquad \frac{\Gamma, \alpha : \kappa \vdash \tau : \kappa'}{\Gamma \vdash \lambda \alpha : \kappa.\tau : \kappa \to \kappa'} \qquad \frac{\Gamma \vdash \tau_1 : \kappa' \to \kappa \qquad \Gamma \vdash \tau_2 : \kappa'}{\Gamma \vdash \tau_1\ \tau_2 : \kappa}$$

E.3 $\Gamma \vdash e : \tau$ – Term typing

$$\frac{\Gamma : \mathsf{wf}}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau.e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\overline{\Gamma \vdash e : \tau} \qquad \Gamma : \mathsf{wf}}{\Gamma \vdash \{\overline{l = e}\} : \{\overline{l : \tau}\}} \qquad \frac{\Gamma \vdash e : \{l : \tau, \overline{l' : \tau'}\}}{\Gamma \vdash e.l : \tau} \qquad \frac{\Gamma, \alpha : \kappa \vdash e : \tau}{\Gamma \vdash \Lambda(\alpha : \kappa).e : \forall(\alpha : \kappa).\tau}$$

$$\frac{\Gamma \vdash e : \forall \alpha : \kappa.\tau' \qquad \Gamma \vdash \tau : \kappa}{\Gamma \vdash e\ \tau : \tau'[\tau \mapsto \alpha]} \qquad \frac{\Gamma \vdash \tau : \kappa \qquad \Gamma \vdash e : \tau'[\tau \mapsto \alpha] \qquad \Gamma \vdash \exists \alpha : \kappa.\tau' : \star}{\Gamma \vdash \mathsf{pack}\ \langle \tau, e \rangle\ \mathsf{as}\ \exists \alpha : \kappa.\tau' : \exists \alpha : \kappa.\tau'}$$

$$\frac{\Gamma \vdash e_1 : \exists \alpha : \kappa.\tau' \qquad \Gamma, \alpha : \kappa, x : \tau' \vdash e_2 : \tau \qquad \Gamma \vdash \tau : \star}{\Gamma \vdash \mathsf{unpack} \langle \alpha, x \rangle = e_1\ \mathsf{in}\ e_2 : \tau}$$

F Elaborated system – Complete rules

F.1 Subtyping

F.1.1 $\Delta \overset{\text{elab}}{\vdash} \mathcal{S} <: \mathcal{S}' \rightsquigarrow \Pi <: \Pi' \Rightarrow f$ – Signature subtyping

E-Sub-Sig-Functor

$$\frac{\Delta, \overline{\alpha}' \overset{\text{elab}}{\vdash} \mathcal{R}'_a <: \mathcal{R}_a[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow \Sigma'_a <: \Sigma_a[\overline{\alpha} \mapsto \overline{\tau}] \Rightarrow f_a \qquad \Delta, \overline{\alpha}', (Y : (\mathcal{R}'_a, \Sigma'_a)) \overset{\text{elab}}{\vdash} \mathcal{S}[\overline{\alpha} \mapsto \overline{\tau}] <: \mathcal{S}' \rightsquigarrow \Pi[\overline{\alpha} \mapsto \overline{\tau}] <: \Pi' \Rightarrow f \qquad Y \notin \Delta}{\Delta \overset{\text{elab}}{\vdash} \forall \overline{\alpha}.\ (Y : \mathcal{R}_a) \to \mathcal{S} <: \forall \overline{\alpha}'.\ (Y : \mathcal{R}'_a) \to \mathcal{S}' \rightsquigarrow \forall \overline{\alpha}.\ \Sigma_a \to \Pi <: \forall \overline{\alpha}'.\ \Sigma'_a \to \Pi' \\ [\lambda\,(g : (\forall \overline{\alpha}.\ \Sigma_a \to \Pi))\,.\,\Lambda \overline{\alpha}'.\lambda\,(x : \Sigma'_a)\,.\,f\ (g\ \overline{\tau}\ (f_a\ x))]}$$

E-Sub-Sig-Sig

$$\frac{\overline{\mathcal{D}_0} \subseteq \overline{\mathcal{D}} \qquad \Delta \overset{\text{elab}}{\vdash}_A \overline{\mathcal{D}_0} <: \overline{\mathcal{D}'} \rightsquigarrow \overline{\zeta_0} <: \overline{\zeta} \Rightarrow \overline{f}}{\Delta \overset{\text{elab}}{\vdash} \texttt{sig}_A\ \overline{\mathcal{D}}\ \texttt{end} <: \texttt{sig}_A\ \overline{\mathcal{D}'}\ \texttt{end} \rightsquigarrow \left\{\overline{\ell_I : \zeta}\right\} <: \left\{\overline{\ell_I : \zeta'}\right\} \Rightarrow \lambda x.\left\{\overline{\ell_{I_0} : (f\ \zeta)}\right\}}$$

E-SubEq-Sig-Sig

$$\frac{\Delta \overset{\text{elab}}{\vdash}_A \overline{\mathcal{D}} <: \overline{\mathcal{D}'} \rightsquigarrow \overline{\zeta} <: \overline{\zeta}' \Rightarrow \overline{f}}{\Delta \overset{\text{elab}}{\vdash} \texttt{sig}_A\ \overline{\mathcal{D}}\ \texttt{end} <: \texttt{sig}_A\ \overline{\mathcal{D}'}\ \texttt{end} \rightsquigarrow \left\{\overline{\ell_I : \zeta}\right\} <: \left\{\overline{\ell_I : \zeta'}\right\} \Rightarrow \lambda x.\left\{\overline{\ell_I : (f\ \zeta)}\right\}}$$

F.1.2 $\Delta \overset{\text{elab}}{\vdash}_A \mathcal{D} <: \mathcal{D}' \rightsquigarrow \zeta <: \zeta' \Rightarrow f$ – Declaration subtyping

E-Sub-Val

$$\frac{\Delta \overset{\text{elab}}{\vdash} \tau <: \tau' \uparrow f}{\Delta \overset{\text{elab}}{\vdash}_A (\mathsf{val}\, x : \tau) <: (\mathsf{val}\, x : \tau') \rightsquigarrow [\![\tau]\!] <: [\![\tau']\!] \Rightarrow \lambda x.[\![f(x.\texttt{val})]\!]}$$

E-Sub-Type

$$\frac{\Delta \overset{\text{elab}}{\vdash} \tau <: \tau' \uparrow f}{\Delta \overset{\text{elab}}{\vdash}_A (\mathsf{type}\, t = \tau) <: (\mathsf{type}\, t = \tau') \rightsquigarrow [\![\tau : \kappa]\!] <: [\![\tau' : \kappa]\!] \Rightarrow \lambda x.[\![\tau' : \kappa]\!]}$$

E-Sub-Mod

$$\frac{\Delta \overset{\text{elab}}{\vdash} \mathcal{R} <: \mathcal{R}' \rightsquigarrow \Sigma <: \Sigma' \Rightarrow f}{\Delta \overset{\text{elab}}{\vdash}_A (\mathsf{module}\, X : \mathcal{R}) <: (\mathsf{module}\, X : \mathcal{R}') \rightsquigarrow [\![\Sigma]\!] <: [\![\Sigma']\!] \Rightarrow \lambda x.[\![f(x.\texttt{mod})]\!]}$$

E-Sub-ModType

$$\frac{\Delta, \overline{\alpha} \overset{\text{elab}}{\vdash} \mathcal{R} <: \mathcal{R}' \rightsquigarrow \Sigma <: \Sigma' \Rightarrow f \qquad \Delta, \overline{\alpha} \overset{\text{elab}}{\vdash} \mathcal{R}' <: \mathcal{R} \rightsquigarrow \Sigma' <: \Sigma \Rightarrow f'}{\Delta \overset{\text{elab}}{\vdash}_A (\mathsf{module\ type}\ T = \lambda \overline{\alpha}.\mathcal{R}) <: (\mathsf{module\ type}\ T = \lambda \overline{\alpha}.\mathcal{R}') \rightsquigarrow [\![\lambda \overline{\alpha}.\Sigma]\!] <: [\![\lambda \overline{\alpha}.\Sigma']\!] \\ \Rightarrow \lambda x.[\![\lambda \overline{\alpha}.\Sigma']\!]}$$

F.2 Typing

F.2.1 $\Delta \overset{\text{elab}}{\vdash} Q.t : \tau$ – Type checking

E-Typ-Type-LocalType

$$\frac{A.t : (\mathsf{type}\, t = \tau, \mathsf{type}\, t = \tau) \in \Delta}{\Delta \overset{\mathsf{elab}}{\vdash} A.t : \tau}$$

E-Typ-Type-PathType

$$\frac{\Delta \overset{\mathsf{elab}}{\vdash} P : \mathtt{sig}_A\ \overline{\mathcal{D}}\ \mathtt{end} \rightsquigarrow _ : \Sigma \qquad (\mathsf{type}\, t = \tau) \in \overline{\mathcal{D}} \qquad \Sigma.\ell_t : \langle\!\langle \tau \rangle\!\rangle}{\Delta \overset{\mathsf{elab}}{\vdash} P.t : \tau}$$

F.2.2 $\Delta \overset{\mathsf{elab}}{\vdash}_A D : \lambda\overline{\alpha}.\mathcal{D} \rightsquigarrow \lambda\overline{\alpha}.\zeta$ – Declaration typing

E-Typ-Decl-Val

$$\frac{\Delta \overset{\mathsf{elab}}{\vdash} \tau : \tau' \qquad A.x \notin \Delta}{\Delta \overset{\mathsf{elab}}{\vdash}_A \mathsf{val}\, x : \tau : \mathsf{val}\, x : \tau' \rightsquigarrow \mathsf{val}\, x : \tau'}$$

E-Typ-Decl-Type

$$\frac{\Delta \overset{\mathsf{elab}}{\vdash} \tau : \tau' \qquad A.t \notin \Delta}{\Delta \overset{\mathsf{elab}}{\vdash}_A \mathsf{type}\, t = \tau : \mathsf{type}\, t = \tau' \rightsquigarrow \mathsf{type}\, t = \tau'}$$

E-Typ-Decl-TypeAbs

$$\frac{A.t \notin \Delta}{\Delta \overset{\mathsf{elab}}{\vdash}_A \mathsf{type}\, t = A.t : \lambda\alpha.\, \mathsf{type}\, t = \alpha \rightsquigarrow \lambda\alpha.\, \mathsf{type}\, t = \alpha}$$

E-Typ-Decl-Mod

$$\frac{\Delta \overset{\mathsf{elab}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{R} \rightsquigarrow \lambda\overline{\alpha}.\Sigma \qquad A.X \notin \Delta}{\Delta \overset{\mathsf{elab}}{\vdash}_A (\mathsf{module}\, X : S) : \lambda\overline{\alpha}.\, (\mathsf{module}\, X : \mathcal{R}) \rightsquigarrow \lambda\overline{\alpha}.\, \mathsf{module}\, X : \Sigma}$$

E-Typ-Decl-ModType

$$\frac{\Delta \overset{\mathsf{elab}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{R} \rightsquigarrow \lambda\overline{\alpha}.\Sigma \qquad A.T \notin \Delta}{\Delta \overset{\mathsf{elab}}{\vdash}_A (\mathsf{module\ type}\ T = S) : (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{R}) \rightsquigarrow (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\Sigma)}$$

E-Typ-Decl-Empty

$$\Delta \overset{\mathsf{elab}}{\vdash}_A \varnothing : \varnothing \rightsquigarrow \varnothing$$

E-Typ-Decl-Seq

$$\frac{\Delta \overset{\mathsf{elab}}{\vdash}_A D_1 : \lambda\overline{\alpha}_1.\mathcal{D}_1 \rightsquigarrow \lambda\overline{\alpha}_1.\zeta_1 \qquad \Delta, \overline{\alpha}_1, A.I_1 : (\mathcal{D}_1, \zeta_1) \overset{\mathsf{elab}}{\vdash}_A \overline{\mathcal{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}} \rightsquigarrow \lambda\overline{\alpha}.\overline{\zeta}}{\Delta \overset{\mathsf{elab}}{\vdash}_A \left(D_1, \overline{D}\right) : \lambda\overline{\alpha}_1\, \overline{\alpha}.\, \left(\mathcal{D}_1, \overline{\mathcal{D}}\right) \rightsquigarrow \lambda\overline{\alpha}_1\, \overline{\alpha}.\, \left(\zeta_1, \overline{\zeta}\right)}$$

F.2.3 $\Delta \overset{\mathsf{elab}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{R} \rightsquigarrow \lambda\overline{\alpha}.\Sigma$ – Signature typing

E-Typ-Sig-Sig

$$\frac{\Delta \overset{\text{elab}}{\vdash}_A \overline{D} : \lambda\overline{\alpha}.\overline{\mathcal{D}} \rightsquigarrow \lambda\overline{\alpha}.\overline{\zeta} \qquad A \notin \Delta}{\Delta \overset{\text{elab}}{\vdash} \texttt{sig}_A\ \overline{D}\ \texttt{end} : \lambda\overline{\alpha}.\texttt{sig}_A\ \overline{\mathcal{D}}\ \texttt{end} \rightsquigarrow \lambda\overline{\alpha}.\ \{\overline{\zeta}\}}$$

E-Cf-ModType

$$\frac{\Delta \overset{\text{elab}}{\vdash} P : \texttt{sig}_A\ \overline{\mathcal{D}}\ \texttt{end} \rightsquigarrow _ : \Sigma' \qquad \text{module type } T = \lambda\overline{\alpha}.\mathcal{R} \in \overline{\mathcal{D}} \qquad \Sigma'.\ell_T : \langle\!\langle \lambda\overline{\alpha}.\Sigma \rangle\!\rangle}{\Delta \overset{\text{elab}}{\vdash} A.T : \lambda\overline{\alpha}.\mathcal{R} \rightsquigarrow \lambda\overline{\alpha}.\Sigma}$$

E-Typ-Sig-LocalModType

$$\frac{A.I : (\text{module type } T = \lambda\overline{\alpha}.\mathcal{R}, \text{module type } T = \lambda\overline{\alpha}.\Sigma) \in \Delta}{\Delta \overset{\text{elab}}{\vdash} A.T : \lambda\overline{\alpha}.\mathcal{R} \rightsquigarrow \lambda\overline{\alpha}.\Sigma}$$

E-Typ-Sig-Functor

$$\frac{\Delta \overset{\text{elab}}{\vdash} S_a : \lambda\overline{\alpha}.\mathcal{R}_a \rightsquigarrow \lambda\overline{\alpha}.\Sigma_a \qquad \Delta, \overline{\alpha}, Y : (\mathcal{R}_a, \Sigma_a) \overset{\text{elab}}{\vdash} S : \lambda\overline{\beta}.\mathcal{R} \rightsquigarrow \lambda\overline{\beta}.\Sigma \qquad Y \notin \Delta}{\Delta \overset{\text{elab}}{\vdash} (Y : S_a) \to S : \forall\overline{\alpha}.\ (Y : \mathcal{R}_a) \to \exists\overline{\beta}.\mathcal{R} \rightsquigarrow \forall\overline{\alpha}.\Sigma_a \to \exists\overline{\beta}.\Sigma}$$

F.2.4 $\Delta \overset{\text{elab}}{\vdash} M : \mathcal{S} \rightsquigarrow e : \Pi$ – Module typing

E-Typ-Mod-FctArg

$$\frac{Y : (\mathcal{R}, \Sigma) \in \Delta}{\Delta \overset{\mathsf{elab}}{\vdash} Y : \mathcal{R} \rightsquigarrow Y : \Sigma}$$

E-Typ-Mod-Local

$$\frac{A.X : (\mathcal{R}, \Sigma) \in \Delta}{\Delta \overset{\mathsf{elab}}{\vdash} A.X : \mathcal{R} \rightsquigarrow A.X : \Sigma}$$

E-Typ-Struct

$$\frac{\Delta \overset{\mathsf{elab}}{\vdash}_A B : \exists\overline{\alpha}.\overline{\mathcal{D}} \rightsquigarrow e : \exists\overline{\alpha}.\left\{\overline{\zeta}\right\} \qquad A \notin \Delta}{\Delta \overset{\mathsf{elab}}{\vdash} \texttt{struct}_A\ B\ \texttt{end} : \exists\overline{\alpha}.\texttt{sig}_A\ \overline{\mathcal{D}}\ \texttt{end} \rightsquigarrow e : \exists\overline{\alpha}.\left\{\overline{\zeta}\right\}}$$

E-Typ-Mod-Sealing

$$\frac{\begin{array}{c}\Delta \overset{\mathsf{elab}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{R} \rightsquigarrow \lambda\overline{\alpha}.\Sigma \\ \Delta \overset{\mathsf{elab}}{\vdash} P : \mathcal{R}' \rightsquigarrow e : \Sigma' \qquad \Delta \overset{\mathsf{elab}}{\vdash} \mathcal{R}' <: \mathcal{R}[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow \Sigma' <: \Sigma[\overline{\alpha} \mapsto \overline{\tau}] \Rightarrow f\end{array}}{\Delta \overset{\mathsf{elab}}{\vdash} (P : S) : \exists\overline{\alpha}.\mathcal{R} \rightsquigarrow \mathsf{pack}\langle\overline{\tau}, (f\ e)\rangle : \lambda\overline{\alpha}.\Sigma}$$

E-Typ-Mod-Functor

$$\frac{\Delta \overset{\mathsf{elab}}{\vdash} S : \lambda\overline{\alpha}.\mathcal{R} \rightsquigarrow \lambda\overline{\alpha}.\Sigma \qquad \Delta, \overline{\alpha}, Y : (\mathcal{R}, \Sigma) \overset{\mathsf{elab}}{\vdash} M : \mathcal{S} \rightsquigarrow e : \Pi}{\Delta \overset{\mathsf{elab}}{\vdash} (Y : S) \to M : \forall\overline{\alpha}.\ (Y : \mathcal{R}) \to \mathcal{S} \rightsquigarrow \forall\overline{\alpha}.\lambda(Y : \mathcal{R}).e : \forall\overline{\alpha}.\Sigma \to \Pi}$$

E-Typ-Mod-App

$$\frac{\begin{array}{c}\Delta \overset{\mathsf{elab}}{\vdash} P : \forall\overline{\alpha}.\ (Y : \mathcal{R}) \to \mathcal{S} \rightsquigarrow e_P : \forall\overline{\alpha}.\Sigma \to \mathcal{S} \\ \Delta \overset{\mathsf{elab}}{\vdash} P' : \mathcal{R}' \rightsquigarrow e : \Sigma' \qquad \Delta \overset{\mathsf{elab}}{\vdash} \mathcal{R}' <: \mathcal{R}[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow \Sigma' <: \Sigma[\overline{\alpha} \mapsto \overline{\tau}] \Rightarrow f\end{array}}{\Delta \overset{\mathsf{elab}}{\vdash} P(P') : \mathcal{R}[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow e_P\ \overline{\tau}\ (f\ e) : \Pi[\overline{\alpha} \mapsto \overline{\tau}]}$$

E-Typ-Mod-Proj

$$\frac{\Delta \overset{\mathsf{elab}}{\vdash} M : \exists\overline{\alpha}.\texttt{sig}_A\ \overline{\mathcal{D}}_1, \mathsf{module}\ X : \mathcal{R}, \overline{\mathcal{D}}_2\ \texttt{end} \rightsquigarrow e : \left\{\overline{\zeta_1}, \ell_X : \Sigma, \overline{\zeta_2}\right\}}{\Delta \overset{\mathsf{elab}}{\vdash} M.X : \exists\overline{\alpha}.\mathcal{R} \rightsquigarrow \mathsf{unpack}\langle\overline{\alpha}, y\rangle = e\ \mathsf{in}\ \mathsf{pack}\langle\overline{\alpha}, e.\ell_X\rangle : \exists\overline{\alpha}.\Sigma}$$

F.2.5 $\Delta \overset{\mathsf{elab}}{\vdash}_A B : \exists\overline{\alpha}.\mathcal{D} \rightsquigarrow e : \exists\overline{\alpha}.\left\{\overline{\ell_I : \zeta}\right\}$ – Bindings typing

E-Typ-Decl-Let

$$\frac{\Delta \overset{\text{elab}}{\vdash} e : \tau \rightsquigarrow e' : \tau' \qquad \Delta \overset{\text{elab}}{\vdash} \tau : \tau' \qquad A.x \notin \Delta}{\Delta \overset{\text{elab}}{\vdash_A} (\mathsf{let}\, x = e) : (\mathsf{val}\, x : \tau') \rightsquigarrow \{\ell_x = e'\} : (\mathsf{val}\, x : \tau')}$$

E-Typ-Decl-Type

$$\frac{\Delta \overset{\text{elab}}{\vdash} \tau : \tau' \qquad A.t \notin \Delta}{\Delta \overset{\text{elab}}{\vdash_A} (\mathsf{type}\, t = \tau) : (\mathsf{type}\, t = \tau') \rightsquigarrow \{\ell_t = \langle\!\langle \tau' \rangle\!\rangle\} : (\mathsf{type}\, t = \tau')}$$

E-Typ-Decl-Mod

$$\frac{\Delta \overset{\text{elab}}{\vdash} M : \exists \overline{\alpha}.\mathcal{R} \rightsquigarrow e : \exists \overline{\alpha}.\Sigma \qquad A.X \notin \Delta}{\begin{array}{l}\Delta \overset{\text{elab}}{\vdash_A} (\mathsf{module}\, X = M) : (\exists \overline{\alpha}.\, \mathsf{module}\, X : \mathcal{R}) \rightsquigarrow \mathsf{unpack}\langle \overline{\alpha}, y\rangle = e \text{ in } \mathsf{pack}\langle \overline{\alpha}, \{\ell_X = y\}\rangle \\ \qquad\qquad : \exists \overline{\alpha}.\, \mathsf{module}\, X : \Sigma\end{array}}$$

E-Typ-Decl-ModType

$$\frac{\Delta \overset{\text{elab}}{\vdash} S : \lambda \overline{\alpha}.\mathcal{R} \rightsquigarrow \lambda \overline{\alpha}.\Sigma \qquad A.T \notin \Delta}{\begin{array}{l}\Delta \overset{\text{elab}}{\vdash_A} (\mathsf{module\ type}\ T = S) : (\mathsf{module\ type}\ T = \lambda \overline{\alpha}.\mathcal{R}) \rightsquigarrow \{\ell_T = \langle\!\langle \lambda \overline{\alpha}.\mathcal{R} \rangle\!\rangle\} \\ \qquad\qquad : (\mathsf{module\ type}\ T = \lambda \overline{\alpha}.\mathcal{R})\end{array}}$$

E-Typ-Decl-Seq

$$\frac{\Delta \overset{\text{elab}}{\vdash_A} B_1 : \exists \overline{\alpha}_1.\mathcal{D}_1 \rightsquigarrow e_1 : \exists \overline{\alpha}_1.\, \{\zeta_1\} \qquad \Delta, \overline{\alpha}_1, A.I_1 : (\mathcal{D}_1, \zeta_1) \overset{\text{elab}}{\vdash_A} \overline{B} : \exists \overline{\alpha}.\overline{\mathcal{D}} \rightsquigarrow e : \exists \overline{\alpha}.\, \{\overline{\zeta}\}}{\begin{array}{ll}\Delta \overset{\text{elab}}{\vdash_A} B_1, \overline{B} : \exists \overline{\alpha}_1 \overline{\alpha}.\overline{\mathcal{D}_1}, \overline{\mathcal{D}} \rightsquigarrow & \mathsf{unpack}\langle \overline{\alpha}_1, x_1\rangle = e_1 \text{ in} \qquad\qquad : \exists \overline{\alpha}_1 \overline{\alpha}.\, \{\zeta_1, \overline{\zeta}\} \\ & \mathsf{unpack}\langle \overline{\alpha}, x\rangle = (\mathsf{let}\, A.I_1 = x_1.\ell_{I_1} \text{ in } e) \text{ in} \\ & \mathsf{pack}\langle \overline{\alpha}_1 \overline{\alpha}, \{\ell_I = x_1.\ell_I, \overline{\ell_I = x.\ell_I}\}\rangle\end{array}}$$