HAL
open science

OCaml modules: formalization, insights and improvements

Clément Blaudeau

To cite this version:

Clément Blaudeau. OCaml modules: formalization, insights and improvements: Bringing the existential types into a generative subset. Computer Science [cs]. 2021. hal-03526068

HAL Id: hal-03526068

https://inria.hal.science/hal-03526068

Submitted on 14 Jan 2022

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.

OCaml modules: formalization, insights and improvements

Bringing the existential types into a generative subset

EPFL Master thesis

Clément Blaudeau

January 14, 2022

This work was done at Inria Paris during the spring semester of 2021, under the direction of Didier Rémy (Inria) and Gabriel Radanne (Inria) and under the supervision of Viktor Kunčak (EPFL).

Abstract Modularity is a key tool for building reusable and structured code. While the OCaml module system is known for being expressive and powerful, specifying its behavior and formalizing its properties has proven to be hard. We propose a comprehensive description of a generative subset of the OCaml module system (called the *source system*), with a focus on the *signature avoidance problem*. By extending the signature syntax with *existential* types (inspired by F^ω), we describe an alternate system (called *canonical*) with a simpler set of judgments that both solves the issues of the source description and provides formal guarantees through elaboration into F^ω. We show that the canonical system is more expressive than the source one and state at which conditions canonical typing derivations can be translated back into the source typing. As a middle point between the path-based representation of OCaml and the formal description of F^ω, the canonical presentation is a framework which we believe could support numerous features (applicative functors, transparent ascription, module aliases, etc.) and could serve as a basis for a redefinition of module system of an industry-grade language as OCaml.

Contents

Introduction **3**

1 A powerful but unperfect module system **4**
1.1 OCaml modules in a nutshell . . . 4
1.2 The signature avoidance problem . . . 5
1.3 An F^{ω}-inspired solution . . . 6

2 A generative subset of the OCaml module system **9**
2.1 System structure and judgments . . . 10
2.2 Equivalence and Strengthening . . . 11
2.2.1 Equivalence . . . 11
2.2.2 Strengthening . . . 12
2.3 Subtyping . . . 13
2.4 Typing . . . 15

3 Canonical signatures and existential types **17**
3.1 The need for existential types . . . 17
3.2 System structure and judgments . . . 18
3.3 Typing and subtyping . . . 20
3.3.1 Subtyping . . . 20
3.3.2 Typing . . . 22
3.4 From the source to the canonical system . . . 23
3.4.1 Canonification revisited . . . 23
3.4.2 Canonification of lookup . . . 23
3.4.3 Canonification of equivalence . . . 24
3.4.4 Canonification of strengthening . . . 24
3.4.5 Canonification of subtyping . . . 25
3.4.6 Canonification of typing . . . 25

4 Anchorability conditions: the power of existentials into a path-based approach **26**
4.1 Anchorable wellformedness and typing . . . 26
4.1.1 Signature avoidance revisited . . . 26
4.1.2 Anchorable wellformedness . . . 27
4.1.3 Anchorable typing . . . 28
4.2 Anchorability result . . . 29
4.2.1 Anchored wellformedness . . . 29
4.2.2 Proof sketch . . . 31

5 Formal guaranties through elaboration in F^{ω} **33**
5.1 F-omega and encoded signatures . . . 33
5.2 The elaborated system . . . 33
5.2.1 System structure . . . 33
5.3 Subtyping . . . 37
5.4 Typing . . . 38
5.5 Correctness and link with the canonical system . . . 40
5.5.1 Correctness . . . 40
5.5.2 Link with the canonical system, backwards . . . 41
5.5.3 Link with the canonical system, forward . . . 41

Conclusion **43**

A OCaml source system **45**
A.1 Lookup . . . 45
A.2 Well-formedness . . . 45
A.3 Equivalence . . . 46
A.4 Strengthening . . . 47
A.5 Subtyping . . . 48
A.6 Typing . . . 50

B Canonical system **51**
B.1 Lookup . . . 51
B.2 Well-formedness . . . 51
B.3 Canonification . . . 52
B.4 Anchorable well-formedness . . . 53
B.5 Anchored well-formedness . . . 54
B.6 Subtyping . . . 57
B.7 Typing . . . 58

C Elaboration **60**
C.1 Lookup rules . . . 60
C.2 Wellformedness . . . 60
C.3 Canonification . . . 61
C.4 Subtyping . . . 64
C.5 Typing . . . 65

Introduction

This report constitutes the master thesis of the Computer Science Master of Science track at EPFL. It was done between February and August 2021 (partially remotely) at Inria Paris, under the direction of Mr Didier Rémy and Mr Gabriel Radanne. The EPFL academic supervision was provided for by Mr Viktor Kuncak. This report is structured as follows. The first section introduces the topic of OCaml modules, present some key challenges and explain our approach to solve them. The second section is a description of (a subset of) the OCaml module language and its mechanisms. The third section is devoted to present the canonical system and show how it extends the OCaml source system. The fourth section explores the link between the canonical and source presentations, specifically the necessary conditions to go be able to translate typing derivations from the former to the latter. In the fifth section, we detail the elaboration (translation) of the canonical terms and judgments into a polymorphic lambda calculus (F^{ω}). We conclude by summing up the theorems linking the three systems, and explaining how it serves a a basis for improving the OCaml module language.

```
(* A simple module to encapsulate
   integers *)
module Integers = struct
  type t = int
  let eq (x:t) (y:t) = (x=y)
end

(* A simple module to encapsulate
   complex numbers *)
module Complex = struct
  type t = {a: int; b:int}  (* a + i*b *)
  let eq (x:t) (y:t) =
    (x.a = y.a && x.b = y.b)

  (* Additionnal fields *)
  let real_part (x:t) = x.a
  let imaginary_part (x:t) = x.b
end

(* A generic interface of a module with
   a base type and an equality function *)
module type Comparable = sig
  type t
  val eq : t -> t -> bool
end
```

```
(* A set functor with signature
   ascription to abstract the set type *)
module Set = functor (A: Comparable) -> (
struct
  type t = A.t
  type set = t list
  let empty_set : set = []
  let rec add x s : set =
    match s with
    | [] -> [x]
    | y::s' ->
      match A.eq y x with
      | true -> s (* no duplicates *)
      | false -> y::(add x s')
end : sig
  type t
  type set
  val empty_set : set
  val add : t -> set -> set
end)

(* Integer sets *)
module IntegerSet = Set(Integer)
(* Complex sets *)
module ComplexSet = Set(Complex)
```

Figure 1: A classic example of functor usage.

1 A powerful but unperfect module system

In this introductory section, we first give an overview of the OCaml module system features and strengths. Then we explain the *signature avoidance* problem that ML modules systems suffer from, and consequent weaknesses. In the last subsection, we present our approach inspired by translations into F^{ω}, that solves the issues of the current OCaml module system while providing formal guaranties[1].

1.1 OCaml modules in a nutshell

Modularity is a key technique to break down a complex program into 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 reusable (typically, a data-structure)—effectively factorizing development—or just to mimic the overall designed structure of the program. Controlling the interactions between modules (through interfaces) is not only a tool for correctness, it also allows for several developers (or teams) to work independently on different parts of the same program.

A wide variety of techniques can be used to apply modularity concepts to software development, from simple macros to full fledged object-oriented programming (OOP). In the languages of the ML-family, modularity is provided by a module language layer built on top of the core language. The interactions between modules are controlled statically by the strict typing system, making modularity both practical and efficient at run-time (with little overhead). A module is described by its interface—called a *signature*–which serves as both a (light) specification and an API.

[1] As we will explain, our solution has only been studied for a *generative* subset of the language, but should hopefully extend to an *applicative* one.

The OCaml module system is especially rich; it provides *functors* as a way to build polymorphic modules that depend on other modules. Signatures can be used to restrict and control the outside view of a module, specifically by *hiding* fields (which correspond to the distinction between *public* and *private* fields in OOP), or by abstracting type components (making types accessible while hiding their definition). Abstract types are especially useful to control field access through typing, in a much lighter way than the *getters/setters* pairs in OOP.

A small example of module, signature, and functor usage is given in Figure 1. We first define two modules `Integer` and `Complex` that have both type fields and value fields. Then we write the module type (signature) `Comparable` we want a module to provide, so that we can build sets of values of its base type. We then define a simple `Set` functor, built on top of any comparable module (any module with a signature that is a subtype of `Comparable`) given as argument. We restrict the signature of `Set` to hide the implementation details (namely, the use of lists to represent sets) which effectively prevents users from adding elements to the list directly, which could break the no-duplicates invariant. Hence, elements can only be added to a set through the `add` function. We can then easily get sets of integers or of complex numbers by applying the functor to the appropriate module.

The OCaml module system is renowned for its expressiveness, ease to use, and the properties it can statically enforce. All sizable OCaml projects use modules to access libraries or define parametric instances of data-structures (sets, hashtables, streams, etc.). Several successful projects have made a heavy use of modules, such as MirageOs[2], where modules and functors are assembled on demand using a DSL (see Radanne et al. [2019]). However, giving a formal, type-theoretic definition of the module system and of its properties has proven to be a hard task. Before mentioning previous works on ML modules, we are going to present one key issue.

1.2 The signature avoidance problem

In this subsection we give an overview of the *signature avoidance* problem: a major difficulty in ML-module systems that can arise due to the complex interaction between the scoping of type variables and module operations (ascription, projection, functor call). At a high level, the issue comes from a mismatch between the expressiveness of the module language and of the signature language: it is possible to build modules that cannot be described by a signature. This mismatch between the *reachable space* of module expressions and the *describable space* of signatures is shown in Figure 2. Dealing with the signature avoidance problem can be done in two ways. First, one can be willing to ensure that the typechecker correctly covers the describable space and fails in the signature avoidance area. Secondly, the signature syntax can be extended to overlap with the reachable space.

This mismatch is caused by the interaction of three mechanisms. First, the ability to abstract types—which is key to use typing to control access and protect invariants—effectively creates new types, only compatible with (aliases of) themselves. Second, the sharing of those abstract types between modules, needed to make them interact, creates trees of type equalities. Finally, hiding types components (through ascription, subtyping) can remove abstract type aliases while other type components that are kept *in scope* may refer to them (for instance, removing the type `t` while keeping a type `t list`). In some situations, there are no possible signature to infer for a module. We present here the issue in more details.

When typing a module, type definitions and aliases can be represented as a dag of type equalities, as shown in Figure 3. The connected components of this dag 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 graph, separating connected components apart (splitting equivalence classes).

A simple example is given in Figure 3 (where modules only have type fields for simplicity). 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 dag): this becomes hidden to the typechecker outside of the body of

[2]https://mirage.io

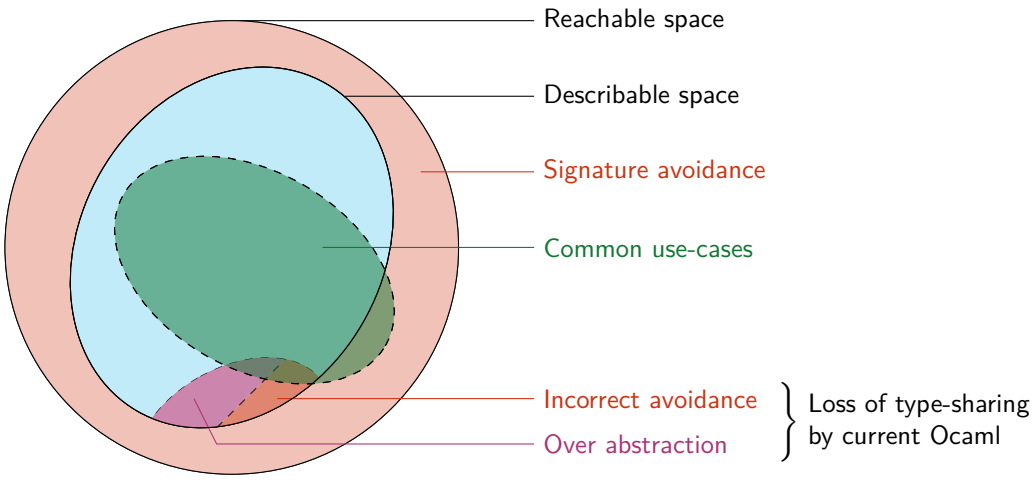


Figure 2: A representation of the mismatch between the *reachable space* of module expression (outer-most circle) and the *describable space* of signatures (inner ellipse). The common use-cases of OCaml are mainly within the area where the typechecker behaves correctly. In some cases, the current OCaml typechecker can lose type-sharing while being in the describable space, which can lead to either producing an over-abstract signature or failing at finding one

the module `X`. On the right-hand side, the projection on the `Y` submodule 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 loss of type-sharing or incorrect signatures.

Strategies for solving signature avoidance When a type alias is pointing to an out-of-scope type, there are mainly three strategies to correct the signature: abstracting the type (effectively ignoring the previous link in the dag), rewriting the type equalities using in-scope aliases (effectively rewiring the dag to maintain connected components), or extending the signature syntax (with existential types). The first strategy can lead to loss of type sharing. It is widely used by the OCaml typechecker. The cases where the second strategy—of finding the right aliases substitution to prevent loss of type sharing while staying *in-scope*—succeeds constitute the *solvable* cases of signature avoidance. The OCaml type-checker only tries to follow directed edges of the dag until it finds an accessible type, but does not follow reverse edges, and thus does not have a notion of *connected components*. In some cases, there are no *in-scope* aliases that can be used, 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 effectively adds 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 (notably, projection of a component of an unnamed module as done in Figure 3 is forbidden). However, it is sometimes cumbersome, which can limit the usability of module-based solutions such as modular implicits. It also prevents a more fine-grained management of accessible types.

1.3 An F^{ω}-inspired solution

Previous work has given solid formal foundations for ML-modules. The link between abstract types in ML-modules and existential types of F^{ω} was already explored by Mitchell and Plotkin [1985]. This vision was

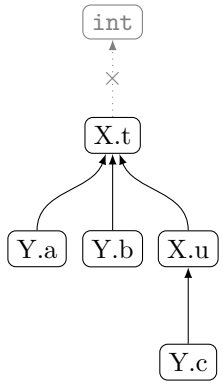
```
module M = struct

  module type S = sig
    type t
    type u = t
  end

  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 M = (struct

  module type S = sig
    type t
    type u = t
  end

  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
```

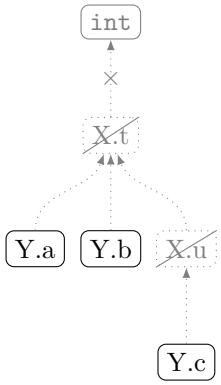


Figure 3: Two examples of modules and associated type-sharing dags.

opposed by MacQueen [1986], who considered existential types to be too weak, and proposed a restriction of dependent types (using strong sums) to describe module systems. Further work, notably on phase separation Harper et al. [1989], supported the idea that dependent types may actually be too powerful (thus, unnecessarily complex) for module systems. SML modules where first described by Harper et al. [1989]. Two approaches for the formalization and improvement of abstract types in SML where 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 in itself was specified by Leroy [2000], with an extension to applicative functors [Leroy, 1995].

Advances in the theory of existential types, notably with Russo [2004], opened the door for a simplified link between modules and F^ω. The type generativity (in the context of recursive modules) via existential types is explored in Dreyer [2007], using stamps. A similar, but logical approach was later developed by Montagu and Rémy [2009]. Pushing the idea further, a close correspondence between ML-modules and F^ω was achieved in Rossberg et al. [2014], where a significant subset of the SML module syntax is elaborated (translated) into a subset of F^ω. Typing and subtyping are defined and proven correct (with respect to F^ω) through elaboration, which made the elaborated system inherit the progress, preservation, and soundness properties. This work covered some difficult points like applicative functors and first-class modules, and was partially mechanized in Coq. Following the *F-ing* approach, Rossberg [2018] achieved a unification of the core and module languages (thus, unstratified). 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 similarities with *F-ing*.

At the beginning of the master thesis, our goal was to apply the *F-ing/1ML* approach to OCaml modules (which have some specific features). Using the elaboration into F^ω as a guide, we wanted to give a clean presentation of the sets of rules needed to define typing (and subtyping) for OCaml modules. Although very insightful for language designers, *F-ing/1ML* only defined typing by elaboration, not writing rules inside of the source system. Building a type-checker using elaboration would make inferred terms and signatures F^ω terms and types, which would (ambitiously) require the users to be able to translate them back, on the fly, into the source syntax.

Instead, we wanted to give a set of source rules, along with their elaboration into F^ω, to benefit both from the formal guaranties of the latter, while staying close to the intuition of the source. This turned out

to be hard, as some mechanisms of the OCaml modules translate into F^ω in a non trivial way. This lead us to consider a middle point, an hybrid system called *canonical* that lightly extends the source syntax with some F^ω inspired constructs (namely, existential types). This intermediary system is both easy to elaborate into F^ω with an *F-ing* approach and close to the traditional OCaml source presentation. As we wanted to show the potential of this system as a basis for a new foundation of OCaml modules, we tried to explicit the link between the OCaml source presentation, the canonical system, and the elaboration into F^ω.

The rest of this document will follow this approach. In the next section, we present a (generative) subset of the OCaml module language with its grammar and all the necessary judgments. In the following section, we present the canonical system, that is both an extension and a simplified version of the source one. We sketch the proof that all source derivations can be expressed in the canonical one. Then, in the following section, we explore the conditions necessary to be able to write a canonical derivation back into the source system, effectively exposing why the source system is less expressive and more cumbersome. Finally, in the last section, we present an elaboration of the canonical system into F^ω. For all systems, the full set of inference rules are given in appendices.

2 A generative subset of the OCaml module system

In Figure 4, we present a grammar for a generative subset of the module language, built on top of an (omitted) core language. The two main parts are the *module expressions* (which combine in different ways module *structures*) and *signatures* (also referred to as *module types*). A module structure is made of a tree of *bindings*, whereas a signature structure contains a list of *declarations*, where the concatenation is associative and the empty list is written ϵ.

Self-references Both structures and signatures have an alpha-convertible *self-reference*, written in subscript, that is used to refer to the current object. This simplifies how scoping and shadowing of variables is expressed in our rules. It is not present in current OCaml syntax, but could be easily added to the source by a *pre-typechecking* phase.

Some restrictions are built-in the grammar. Specifically, functor application and ascription can only be done on paths while projection however, can be done on any module expression. We also distinguish between four kinds of identifiers: X for modules, A for module type variables, x for values and t for types. As paths are defined only with module identifiers X, the grammar prevents field access inside module types. More importantly, as paths cannot contain functor calls, the type system cannot track equality of types created by functors. This restriction makes the system only *generative*, which simplifies the presentation. We plan to expand our presentation to *applicative* functors (with sharing of types created by functor applications) in future work.

Path

$P ::= X \mid P.X$	(no functor application)

Module Expressions

$M ::= P$	(Variables)
$\mid M.X$	(Projection)
$\mid (P : S)$	(Sealing)
$\mid P_1(P_2)$	(Functor application)
$\mid (X : S) \to M$	(Functor)
$\mid \texttt{struct}_Y\ B\ \texttt{end}$	(Structure)

Bindings

$B ::= \mathsf{let}\, x = E$	(Values)
$\mid \mathsf{type}\, t = T$	(Types)
$\mid \mathsf{module}\, X = M$	(Modules)
$\mid \mathsf{module\ type}\, A = S$	(Module types)
$\mid B; B$	(Sequence)
$\mid \varepsilon$	(Empty)

Core language

$E ::= P.x$	(Qualified variable)
$\mid \ldots$	(Other expressions)
$T ::= P.t$	(Qualified type)
$\mid Op\,(T_1, \ldots, T_n)$	(Type operation)
$\mid \ldots$	(Other types)

Signatures

$S ::= P.A$	(Variables)
$\mid (X : S_1) \to S_2$	(Functor)
$\mid \texttt{sig}_Y\ \overline{D}\ \texttt{end}$	(Signature)

Declarations

$D ::= \mathsf{val}\, x : T$	(Values)
$\mid \mathsf{type}\, t = T$	(Types)
$\mid \mathsf{type}\, t$	(Abstract types)
$\mid \mathsf{module}\, X : S$	(Modules)
$\mid \mathsf{module\ type}\, A = S$	(Module types)

Environments

$\Gamma ::= \varepsilon$	(Empty)
$\mid \Gamma, (\mathsf{module}\, X : S)$	(Functor Argument)
$\mid \Gamma, Y.D$	(Declaration)

Identifiers

$Z ::= A \mid X \mid x \mid t$	(Any identifier)

Figure 4: Syntax of the module language

In addition to restricting the system to be generative, we omitted some constructs for the sake of simplicity. Specifically, the include operator, the explicit constraints S with type $t = \ldots$ (and S with module $X = \ldots$) and their counterparts, the deleting constraints S with type $t := \ldots$ (and S with module $X := \ldots$) are omitted. We believe that they do not impact the overall structure of the system, only adding more cases in the set of rules.

As we mentioned, we chose to allow projection on any module expression, but disallow functor application on anything else than paths. The current OCaml syntax does the opposite, mainly to prevent cases prone to trigger signature avoidance. The two choices are actually equivalent: as long as either projection or functor call can be done on unnamed modules, the other one becomes syntactic sugar.

2.1 System structure and judgments

The judgments and their dependencies are presented in Figure 5. The system we present revolves around two main components: *typing* and *subtyping*. In addition, a *wellformedness* predicate that checks freshness of names and scope of variables is used. The *path-based* representation of type sharing in OCaml modules requires us to define two more judgments: *strengthening* and *equivalence*. Strengthening is used to keep track of type sharing between module aliases. Equivalence ensures that two signatures use the same types, up to a substitution of aliases: two equivalent signatures have the same connected components in their type-sharing dag.

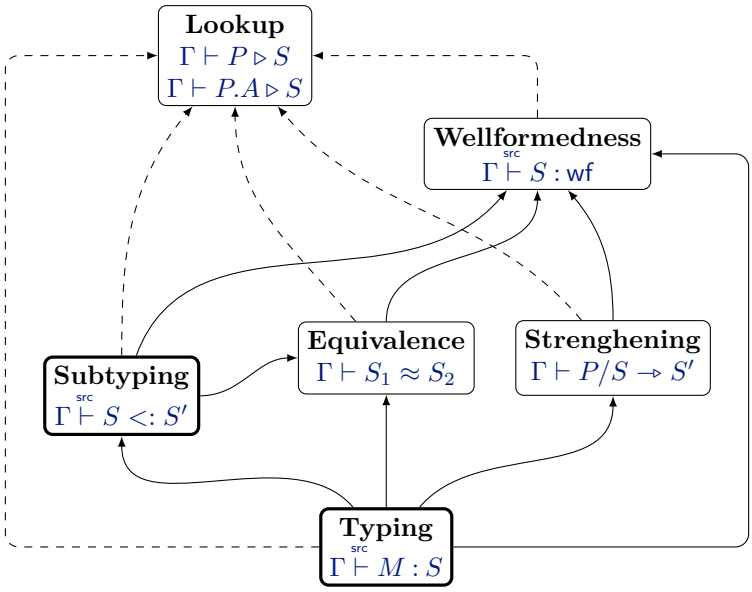


Figure 5: Structure of judgments for the source system

Lookup In the OCaml module system, the *lookup* operation $\Gamma \vdash P \triangleright S$ is central and more involved than a simple access to the environment. Indeed, declared variables (and their types/signatures) are not individually stored in the context, but packaged inside modules. Retrieving the signature of given path with multiple levels of depth (like $P.X.Y$) thus requires a lookup operation capable of inspecting intermediary signatures and accessing submodule fields. In addition, as signatures can be module types variables (like $P.A$), the lookup operation must be able to recursively follow those aliases. We show here two examples of lookup rules. The rule S-Lkp-Alias does the recursive lookup of module type variables: $P'.A$ is a valid signature of P in Γ, but some judgments require the lookup to return the original definition of the module type (either some signature `sig` ... `end` or functor signature $(X : \ldots) \to \ldots$). The rule S-Lkp-Proj-Mod accesses a

submodule X of a module at P. The premise forces the signature returned by the lookup of P to be of the form $\texttt{sig}\ \ldots, \mathsf{module}\, X : S, \ldots\ \texttt{end}$), which can be done using C-Lkp-Alias recursively. Finally, a substitution of the self-reference Y is made in the result to transform relative paths into absolute ones.

$$\textsc{S-Lkp-Proj-Mod}\quad \frac{\Gamma \vdash P \triangleright \texttt{sig}_Y\ \overline{D}\ \texttt{end} \qquad (\mathsf{module}\, X : S) \in \overline{D}}{\Gamma \vdash P.X \triangleright S[Y \mapsto P]} \qquad \textsc{S-Lkp-Alias}\quad \frac{\Gamma \vdash P \triangleright P'.A \qquad \Gamma \vdash P'.A \triangleright S}{\Gamma \vdash P \triangleright S}$$

As presented here, the lookup operation $\Gamma \vdash P \triangleright S$ is thus more involved than checking if some element is in the environment (simply written as $x \in \Gamma$). The latter only browse the environment as a list, whereas the former can both follow module type aliases and use projection to access module and module type fields. As all judgments manipulate signatures (wellformdness of signatures, equivalence between signatures, strengthening of a signature, and subtyping between signatures), they all rely on the lookup, making it central to the presentation. The full set of lookup rules is given in Appendix A.1.

Wellformedness A wellformedness predicate is defined over signatures $\Gamma \overset{\text{src}}{\vdash} S : \mathsf{wf}$, declarations $\Gamma \overset{\text{src}}{\vdash}_Y D : \mathsf{wf}$, and environments $\Gamma : \mathsf{wf}$ to ensure two properties. First, all used variables should be bound in the environment (informally, $\Gamma \overset{\text{src}}{\vdash} S : \mathsf{wf}$ implies $\texttt{fv}(S) \subseteq \Gamma$). Secondly, no shadowing is allowed inside the environment: for instance, when checking a field with identifier Z inside a signature of self-reference Y, no field should already be bound to $Y.Z$ in the environment. This is typically the purpose of the following rules:

$$\textsc{S-Wf-Val}\quad \frac{\Gamma \overset{\text{src}}{\vdash} T : \mathsf{wf} \qquad Y.x \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_Y \mathsf{val}\, x : T : \mathsf{wf}} \qquad \textsc{S-Wf-Type}\quad \frac{\Gamma \overset{\text{src}}{\vdash} T : \mathsf{wf} \qquad Y.t \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_Y \mathsf{type}\, t = T : \mathsf{wf}} \qquad \textsc{S-Wf-TypeAbs}\quad \frac{\Gamma : \mathsf{wf} \qquad Y.t \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_Y \mathsf{type}\, t : \mathsf{wf}}$$

In addition, the wellformedness of module types variables require the lookup operation to succeed and the result of the lookup to be wellformed. This effectively ensures that all prefixes of the path can be looked up to a signature containing a submodule field with the identifier of the next hop : given a path $X.Y.Z\ldots$, the signature of X can be looked up to some $\texttt{sig}\ \overline{D}\ \texttt{end}$ containing a module field Y, the signature of $X.Y$ can be looked up to some $\texttt{sig}\ \overline{D}'\ \texttt{end}$ containing a module field Z, etc. This is done by the rule S-Wf-Path.

$$\textsc{S-Wf-Path}\quad \frac{\Gamma \vdash P.A \triangleright S \qquad \Gamma \overset{\text{src}}{\vdash} S : \mathsf{wf}}{\Gamma \overset{\text{src}}{\vdash} P.A : \mathsf{wf}}$$

The full set of wellformedness rules is given in Appendix A.2.

2.2 Equivalence and Strengthening

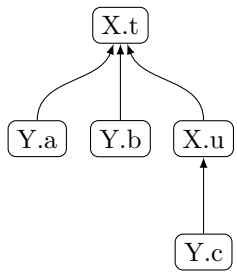
In the OCaml module system, two additional judgments are needed to manipulate signatures and types in order to keep type sharing information.

2.2.1 Equivalence

As we saw in the Section 1.2, the type-sharing dag that underlies a modular definition can be of any complexity and depth. Therefore, checking if two types are equal (belonging to the same connected component of the graph) is an involved operation that requires to follow type aliases and lookup signatures. We define an equivalence relationship on types (reflexive, symmetric, transitive) to capture this idea: $\Gamma \vdash T_1 \approx T_2$. Two rules are given to extract type definitions from the context (with a translation from local to absolute paths when necessary) :

$$\textsc{S-Eqv-Env}\quad \frac{Y.(\mathsf{type}\, t = T) \in \Gamma}{\Gamma \vdash Y.t \approx T} \qquad \textsc{S-Eqv-Lkp}\quad \frac{\Gamma \vdash P \triangleright \texttt{sig}_Y\ \overline{D}\ \texttt{end} \qquad (\mathsf{type}\, t = T) \in D}{\Gamma \vdash P.t \approx T[Y \mapsto P]}$$

```
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
```

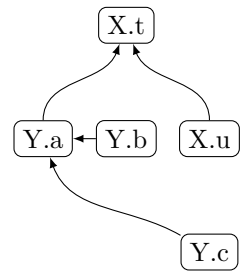


Figure 6: Two equivalent signatures S_1 and S_2 : $\emptyset \vdash S_1 \approx S_2$. They define different type sharing dags that have the same connected components. Specifically, two equivalent types in S_1 are also equivalent in S_2 and conversely

Then, we add the rules for reflexivity, symmetry, transitivity, congruence:

S-Eqv-Refl
$$\Gamma \vdash T \approx T$$

S-Eqv-Sym
$$\frac{\Gamma \vdash T' \approx T}{\Gamma \vdash T \approx T'}$$

S-Eqv-Trans
$$\frac{\Gamma \vdash T_1 \approx T_2 \qquad \Gamma \vdash T_2 \approx T_3}{\Gamma \vdash T_1 \approx T_3}$$

S-Eqv-Cgr
$$\frac{\forall i \in [\![0, n]\!],\ \Gamma \vdash T_i \approx T'_i}{\Gamma \vdash Op(T_0, \ldots, T_n) \approx Op(T'_0, \ldots, T'_n)}$$

From the relationship on types, we can define a relationship on declarations, and on whole signatures: $\Gamma \vdash S_1 \approx S_2$. Two signatures are equivalent if we can go from the first to the second only by substituting type aliases. It means that they describe types sharing dags that have the same connected components, as for instance in the example of Figure 6. The full set of equivalence rules is given in Appendix A.3.

2.2.2 Strengthening

The need for strengthening of a signature comes from the aliasing of modules. When we alias a module, copying its signature effectively copies the type sharing dag, but without an explicit link to the original module. If the type sharing dag is not *anchored* by points outside of the module—that is, if the module defines some abstract types—then type sharing information is lost between fields of a module and its alias. A simple example is given in Figure 7.

```
(* Original module *)
module M : sig
  type t (* abstraction *)
  type u = t
  val  x : t
end = struct
  type t = int
  type u = t
  let  x = 42
end
```

```
(* Without strengthening *)
module N = M
(* :
sig
  type t
  type u = t
  val  x : t
end
*)
```

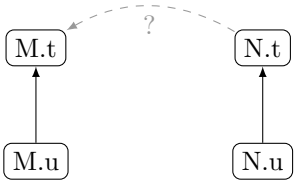


Figure 7: Example of a loss of type sharing between a module and its alias when no strenghening is used. Here, the values $M.x$ and $N.x$ are not even comparable, let alone beeing equal.

In the example of Figure 7, without strengthening, the module `N` gets the same signature as the module `M`: `sig type` t , `type` $u = t$ `end`. It makes the type `N.t` abstract, and thus, incompatible with `M.t`. To keep the type-sharing information, `N` gets the signature `S` *strengthened* by `M`, as shown in Figure 8.

```
(* Original module *)
module M : sig
  type t (* abstraction *)
  type u = t
  val  x : t
end = struct
  type t = int
  type u = t
  let  x = 42
end
```

```
(* With strengthening *)
module N = M
(* :
sig
  type t = M.t
  type u = t
  val  x : t
end
*)
```

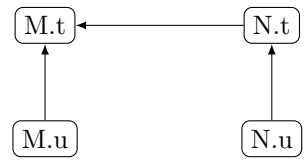


Figure 8: Example of type sharing beeing kept between a module and its alias by strenghening the signature of M is used. Here, the values $M.x$ and $N.x$ are of the same type $M.t$ (and equal).

The key insight is the semantic difference between a declaration of a type $\mathsf{type}\, t = T$ and of an abstract type $\mathsf{type}\, t$. An abstract type definitions state the *existence* and *identity* of the type, whereas concrete type definitions only state the *identity*, without creating a new type. Understanding where types are created and, in signature avoidance cases, changing the *defining instance* of an abstract type is one the main challenges of the source presentation that gets solved by the canonical approach of next section. The *identity*/*existence* duality is explored in more details in the Section 4.

We define the strengthening operation $\Gamma \vdash P/S \rightarrow S'$ of a signature S by a path P, inside an environment Γ, producing another signature S'. This operation is defined only when the signature we strengthen is that *of the module* P, that is, if $\Gamma \vdash P \triangleright S$. The key rule is S-Lkp-TypeAbs, the other ones are mainly to do a depth-first traversal of the signature. Module type variables are not strengthened, as no access can be done in them ($P.A.t$ is not a valid path).

S-Str-TypeAbs
$$\Gamma \vdash_Y (\mathsf{type}\, t)\, /P \rightarrow \mathsf{type}\, t = P.t$$

S-Str-Type
$$\Gamma \vdash_Y (\mathsf{type}\, t = T)\, /P \rightarrow \mathsf{type}\, t = T$$

S-Str-Val
$$\Gamma \vdash_Y (\mathsf{val}\, x : T)\, /P \rightarrow \mathsf{val}\, x : T$$

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

S-Str-ModType
$$\Gamma \vdash_Y (\mathsf{module\ type}\, X = S)\, /P \rightarrow \mathsf{module\ type}\, X = S$$

The full set of rules is given in Appendix A.4.

2.3 Subtyping

The subtyping judgement has a fundamental role in the OCaml module system, as it allows the user to define abstract, polymorphic interfaces and use modules that have a richer signature than those interfaces. For example, functors are defined with an argument that has a given signature, but can be called on any module, as long as it provides the required fields, regardless of whether or not it has additionnal fields. This has significant consequences in terms of compilation of modules. We distinguish between two modes of subtyping:

- subtyping with deletion and reordering of fields : $\Gamma \overset{\mathsf{src}}{\vdash} S_1 <: S_2$
- sybtyping by abstraction only : $\Gamma \overset{\mathsf{src}}{\vdash} S_1 \prec: S_2$

Their key difference comes from considerations of compilation: abstraction only subtyping is *code free* (it does not change values, just types), whereas normal subtyping requires a special action from the compiler (as it allows removal or reordering or value fields).

Declaration subtyping At high level, the subtyping operation betwen S_1 and S_2 checks that S_1 defines all the fields of S_2, and that, recursively, all field declarations (values, types, modules, and module-types) of S_1 are subtypes of the ones of S_2. The field-by-field comparison (declaration subtyping) is straigthforward (and the same in the two modes of subtyping), as we can see in the following rules :

S-Sub-Mod
$$\frac{\Gamma \overset{\text{src}}{\vdash} S <: S'}{\Gamma \overset{\text{src}}{\vdash}_Y (\mathsf{module}\, X : S) <: (\mathsf{module}\, X : S')}$$

S-Sub-Val
$$\frac{\Gamma \overset{\text{src}}{\vdash} T_1 <: T_2}{\Gamma \overset{\text{src}}{\vdash}_Y (\mathsf{val}\, x : T_1) <: (\mathsf{val}\, x : T_2)}$$

S-Sub-TypeAbs
$$\Gamma \overset{\text{src}}{\vdash}_Y (\mathsf{type}\, t) <: (\mathsf{type}\, t)$$

S-Sub-Type
$$\frac{\Gamma \overset{\text{src}}{\vdash} T_1 <: T_2}{\Gamma \overset{\text{src}}{\vdash}_Y (\mathsf{type}\, t = T_1) <: (\mathsf{type}\, t = T_2)}$$

In addition to those rules, we allow types definitions to be subtypes of abstract types. This is key to define abstract interfaces, like typically functors that are agnostic as to the type fields defined by their argument.

S-Sub-TypeToAbs
$$\frac{\Gamma \overset{\text{src}}{\vdash} T : \mathsf{wf}}{\Gamma \overset{\text{src}}{\vdash}_Y (\mathsf{type}\, t = T) <: (\mathsf{type}\, t)}$$

Finally, as module type fields can be used in both covariant and contra-variant positions (as functor arguments for example), the rule for subtyping a module type declaration is a bit more involved :

S-Sub-ModType
$$\frac{\Gamma \overset{\text{src}}{\vdash} S <: S' \qquad \Gamma \overset{\text{src}}{\vdash} S' <: S}{\Gamma \overset{\text{src}}{\vdash}_Y (\mathsf{module\ type}\, A = S) <: (\mathsf{module\ type}\, A = S')}$$

Module type subtyping At the module type level, the subtyping checks that there is a structural match (up to a lookup operation): functor signature against functor signature and signature against signature. The rule for the latter is where the distinction between the two modes of subtyping appears: for the abstraction-only subtyping, the two lists of declarations $\overline{D}$ and $\overline{D'}$ are compared directly, whereas in the normal mode of subtyping, a subset $\overline{D}_0$ of $\overline{D}$ list is compared with $\overline{D'}$, allowing for reordering and deletion. Using a subset and not a subsequence is crucial as it allows for reordering.

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

S-Sub-Sig-Eq
$$\frac{\Gamma \overset{\text{src}}{\vdash}_Y \overline{D} : \mathsf{wf} \qquad \overline{\Gamma, \overline{Y.D} \overset{\text{src}}{\vdash}_Y D \prec: D'} \qquad \Gamma \overset{\text{src}}{\vdash}_Y \overline{D'} : \mathsf{wf}}{\Gamma \overset{\text{src}}{\vdash} \mathtt{sig}_Y\ \overline{D}\ \mathtt{end} \prec: \mathtt{sig}_Y\ \overline{D'}\ \mathtt{end}}$$

In both cases, the subtyping is done *declaration by declaration* (not as a telescope), in an environment that contains all the declarations of the richer (left-hand side) signature. As usual, the subtyping is contra-variant for functor arguments, as we can see in the rule S-Sub-Fct.

S-Sub-Fct
$$\frac{\Gamma \overset{\text{src}}{\vdash} S'_a <: S_a \qquad \Gamma, \mathsf{module}\, X : S_a \overset{\text{src}}{\vdash} S_r <: S'_r \qquad X \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash} (X : S_a) \to S_r <: (X : S'_a) \to S'_r}$$

The full set of subtyping rules is given in Appendix A.5.

2.4 Typing

Once all the technical judgments have been layed out, the typing of modules is pretty straightforward. Similarly to judgments that are defined on declarations and on signatures, typing is defined on bindings and on module expressions.

Bindings typing The typing of bindings relies on the wellformedness judgement, on a core language typing operation and on the typing of modules, recursively (for submodules). The bindings rules are as follows:

S-Typ-Let
$$\frac{\Gamma \overset{\text{src}}{\vdash} E : T \qquad Y.x \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_Y (\mathsf{let}\, x = E) : (\mathsf{val}\, x : T)}$$

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

S-Typ-Type
$$\frac{\Gamma \overset{\text{src}}{\vdash} T : \mathsf{wf} \qquad Y.t \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_Y (\mathsf{type}\, t = T) : (\mathsf{type}\, t = T)}$$

S-Typ-ModType
$$\frac{\Gamma \overset{\text{src}}{\vdash} S : \mathsf{wf} \qquad Y.A \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_Y (\mathsf{module\ type}\, A = S) : (\mathsf{module\ type}\, A = S)}$$

This set of rules is completed by two rules for sequences of declarations:

S-Typ-Empty
$$\frac{\Gamma : \mathsf{wf}}{\Gamma \overset{\text{src}}{\vdash}_Y \varepsilon : \varepsilon}$$

S-Typ-Seq
$$\frac{\Gamma \overset{\text{src}}{\vdash}_Y B_1 : \overline{D}_1 \qquad \Gamma, Y.\overline{D}_1 \overset{\text{src}}{\vdash}_Y B_2 : \overline{D}_2}{\Gamma \overset{\text{src}}{\vdash}_Y B_1; B_2 : \overline{D}_1 +\!\!+ \overline{D}_2}$$

Module typing The typing of modules contains both syntax-directed and free floating rules. This could be converted into a syntax-directed set of rules, but our goal here is to give a specification, not an implementation. An algorithmic presentation equivalent to this one would be more complex and less readable, which could hide the key insights of the separation of operations (lookup, wellformedness, equivalence, strenghtening, subtyping, typing). As we will see in the next section, the canonical model can inspire algorithmic solutions for the source typing. The syntax directed rules are:

S-Typ-Var
$$\frac{\Gamma \vdash P \triangleright S \qquad \Gamma : \mathsf{wf}}{\Gamma \overset{\text{src}}{\vdash} P : S}$$

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

S-Typ-App
$$\frac{\Gamma \overset{\text{src}}{\vdash} P_f : (X : S_a) \to S_r \qquad \Gamma \overset{\text{src}}{\vdash} P : S \qquad \Gamma \overset{\text{src}}{\vdash} S <: S_a}{\Gamma \overset{\text{src}}{\vdash} P_f(P) : S_r[X \mapsto P]}$$

S-Typ-Fct
$$\frac{X \notin \Gamma \qquad \Gamma; \mathsf{module}\, X : S_a \overset{\text{src}}{\vdash} M : S_r}{\Gamma \overset{\text{src}}{\vdash} ((X : S_a) \to M) : (X : S_a) \to S_r}$$

S-Typ-Struct
$$\frac{\Gamma \overset{\text{src}}{\vdash}_Y B : \overline{D} \qquad Y \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash} \mathtt{struct}_Y\ B\ \mathtt{end} : \mathtt{sig}_Y\ \overline{D}\ \mathtt{end}}$$

The rules that could trigger *signature-avoidance* situations (S-Typ-App and S-Typ-Sig) are made simple by their restriction to paths. However, as the grammar allows projection on any module expression (which could trigger signature avoidance), we need the following—more involved—rule:

S-Typ-Proj
$$\frac{\Gamma \overset{\text{src}}{\vdash} M : \mathtt{sig}_Y\ \overline{D}_1, \mathsf{module}\, X : S, \overline{D}_2\ \mathtt{end} \qquad \Gamma, \overline{D}_1 \overset{\text{src}}{\vdash} S \prec: S' \qquad \Gamma \overset{\text{src}}{\vdash} S' : \mathsf{wf}}{\Gamma \overset{\text{src}}{\vdash} M.X : S'}$$

In this rule, the abstraction subtyping allows for cutting links to types that become inaccessible, so that the resulting signature S' can be wellformed *outside* of the signature of M, that is, without relying on fields declared in $\overline{D}_1$—thus preventing any escaping of scope. This rule is however permissive, as we allow to abstract more than necessary and lose type sharing information. The condition to prevent loss of type sharing is complex to write down whithout introducing new concepts (but will be easy to express in the canonical system). Informally, we want that any two types that are in different connected components in S' are be already in different connected components in S.

Free-Floating rules Before doing a projection $M.X$, there could typically be a need to *rewire* the type sharing dag, so that types used in X point to other aliases (either external—accessible outside of M—or local ones—accessible inside of X). Similarly, strenghtening is sometimes required to keep sharing information between aliases. This leads to the following two *free-floating* rules :

S-Typ-Strengthen

$$\frac{\Gamma \overset{\text{src}}{\vdash} P : S \qquad \Gamma \vdash S/P \rightarrowtriangle S'}{\Gamma \overset{\text{src}}{\vdash} P : S'}$$

S-Typ-Equiv

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

Notice, however, that the subtyping operation is not free floating: it would allow for *code-not-free* operations (deletion and reordering) of fields at any point of the typing. As we mentionned in the subsection on subtyping, this has a significant impact on compilation, as it requires a special handling by the compiler. The full set of typing rules is given in Appendix A.6.

3 Canonical signatures and existential types

As we have seen, some of the issues of the source presentation come from the ambiguity and weaknesses of the path-based type-sharing. In this section, we present a module system with an extended module types (signature) syntax inspired by previous work on representation of modules in F^{ω}. This *canonical* system has a smaller set of judgments, and its typing 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 the same amount of type sharing (or more). Finally, we give the conditions at which a typing derivation in the canonical system can be translated back into a typing derivation in the source system. This provides a way of understanding complicated cases of typing in the source by a light elaboration into the canonical system.

3.1 The need for existential types

In the source presentation, two ad-hoc techniques were needed to prevent signature avoidance. First, we manipulate the equivalence classes (rewirering inside connected components of the type dag) with the equivalence judgement through paths rewriting—which is fragile and cumbersome. Second, we allowed abstraction of types through subtyping, to lose links to out-of-scope types. This is required when the declaration creating the abstract type becomes out of scope, but can lead to loss of type equalities.

A path-independent solution As we saw in the source presentation, maintaining the connected components of the type sharing dag throughout module operations where field can be deleted is hard and can require rewriting signatures (through equivalence). However, the information we are actually interested in is not the type sharing dag itself, but only its connected components. By introducing an existential type, we give a canonical representative to the whole equivalence class, which become insensible to the removal of some of its members. In the type sharing dag, it can be seen as an additional point with a special status (see Figure 9).

```
module M = struct

  module X : sig
    type t
    type u = t
  end = struct
    type t = int
    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
```

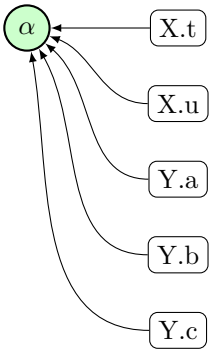


Figure 9: We extend the signature of the module M with an existential type α that acts as a canonical representative for all aliases of $X.t$. Doing so, the type-sharing dag of M becomes trivial. Unlike in Figure 3 the removal of some fields (through projection for instance) does not affect the connected component.

Identifying where abstract types are created Existential types are not only a technical solution to represent abstract types, they also convey the logical meaning and intuition. First, as with existential binding

in mathematics, existential types are alpha-convertible. As they serve as representatives of equivalence classes, it is logical that their name does not matter[3]. Second, as with the logical binder, no introspection is possible with an existential type: from the point of view of an user of a module, an abstract type just *exists*. Preventing introspection is crucial to use abstract types to maintain module invariants. As we saw, using existential types not only gives a path independent way to refer to an equivalence class, but it also flatten the type sharing dag to a single level of depth, which will simplifies the lookup operation. Finally, by introducing existential types, we expose at the signature level how many abstract types are created by the module, regardless of which name was used first to define it.

3.2 System structure and judgments

As existential types are canonical representatives of equivalence classes, signatures that use them are no longer valid only "up to an equivalence relation" but truly canonical. By extension, we call the system based upon them *canonical*. In this subsection we are going to give its grammar and the structure of its judgments.

Canonical Types	
$\tau ::= \alpha$	(Existential identifier)
$\mid Op\,(\tau_1, \ldots, \tau_n)$	(Type operation)
Environments	
$\Gamma ::= \varepsilon$	(Empty)
$\mid \Gamma\,, \overline{\alpha}$	(Abstract types)
$\mid \Gamma\,, (\mathsf{module}\, X : \mathcal{R})$	(Functor Argument)
$\mid \Gamma\,, (Y.\mathcal{D})$	(Declaration)
Canonical abstract signatures	
$\mathcal{S} ::= \exists\overline{\alpha}.\mathcal{R}$	(Abstract signature)
Canonical manifest signatures	
$\mathcal{R} ::= \forall\overline{\alpha}.\,(X : \mathcal{R}) \to \mathcal{S}$	(Functor)
$\mid \mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end}$	(Signature)
Canonical declarations	
$\mathcal{D} ::= \mathsf{val}\, x : (\approx \tau)$	(Values)
$\mid \mathsf{type}\, t \approx \tau$	(Types)
$\mid \mathsf{module}\, X : \mathcal{R}$	(Modules)
$\mid \mathsf{module\ type}\, A = \mathcal{S}$	(Module types)

Figure 10: Syntax extensions of the *canonical* module language

The grammar of the language, given in Figure 10, extends the OCaml grammar of Figure 4. The source syntax (for modules and signatures) remains the same, but new syntactical categories are introduced to represent the types and signatures in a canonical form. We distinguish between *abstract* canonical signatures $\mathcal{S}$ that specify the list of existential types created by the module with an existential binder, and *manifest* canonical signatures $\mathcal{R}$ that only refer to existential types of the context. The grammar of canonical signatures does not allow for module type variables. Signatures of functors use a forall quantifier in front, which comes from the contra-variant position of the existential quantification of the argument signature. This forall quantification in front $\forall\overline{\alpha}.\,(X : \mathcal{R}) \to \mathcal{S}$ should be understood as *for any list of types* $\overline{\alpha}$ *defined by the argument, the body has a signature* $\mathcal{S}$. As we will see in the typing rules, the existential binding is always done at the highest level, in front of the signature: inside a manifest signature, all submodules have manifest signatures (without existential binding).

The key idea of the canonical system is to express the typing, subtyping, and wellformedness predicates using existential types and canonical signatures. As we will see, this removes the need for equivalence and strengthening, and simplifies the overall presentation. However, to transform source signatures into canonical ones, a new judgement, *canonification* is introduced. The system structure is given in figure Figure 11.

[3]This is a typical issue if one tries to build a solution to the signature avoidance problem by adding missing type fields: they would not be alpha-convertible, and thus equivalent signatures could become incompatible (not subtype of one another)

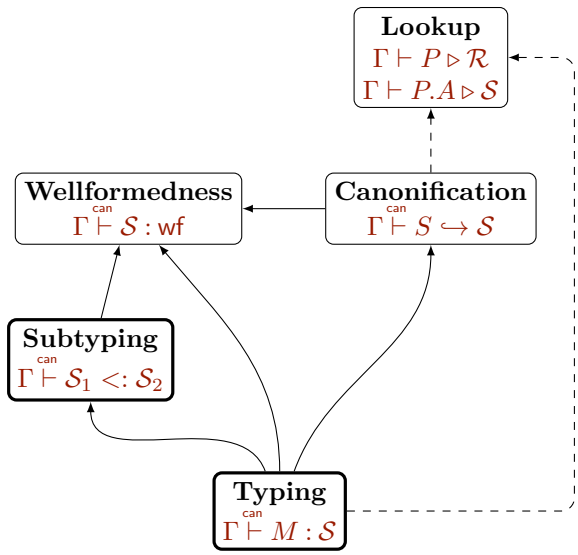


Figure 11: Structure of judgments for the canonical system

Lookup The lookup operation is made simpler by the absence of module type variables in the canonical syntax, which removes the need for follow-up of aliases. However, as in the source presentation, there are recursive rules to access submodules and inspect intermediary structures (such as C-Lkp-Proj-Mod given below). As constrained by the grammar, the lookup of a module type variable $P.A$ returns an abstract signature $\mathcal{S}$, but the lookup of a module path always returns a manifest signature $\mathcal{R}$. The intuition being that all the abstract types created when typing some module saved at a path P are pushed as individual components $\overline{\alpha}$ in the environment before the signature of P, and thus, no abstract types are *created* when using the module P. The full set of rules is given in Appendix B.1.

$$\text{C-Lkp-Proj-Mod}\quad \frac{\Gamma \vdash P \triangleright \mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} \qquad (\mathsf{module}\, X : \mathcal{R}) \in D}{\Gamma \vdash P.X \triangleright \mathcal{R}}$$

Wellformedness As in the source presentation, we use a wellformedness predicate on canonical signatures to ensure that no free variables and only fresh names are used. The judgement does not depend on the lookup, as there are no module type variables and as types are in a canonical form[4]. Existential types binders are delt with by rules like C-Wf-Abs and C-Wf-Funct and otherwise, the judgment is similar to the source one. The full set of rules is given in Appendix B.2.

$$\text{C-Wf-Abs}\quad \frac{\Gamma, \overline{\alpha} \overset{\text{can}}{\vdash} \mathcal{R} : \mathsf{wf}}{\Gamma \overset{\text{can}}{\vdash} \exists\overline{\alpha}.\mathcal{R} : \mathsf{wf}} \qquad \text{C-Wf-Funct}\quad \frac{\Gamma \overset{\text{can}}{\vdash} \exists\overline{\alpha}.\mathcal{R} : \mathsf{wf} \qquad \Gamma, \overline{\alpha}, \mathsf{module}\, X : \mathcal{R} \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf} \qquad X \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash} \forall\overline{\alpha}.\,(X : \mathcal{R}) \to \mathcal{S} : \mathsf{wf}}$$

Canonification We introduce a new judgement to convert source signatures into their canonical form: $\Gamma \overset{\text{can}}{\vdash} S \hookrightarrow \mathcal{S}$. The judgement is first given for converting types components into canonical ones. As the environment is in canonical form, all type paths $P.t$ are bound to some canonical type τ. The following set of rules allows one to obtain the canonical version of a type by getting its declaration (either locally with

[4]If canonical wellformedness rules do not use the lookup $\Gamma \vdash P \triangleright \mathcal{R}$, some rules do however check if some element is (or not) in the environment (such as $\alpha \in \Gamma$, $Y.ix \notin \Gamma$, etc.). This is weaker than the lookup operation, as all checked names are saved *as is* in the environment, without the need for projection inside saved structures.

C-Cf-LocalType or through a path with C-Cf-Type) and combining with congruence with C-Cf-Cgr.

$$\textsc{C-Cf-LocalType}\quad \frac{\Gamma : \mathsf{wf} \qquad \mathsf{type}\, Y.t \approx \tau \in \Gamma}{\Gamma \overset{\mathsf{can}}{\vdash} Y.t \hookrightarrow \tau}$$

$$\textsc{C-Cf-Type}\quad \frac{\Gamma : \mathsf{wf} \qquad \Gamma \vdash P \triangleright \mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} \qquad (\mathsf{type}\, t \approx \tau) \in \overline{\mathcal{D}}}{\Gamma \overset{\mathsf{can}}{\vdash} P.t \hookrightarrow \tau}$$

$$\textsc{C-Cf-Cgr}\quad \frac{\Gamma : \mathsf{wf} \qquad \forall i \in [\![0, n]\!],\ \Gamma \overset{\mathsf{can}}{\vdash} T_i \hookrightarrow \tau_i}{\Gamma \overset{\mathsf{can}}{\vdash} Op(T_0, \ldots, T_n) \hookrightarrow Op(\tau_0, \ldots, \tau_n)}$$

We can define the canonification of declarations and signatures from the canonification of types. An abstract type component effectively introduces a new type α, as we can see in the rule C-Cf-TypeAbs. As types created by a submodule are accessible inside the surrounding module, the existential quantification is lifted up from the declaration level to the signature level, as we can se in the rule C-Cf-Mod and C-Cf-Sig. This is crucial to get wellformed signatures, where all the existentials are bound at the top level (and all submodules have *manifest* signatures). Module type variables are inlined by the rule C-Cf-ModeTypeVar and can have an abstract signature. For simplicity, a subset of the rules are presented here, the full set is given in Appendix B.3.

$$\textsc{C-Cf-Val}\quad \frac{\Gamma \overset{\mathsf{can}}{\vdash} T \hookrightarrow \tau \qquad Y.x \notin \Gamma}{\Gamma \overset{\mathsf{can}}{\vdash}_Y \mathsf{val}\, x : T \hookrightarrow \mathsf{val}\, x : (\approx \tau)}$$

$$\textsc{C-Cf-Type}\quad \frac{\Gamma \overset{\mathsf{can}}{\vdash} T \hookrightarrow \tau \qquad Y.t \notin \Gamma}{\Gamma \overset{\mathsf{can}}{\vdash}_Y \mathsf{type}\, t = T \hookrightarrow \mathsf{type}\, t \approx \tau}$$

$$\textsc{C-Cf-TypeAbs}\quad \frac{\Gamma : \mathsf{wf} \qquad Y.t \notin \Gamma}{\Gamma \overset{\mathsf{can}}{\vdash}_Y \mathsf{type}\, t \hookrightarrow \exists \alpha.\, \mathsf{type}\, t \approx \alpha}$$

$$\textsc{C-Cf-Mod}\quad \frac{\Gamma \overset{\mathsf{can}}{\vdash} S \hookrightarrow \exists \overline{\alpha}.\mathcal{R} \qquad Y.X \notin \Gamma}{\Gamma \overset{\mathsf{can}}{\vdash}_Y (\mathsf{module}\, X : S) \hookrightarrow \exists \overline{\alpha}.\, (\mathsf{module}\, X : \mathcal{R})}$$

$$\textsc{C-Cf-Sig}\quad \frac{\Gamma \overset{\mathsf{can}}{\vdash}_Y \overline{D} \hookrightarrow \exists \overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \overset{\mathsf{can}}{\vdash} \mathtt{sig}_Y\ \overline{D}\ \mathtt{end} \hookrightarrow \exists \overline{\alpha}.\mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end}}$$

$$\textsc{C-Cf-ModTypeVar}\quad \frac{\Gamma : \mathsf{wf} \qquad \Gamma \vdash P.A \triangleright \mathcal{S}}{\Gamma \overset{\mathsf{can}}{\vdash} P.A \hookrightarrow \mathcal{S}}$$

3.3 Typing and subtyping

The canonical signatures and existential types make the typing and subtyping set of rules both smaller and simpler than in the source presentation. In this subsection we give an overview of those judgments.

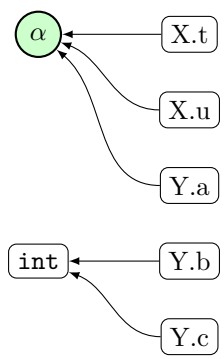
3.3.1 Subtyping

The key mechanism for subtyping canonical signatures is the substitution of existential types. When checking that a given signature $\exists \overline{\alpha}.\mathcal{R}$ is a subset of another one $\exists \overline{\alpha}'.\mathcal{R}'$, we should consider two cases before being able to compare them *declaration by declaration*. First, they can define the same abstract types (up to alpha-conversion), in which case a simple (renaming) substitution $\overline{\alpha \mapsto \alpha'}$ is sufficient before comparing the fields. Second, as we had in the source presentation, the left-hand side signature $\mathcal{R}$ can be *less abstract* (binding fewer abstract types) than the right-hand side one $\mathcal{R}'$, as we can see in the example of Figure 12.

In the source presentation, we dealt with the difference of abstraction by making type declarations subtypes of *abstract* type declarations ($\mathsf{type}\, t = T <: \mathsf{type}\, t$). Here, it is actually easier, as we only need to substitue the extra abstract types of $\mathcal{R}'$ by the concrete types used in $\mathcal{R}$. Both cases are handled by the rule C-Sub-Abs, where the substitution should be understood as beeing able to replace a right-hand-side abstract

```
module type S1 =
∃α, sig
  module X : sig
    type t = α
    type u = α
  end

  module Y : struct
    type a = α
    type b = int
    type c = int
  end
end
```




```
module type S2 =
∃β, γ, sig
  module X : sig
    type t = β
    type u = β
  end

  module Y : struct
    type a = β
    type b = γ
    type c = γ
  end
end
```

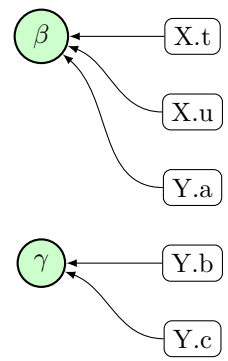


Figure 12: The canonical signature S_1 is a subtype of S_2 using the substitution $[\beta \to \alpha, \gamma \to \texttt{int}]$ that is both *from abstract type to abstract type* and *from abstract type to concrete type*. Before comparing them field by field, the left-hand side existential α is pushed in the context, the substitution is applied to S2. Then both signatures bodies, made *concrete*, can be examined.

type α' by either some left-hand-side abstract type α or by some concrete type τ[5].

$$\textsc{C-Sub-Abs}\quad \frac{\Gamma, \overline{\alpha} \overset{\text{can}}{\vdash} \mathcal{R} <: \mathcal{R}'[\overline{\alpha}' \mapsto \overline{\tau}]}{\Gamma \overset{\text{can}}{\vdash} \exists\overline{\alpha}.\mathcal{R} <: \exists\overline{\alpha}'.\mathcal{R}'}$$

The same idea is present for the subtyping of functor signatures (rule C-Sub-Funct), with a little more complexity coming from the contravariant position of the argument signature. The substitution used to show that $\mathcal{R}'$ is a subtype of $\mathcal{R}$ is reused for the subtyping between the two bodies $\mathcal{S}$ and $\mathcal{S}'$ (with inverted direction):

$$\textsc{C-Sub-Funct}\quad \frac{\Gamma, \overline{\alpha}' \overset{\text{can}}{\vdash} \mathcal{R}' <: \mathcal{R}[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma, \overline{\alpha}', \mathsf{module}\, X : \mathcal{R}' \overset{\text{can}}{\vdash} \mathcal{S}[\overline{\alpha} \mapsto \overline{\tau}] <: \mathcal{S}' \qquad X \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash} \forall\overline{\alpha}.\,(X : \mathcal{R}) \to \mathcal{S} <: \forall\overline{\alpha}'.\,(X : \mathcal{R}') \to \mathcal{S}'}$$

As with the source presentation, we distinguish between two kinds of subtyping: abstraction-only subtyping $\Gamma \overset{\text{can}}{\vdash} \mathcal{S}_1 \prec: \mathcal{S}_2$ and normal subtyping $\Gamma \overset{\text{can}}{\vdash} \mathcal{S}_1 <: \mathcal{S}_2$, which allows both abstraction and deletion of fields. They differ only on one rule C-Sub-Sig vs C-Sub-Sig-Eq: the abstraction-only requires the two lists of declarations $\overline{\mathcal{D}}$ and $\overline{\mathcal{D}}'$ to be compared directly, whereas the normal subtyping compares $\overline{\mathcal{D}}'$ with a subset $\overline{\mathcal{D}}_0$.

$$\textsc{C-Sub-Sig}\quad \frac{\Gamma \overset{\text{can}}{\vdash}_Y \overline{\mathcal{D}} : \mathsf{wf} \qquad \overline{\mathcal{D}}_0 \subseteq \overline{\mathcal{D}} \qquad \overline{\Gamma, \overline{\mathcal{D}} \overset{\text{can}}{\vdash} \mathcal{D}_0 <: \mathcal{D}'} \qquad \Gamma \overset{\text{can}}{\vdash}_Y \overline{\mathcal{D}'} : \mathsf{wf}}{\Gamma \overset{\text{can}}{\vdash} \texttt{sig}_Y\ \overline{\mathcal{D}}\ \texttt{end} <: \texttt{sig}_Y\ \overline{\mathcal{D}'}\ \texttt{end}}$$

$$\textsc{C-Sub-Sig-Eq}\quad \frac{\Gamma \overset{\text{can}}{\vdash}_Y \overline{\mathcal{D}} : \mathsf{wf} \qquad \overline{\Gamma, \overline{\mathcal{D}} \overset{\text{can}}{\vdash} \mathcal{D} \prec: \mathcal{D}'} \qquad \Gamma \overset{\text{can}}{\vdash}_Y \overline{\mathcal{D}'} : \mathsf{wf}}{\Gamma \overset{\text{can}}{\vdash} \texttt{sig}_Y\ \overline{\mathcal{D}}\ \texttt{end} \prec: \texttt{sig}_Y\ \overline{\mathcal{D}'}\ \texttt{end}}$$

The subtyping judgement for declaration is also simpler than in the source presentation, as no conversion from manifest to abstract type is required. The full set of subtyping rules is given in Appendix B.6.

[5]The rule is made simple by the fact that both fall under the same grammatical category

3.3.2 Typing

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

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

C-Typ-Let

$$\frac{\Gamma \overset{\text{can}}{\vdash} E : T \qquad Y.x \notin \Gamma \qquad \Gamma \overset{\text{can}}{\vdash} T \hookrightarrow \tau}{\Gamma \overset{\text{can}}{\vdash}_Y (\mathsf{let}\, x = E) : (\mathsf{val}\, x : (\approx \tau))}$$

C-Typ-Mod

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

C-Typ-Type

$$\frac{\Gamma \overset{\text{can}}{\vdash} T \hookrightarrow \tau \qquad Y.t \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y (\mathsf{type}\, t = T) : (\mathsf{type}\, t \approx \tau)}$$

C-Typ-ModType

$$\frac{\Gamma \overset{\text{can}}{\vdash} S \hookrightarrow \mathcal{S} \qquad Y.A \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y (\mathsf{module\ type}\, A = S) : (\mathsf{module\ type}\, A = \mathcal{S})}$$

C-Typ-Empty

$$\frac{\Gamma : \mathsf{wf}}{\Gamma \overset{\text{can}}{\vdash}_Y \varepsilon : \varepsilon}$$

C-Typ-Seq

$$\frac{\Gamma \overset{\text{can}}{\vdash}_Y B_1 : \exists \overline{\alpha}_1.\overline{\mathcal{D}_1} \qquad \Gamma, \overline{\alpha}_1, \overline{Y.\mathcal{D}_1} \overset{\text{can}}{\vdash}_Y B_2 : \exists \overline{\alpha}_2.\overline{\mathcal{D}_2}}{\Gamma \overset{\text{can}}{\vdash}_Y B_1; B_2 : \exists \overline{\alpha}_1 \overline{\alpha}_2.\overline{\mathcal{D}_1} +\!\!+ \overline{\mathcal{D}_2}}$$

The existential types—which are lifted from submodules with the rule C-Typ-Mod—are merged by C-Typ-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 As mentionned above, the set of typing rules is syntax-directed. Existential types obtained from typing binders are lifted by the rule C-Typ-Struct. Applying a functor to its argument (rule C-Typ-App) reuses the same substitution of abstract types as detailed for subtyping.

C-Typ-Struct

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

C-Typ-Var

$$\frac{\Gamma : \mathsf{wf} \qquad \Gamma \vdash P \triangleright \mathcal{R}}{\Gamma \overset{\text{can}}{\vdash} P : \mathcal{R}}$$

C-Typ-Sig

$$\frac{\Gamma \overset{\text{can}}{\vdash} S \hookrightarrow \mathcal{S} \qquad \Gamma \overset{\text{can}}{\vdash} P : \mathcal{R} \qquad \Gamma \overset{\text{can}}{\vdash} \mathcal{R} <: \mathcal{S}}{\Gamma \overset{\text{can}}{\vdash} (P : S) : \mathcal{S}}$$

C-Typ-Fct

$$\frac{X \notin \Gamma \qquad \Gamma \overset{\text{can}}{\vdash} S_a \hookrightarrow \exists \overline{\alpha}.\mathcal{R} \qquad \Gamma, \overline{\alpha}, \mathsf{module}\, X : \mathcal{R} \overset{\text{can}}{\vdash} M : \mathcal{S}}{\Gamma \overset{\text{can}}{\vdash} ((X : S_a) \to M) : \forall \overline{\alpha}.\, (X : \mathcal{R}) \to \mathcal{S}}$$

C-Typ-App

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

In the source presentation, the projection rule has a special status, because signature avoidance can arise. In the canonical presentation, all the abstract types are specified using the existential binder in front of the signature. Doing so, projecting a component (rule C-Typ-Proj) 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 $\overline{\alpha}$, that remained binded in the signature of $M.X$.

C-Typ-Proj

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

The full set of typing rules is given in Appendix B.7.

3.4 From the source to the canonical system

In this subsection, we explore how typing derivations can be translated from the source system into the canonical system. Both presentations share the same grammar for module expressions and signatures, and thus have the same input for typing. As we will see, using an extension of the canonification predicate to environments, we can translate judgments (lookup, equivalence, strengthening) of the source presentation into the canonical one, giving interesting insight on how the source mechanisms could be understood in the canonical framework. As the source subtyping can lead to loss of type-sharing—which is not the case into the canonical system —, the signature obtained from the source will only be a *subtype* of the corresponding canonical one. We end this subsection with a theorem stating that source typing derivations can be translated into canonical ones. In this whole subsection, proofs are omitted but sketches can be found in appendices.

3.4.1 Canonification revisited

We defined a canonification predicate $\Gamma \overset{\text{can}}{\vdash} S \hookrightarrow \mathcal{S}$ in Section 3.2, to translate source signatures into canonical ones. We extend this predicate to also translate *environments* with the following rules :

C-Cf-Empty
$$\varepsilon \hookrightarrow \varepsilon$$

C-Cf-FctArg
$$\frac{\Gamma_s \hookrightarrow \Gamma_c \qquad \Gamma_c \overset{\text{can}}{\vdash} S \hookrightarrow \exists\overline{\alpha}.\mathcal{R}}{(\Gamma_s\,, \mathsf{module}\, X : S) \hookrightarrow (\Gamma_c\,, \overline{\alpha}\,, \mathsf{module}\, X : \mathcal{R})}$$

C-Cf-Decl
$$\frac{\Gamma_s \hookrightarrow \Gamma_c \qquad \Gamma_c \overset{\text{can}}{\vdash}_Y D \hookrightarrow \exists\overline{\alpha}.\mathcal{D}}{(\Gamma_s\,, Y.D) \hookrightarrow (\Gamma_c\,, \overline{\alpha}\,, Y.\mathcal{D})}$$

As both predicates are syntax-directed, we can define three (mutually recursive) canonification *operations*. One for environments, written $\mathtt{c}_{\mathtt{env}}(\cdot)$, one for signatures (with an environment as parameter), written $\mathtt{c}_{\mathtt{sig}}(\cdot,\cdot)$ and one for declarations (with an environment and a self-reference as parameters), written $\mathtt{c}_{\mathtt{dec}}(\cdot,\cdot,\cdot)$. They satisfy:

$$\Gamma : \mathsf{wf} \implies \Gamma \hookrightarrow \mathtt{c}_{\mathtt{env}}(\Gamma)$$
$$\Gamma \overset{\text{src}}{\vdash} S : \mathsf{wf} \implies \mathtt{c}_{\mathtt{env}}(\Gamma) \overset{\text{can}}{\vdash} S \hookrightarrow \mathtt{c}_{\mathtt{sig}}(\Gamma, S)$$
$$\Gamma \overset{\text{src}}{\vdash}_Y D : \mathsf{wf} \implies \mathtt{c}_{\mathtt{env}}(\Gamma) \overset{\text{can}}{\vdash}_Y D \hookrightarrow \mathtt{c}_{\mathtt{dec}}(\Gamma, Y, D)$$

For conciseness, we write $\mathtt{c}_{\mathtt{env}}(\Gamma)$ as Γ^c, $\mathtt{c}_{\mathtt{sig}}(\Gamma, S)$ as $\mathcal{S}^c$, and $\mathtt{c}_{\mathtt{dec}}(\Gamma, Y, D)$ as $\mathcal{D}^c$ when it is clear from context. Using an operation instead of a relation makes the following results easier to write.

3.4.2 Canonification of lookup

Now that we can convert environments from source to canonical, we can state how the lookup behaves regarding this transformation. The main difference between the two systems comes from the fact that a module that defines an abstract type $\mathtt{t}$, stored at a location P, has (without strenghtening) a signature containing an abstract type field: $\mathsf{type}\, t$. The canonification of this field gives $\exists\alpha.\,\mathsf{type}\, t(\approx \alpha)$: overall, the canonification exposes the existential types. But in the canonical model, the stored signature is *manifest*: first all existential types are stored in the environment, then the signature (referring to them) is stored. Thus the signature returned by the canonical lookup will be stripped from the existential obtained by canonification. On the other hand, module type variables are not made *manifest* in the canonical system, which makes for a simpler link. This gives us the following result:

Theorem 1 (Canonification of lookup)**.** Given any wellformed environment $\Gamma : \mathsf{wf}$, we have:

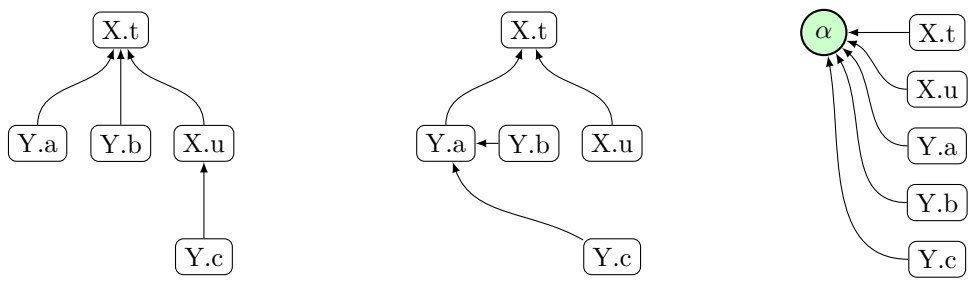
$$\Gamma \vdash P \triangleright S \implies \exists\overline{\alpha}, \mathcal{R}, \begin{cases} \Gamma^c \vdash P \triangleright \mathcal{R} \\ \mathcal{S}^c = \exists\overline{\alpha}.\mathcal{R} \end{cases} \tag{1}$$

$$\Gamma \vdash P.A \triangleright S \implies \Gamma^c \vdash P.A \triangleright \mathcal{S}^c \tag{2}$$

As expected, the signature obtained in the canonical lookup is the same as the canonification of the source one, stripped of its existentials. The module type variables are however not stripped of their existentials.

3.4.3 Canonification of equivalence

The canonification of the equivalence judgement is both simple and crucial. As we described in the previous section, two equivalent signatures have the same connected components of their type sharing dag. In the canonical model, all connected components are flattened into a star trees (tree of depth 1), with an existentially quantified variable at the root. This means that two equivalent signatures actually have the same canonification, as shown in Figure 13. It matches the intuition (and naming) that signatures are indeed *canonical*.



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

Figure 13: Two equivalent signatures S_1 and S_2 (from Figure 6) have the same canonification $\mathcal{S}$. Formally, we have : $\mathtt{c_{sig}}\,(\varepsilon, S_1) = \mathtt{c_{sig}}\,(\varepsilon, S_2) = \mathcal{S}$

This is captured by the following result:

Theorem 2. Given a wellformed signature $\Gamma \overset{\text{src}}{\vdash} S_1 : \mathsf{wf}$, for any signature S_2:

$$\Gamma \vdash S_1 \approx S_2 \implies \mathcal{S}_{1_c} = \mathcal{S}_{2_c} \tag{3}$$

3.4.4 Canonification of strengthening

The case of canonification of strengthening also gives an interesting insight on how this mechanism, necessary for the source presentation, becomes completely transparent with the existential types. Strengthening the signature of P, by turning abstract types into concrete ones ($\mathsf{type}\, t \twoheadrightarrow \mathsf{type}\, t = P.t$), makes the signature *manifest*, while keeping the same fields. In the canonical model, as we store *manifest* signatures in the context (storing the existential types before the signature), all copies of a signature directly refer to the existentials in the context. The link between module aliases is therefore obtained directly, through the saved existentials. As with equivalence, having the existentials as accessible roots for the connected components of the type sharing dag makes the presentation simple.

Theorem 3. Given an environment Γ and a path P of signature S, that is $\Gamma \overset{\text{src}}{\vdash} P : S$, we have:

$$\Gamma \vdash P/S \twoheadrightarrow S' \implies \begin{cases} \exists \overline{\alpha}.\mathcal{R} = \mathtt{c_{sig}}\,(\Gamma, S) \\ \mathcal{R} = \mathtt{c_{sig}}\,(\Gamma, S') \\ \Gamma^c \overset{\text{can}}{\vdash} P : \mathcal{R} \end{cases} \tag{4}$$

3.4.5 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, through over-abstraction, source subtyping can lose type equalities. However, in the canonical system, with the rule C-Sub-Abs, the correspondence between the abstract types is dealt with a the signature level, no abstraction is done at the declaration level. We get the following result:

Theorem 4. Given an environment Γ and two wellformed signatures, we have:

$$\Gamma \overset{\text{src}}{\vdash} S_1 <: S_2 \implies \Gamma^c \overset{\text{can}}{\vdash} \mathcal{S}_{1_c} <: \mathcal{S}_{2_c} \tag{5}$$

3.4.6 Canonification of typing

Once all the 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 subtyping can lead to a source signature having lost some type sharing. This type-sharing loss however, can be delayed to the last step of the derivation (combining the losses of all the stored signatures), which gives the following result:

Theorem 5 (Canonification of typing)**.** Given any environment Γ, module M, and signature S, we have:

$$\Gamma \overset{\text{src}}{\vdash} M : S \implies \exists S', \begin{cases} \Gamma^c \overset{\text{can}}{\vdash} M : \mathcal{S}' \\ \Gamma^c \overset{\text{can}}{\vdash} \mathcal{S}' \prec: \mathcal{S}^c \end{cases} \tag{6}$$

This main result matches the intuition that the canonical system is *more powerful* than the source system, and that the existentials are an efficient and simple way of dealing with type-sharing. As we mentioned in the description of the projection typing-rule, we could add some (cumbersome) conditions to the source system to force the preservation of type equalities.

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 Anchorability conditions: the power of existentials into a path-based approach

In this section we are going to explore at which conditions canonical typing derivations can be translated back into the source typing. This would allow us to use the canonical system as a guide to implement mechanisms to improve the current OCaml type-checker, especially regarding the type-sharing preservation. The canonical system thus provides both the intuition, formal guaranties, and algorithmic insights for the design of OCaml modules mechanisms. In the first subsection, we define a notion of anchorability that captures the cases where canonical signatures could be expressed in the source (*anchorable wellformedness*). Using this notion as a new wellformedness predicate, we can easily change the typing rules so that all intermediary signatures of a typing derivation could be translated back into the source (*anchorable typing*). In the last subsection, we give the formal results showing the correspondence between this restricted typing and the source typing, by extending the notion of *anchorable* signatures into *anchored* signatures.

4.1 Anchorable wellformedness and typing

4.1.1 Signature avoidance revisited

Existence vs Identity (of types) As we mentioned in Section 1.2, some cases of signature avoidance are *solvable*, whereas some other have no solution in the source syntax. The key distinction between the two cases comes from the fact that the source definition for an abstract type (`type` t) captures both the *existence* and *identity* of the type. There is no way to declare the existence of an abstract type without an alias (an identity) for this type. However, sill in the source syntax, the user can hide the definition (hide the identity) of an abstract type while keeping fields that refer to it (keep the existence) using ascription, projection, or functor application: the resulting signature cannot be expressed. An example is given in Figure 14. Canonical signatures do not suffer from the signature avoidance problem mainly because they can express the existence of types by using existentials, even when they have no identity (no aliases).

```
(* Argument signature :
   type u and v share the
   same first component : t *)
module type S = sig
  type t
  type u = t * int
  type v = t * string
end

(* Functor :
   copies u and v (not t) *)
module F = functor (A: S) ->
  type u = A.u
  type v = A.v
end
```

```
(* Unnamed argument in functor call :
  the signature of module X cannot
  refer to the type t, first component
  of u and v *)
module X = F(struct
  type t = bool
  type u = t * int
  type v = t * string
end : S) (* : sig ? end *)

(* The canonical signature of X *)
module type XSig = ∃α. sig
  type u = α * int
  type v = α * string
end
```

Figure 14: A typical example of an *unsolvable* case of signature avoidance. The source syntax cannot express the existence of a type that is the first component of both `u` and `v` without giving it a name. Fixing the signature by adding a type field for `t` would not work, as it would not be alpha-convertible. However, the module `X` can be given a canonical signature `XSig`.

The *solvable* cases of signature avoidance are the ones where each use of an abstract type have an *in-scope* alias defined (*identified*) before the type is used. Typically, a definition `type` $t \approx \alpha$ should precedes any use of α (such as `val` t : ($\approx \alpha$), `type` $u \approx \alpha \times$ `int`, etc.). In such a case, we say that the existential type is

anchorable: there exists an *in-scope* path we can use to refer to it (to *anchor* it). As we will see in the following subsections, substituting the existential for the corresponding *anchoring instance* allows one to rebuild a source signature. The two cases explored in the figures of Section 1.2 are illustrated in Figure 15.

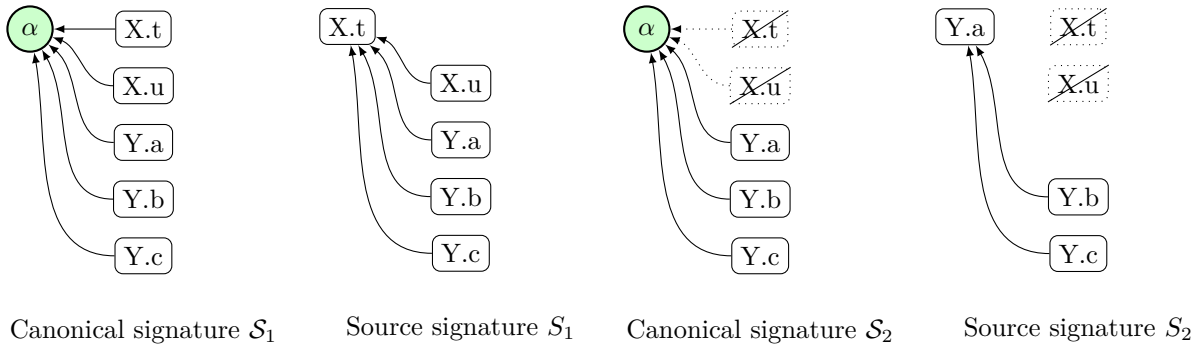


Figure 15: In the two canonical signatures $\mathcal{S}_1$ and $\mathcal{S}_2$, the existential type α is anchorable. For $\mathcal{S}_1$, we can use $X.t$ as the anchoring path, as done in S_1. For $\mathcal{S}_2$, a projection removed $X.t$ and $X.u$. The next available alias is $Y.a$: it is done in S_2.

4.1.2 Anchorable wellformedness

In this subsection we translate the idea of *anchorable* signatures (where existential types can be anchored) into a judgment $\Gamma \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a$. This judgment is presented as a special kind of wellformedness judgment, so that changing from typing derivation to anchorable typing derivation could be done by only changing the underlying notion of wellformedness. In addition to checking the same properties as normal wellformedness (defined in Section 3.2), we enforce two other key properties that should be understood as consequences of forcing the overlapping of *identity* and *existence*. First, a notion of *restricting the meaning of existentials* that should be thought of as the *local* consequence, and, second, a notion of *storing only existentials with a path*, that should be understood as the *global* consequence.

Restricting the meaning of existentials The normal version of the wellformedness predicate use a *top-down* approach: the set of existentials is specified in front of the signature ($\exists\overline{\alpha}.\texttt{sig}\ \ldots\ \texttt{end}$) and can be used freely in the declarations. Here, we use a *bottom-up* approach: existentials are added in front of a signature only when we encounter defining instances of the form $\mathsf{type}\, t \approx \alpha$. Their meaning is thus restricted from expressing existence to expressing existence *and* identity. To do so, we extend the predicate for declaration to indicate the set of existentials : $\Gamma \overset{\text{can}}{\vdash}_Y \exists\alpha.\overline{D} : \mathsf{wf}_a$. The rules for individual declarations are given bellow. The only one where an existential can be introduced is C-AWf-TypeAbs (and, recursively for sub-modules, C-AWf-Mod).

C-AWf-Val
$$\frac{\Gamma \overset{\text{can}}{\vdash} \tau : \mathsf{wf}_a \qquad Y.x \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \mathsf{val}\, x : (\approx \tau) : \mathsf{wf}_a}$$

C-AWf-Type
$$\frac{\Gamma \overset{\text{can}}{\vdash} \tau : \mathsf{wf}_a \qquad Y.t \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \mathsf{type}\, t \approx \tau : \mathsf{wf}_a}$$

C-AWf-TypeAbs
$$\frac{\Gamma : \mathsf{wf}_a \qquad Y.t \notin \Gamma \qquad \alpha \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \exists\alpha.\, (\mathsf{type}\, t \approx \alpha) : \mathsf{wf}_a}$$

C-AWf-Mod
$$\frac{\Gamma \overset{\text{can}}{\vdash} \exists\overline{\alpha}.\mathcal{R} : \mathsf{wf}_a \qquad Y.X \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}.\, (\mathsf{module}\, X : \mathcal{R}) : \mathsf{wf}_a}$$

C-AWf-ModType
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a \qquad Y.A \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y (\mathsf{module\ type}\, A = \mathcal{S}) : \mathsf{wf}_a}$$

In addition, there are two sequence rules, given bellow. The rule C-AWf-Seq is used to merge the existentials: the wellformedness of $\mathcal{D}_1$ is not checked with the rest of the $\overline{\alpha}$ in the context (forcing them to

be defined before being used). As one could expect, this has similarities with the canonification judgment (as both manipulate signatures where the notions of identity and existence of types are overlapping).

$$\text{C-AWf-Empty}\quad \frac{\Gamma : \mathsf{wf}_a}{\Gamma \overset{\text{can}}{\vdash}_Y \varepsilon : \mathsf{wf}_a} \qquad \text{C-AWf-Seq}\quad \frac{\Gamma \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}_1.\mathcal{D}_1 : \mathsf{wf}_a \qquad \Gamma\,,\overline{\alpha}_1\,,Y.\mathcal{D}_1 \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}.\overline{\mathcal{D}} : \mathsf{wf}_a \qquad \overline{\alpha}\cap\overline{\alpha}_1 = \emptyset}{\Gamma \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}_1\overline{\alpha}.\left(\mathcal{D}_1\,,\overline{\mathcal{D}}\right) : \mathsf{wf}_a}$$

Finally, for the source judgment ($\Gamma \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a$), we remove the rule C-WF-Abs that allowed for introducing any number of existentials, regardless of whether or not they were anchored. Instead, we only have the two rules given bellow. C-AWf-Sig is used for signatures with the existential types being obtained from the declaration judgment.

$$\text{C-AWf-Sig}\quad \frac{\Gamma \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}.\overline{\mathcal{D}} : \mathsf{wf}_a}{\Gamma \overset{\text{can}}{\vdash} \exists\overline{\alpha}.\mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} : \mathsf{wf}_a} \qquad \text{C-AWf-Funct}\quad \frac{\Gamma \overset{\text{can}}{\vdash} \exists\overline{\alpha}.\mathcal{R} : \mathsf{wf}_a \qquad \Gamma\,,\overline{\alpha}\,,\mathsf{module}\,X : \mathcal{R} \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a \qquad X \notin \Gamma \qquad \overline{\alpha} \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash} \forall\overline{\alpha}.\,(X : \mathcal{R}) \to \mathcal{S} : \mathsf{wf}_a}$$

Storing only existentials with a path Using the restricted meaning of existentials, we know that there is a *local* alias for all the introduced existentials. Before pushing any existential to the context Γ, we also require that these local aliases are actually reachable with a path. The rules for the environment judgment are given bellow. The only two rules where existential types can be stored in the environment are C-AWf-FctArg and C-AWf-Dec, where a path is available to refer to the local alias they correspond to (either the functor name for C-Awf-FctArg or the self reference for C-Awf-Dec).

$$\text{C-AWf-Empty}\quad \varepsilon : \mathsf{wf}_a \qquad \text{C-AWf-FctArg}\quad \frac{\Gamma \overset{\text{can}}{\vdash} \exists\overline{\alpha}.\mathcal{R} : \mathsf{wf}_a \qquad X \notin \Gamma \qquad \overline{\alpha} \notin \Gamma}{\Gamma\,,\overline{\alpha}\,,(\mathsf{module}\,X : \mathcal{R}) : \mathsf{wf}_a} \qquad \text{C-AWf-Dec}\quad \frac{\Gamma \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}.\mathcal{D} : \mathsf{wf}_a \qquad \overline{\alpha} \notin \Gamma}{\Gamma\,,\overline{\alpha}\,,Y.\mathcal{D} : \mathsf{wf}_a}$$

The full set of rules is given in B.4. As the *anchorability* only adds new premises to the rules, we show by an easy induction that:

$$\forall \Gamma, \mathcal{S},\ \Gamma \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a \implies \Gamma \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf} \tag{7}$$

This matches the intuition that anchorable wellformedness is more restrictive—which was expected of a statement matching the source presentation restrictions.

4.1.3 Anchorable typing

We can easily modify the typing judgment, by replacing the notion of wellformedness wf by wf_a. Otherwise, the new judgment $\Gamma \overset{\text{can}}{\vdash}_a M : \mathcal{S}$ reuses exactly the same rules as the ones defined in Section 3.3.2. For example, the projection rule becomes:

$$\text{C-ATyp-Proj}\quad \frac{\Gamma \overset{\text{can}}{\vdash}_a M : \exists\overline{\alpha}.\mathtt{sig}_Y\ \overline{\mathcal{D}}_1\,,\mathsf{module}\,X : \mathcal{R}\,,\overline{\mathcal{D}}_2\ \mathtt{end} \qquad \Gamma \overset{\text{can}}{\vdash} \exists\overline{\alpha}.\mathcal{R} : \mathsf{wf}_a}{\Gamma \overset{\text{can}}{\vdash}_a M.X : \exists\overline{\alpha}.\mathcal{R}}$$

Doing so, all the signatures used in intermediary steps are *anchorable*, translatable into source ones. In the next section we give the formal translation theorems, which requires extending the judgments for the proofs.

4.2 Anchorability result

In this section we present the formal result of anchorability. To prove it, we extend the *anchorable* wellformedness to an *anchored* wellformedness: a judgment that also returns a substitution and a source signature to exhibit a solution. First, we state the result.

Theorem 6 (Anchorable typing)**.** Given any environment Γ, module expression M and (canonical) signature $\mathcal{S}$, we have:

$$\Gamma \overset{\text{can}}{\vdash}_a M : \mathcal{S} \implies \exists \Gamma_s, S, \begin{cases} \Gamma_s \hookrightarrow \Gamma \\ \Gamma \overset{\text{can}}{\vdash} S \hookrightarrow \mathcal{S} \\ \Gamma_s \overset{\text{src}}{\vdash} M : S \end{cases} \tag{8}$$

This completes the link between the source and canonical systems: canonical derivations can be translated back into the source when we restrict their expressiveness. In the rest of this section, we sketch the tools and strategy we use for the proof. As we mentioned, in *anchorable* signatures, all existentials can be associated with a path, so there exists a substitution σ from existential identifiers to paths. Substituting the existential for their associated paths allows us to build a source signature. We will first extend the anchorable wellformedness into an *anchored* wellformedness judgment that also binds the substitution and associated source signature.

4.2.1 Anchored wellformedness

Anchorable wellformedness was presented in the last section as a (simple) set of necessary conditions for source translation. We extend it to return both a source evidence term and a substitution for the (canonical) existentials to the (source) paths $\sigma : \alpha \mapsto T$. This gives the following mutually recursive judgments:

- Environments: $\Gamma : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma)$
- Signatures: $\Gamma \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma), (S, \sigma_S)$
- Declarations: $\Gamma \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}.\overline{\mathcal{D}} : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma), (\overline{D}, \sigma)$
- Types: $\Gamma \overset{\text{can}}{\vdash} \tau : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma), T$

Anchored types The simplest judgment is the anchored wellformedness of types. Based on the substitution of existentials for paths of the current environment, we can easily get the source evidence term from the canonical one, with the two following rules:

C-AWEf-Existential
$$\frac{\Gamma : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma)}{\Gamma \overset{\text{can}}{\vdash} \alpha : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), \sigma_\Gamma(\alpha)}$$

C-AWEf-Congruence
$$\frac{\forall i \in [\![1, n]\!], \Gamma \overset{\text{can}}{\vdash} \tau_i : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), T_i}{\Gamma \overset{\text{can}}{\vdash} Op(\tau_1, \ldots, \tau_n) : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), Op(T_1, \ldots, T_n)}$$

Anchored declarations We can see the two principles of anchorable wellformedness (*existential restriction* and *storing only with a path*) at play in the rules. The declaration version of the judgment is where the substitutions σ_Γ and σ are built, field by field. The only place where an explicit substitution is added is (unsurprisingly) for the abstract type declaration (rule C-AWEf-TypeAbs[6]), whereas other rules (like

[6]There is a distinction between type declarations with an existential $\exists\alpha.\,\mathsf{type}\, t \approx \alpha$ that correspond to the defining instance, and type declaration without an existential $\mathsf{type}\, t \approx \beta$, that correspond only to an alias

C-AWEF-VAL, C-AWEF-TYPE) only return empty substitutions.

C-AWEF-VAL

$$\frac{\Gamma \overset{\text{can}}{\vdash} \tau : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), T \qquad Y.x \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \mathsf{val}\, x : (\approx \tau) : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), (\mathsf{val}\, x : T, \emptyset)}$$

C-AWEF-TYPE

$$\frac{\Gamma \overset{\text{can}}{\vdash} \tau : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), T \qquad Y.t \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \mathsf{type}\, t \approx \tau : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), (\mathsf{type}\, t = T, \emptyset)}$$

C-AWEF-TYPEABS

$$\frac{\Gamma : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma) \qquad Y.t \notin \Gamma \qquad \alpha \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \exists \alpha. (\mathsf{type}\, t \approx \alpha) : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), (\mathsf{type}\, t, \alpha \mapsto Y.t)}$$

For module fields, we need to distinguish between structures and functors. Indeed, as functors are generative, no type access can be made inside functor modules: the substitution is empty (rule C-AWEF-MODFCT). For modules with a signature of a structure ($\exists \overline{\alpha}.\mathtt{sig}\ \overline{\mathcal{D}}\ \mathtt{end}$), we extend the access paths in the substitution (rule C-AWEF-MODABS).

C-AWEF-MODFCT

$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathcal{R} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), (S, \emptyset) \qquad Y.X \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y (\mathsf{module}\, X : \mathcal{R}) : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), ((\mathsf{module}\, X : S), \emptyset)}$$

C-AWEF-MODABS

$$\frac{\Gamma \overset{\text{can}}{\vdash} \exists \overline{\alpha}. \left(\mathtt{sig}_X\ \overline{\mathcal{D}}\ \mathtt{end}\right) : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), \left(\mathtt{sig}_X\ \overline{D}\ \mathtt{end}, \sigma\right) \qquad Y.X \notin \Gamma \qquad \sigma' = \alpha \mapsto Y.\sigma(\alpha)}{\Gamma \overset{\text{can}}{\vdash}_Y \exists \overline{\alpha}. \left(\mathsf{module}\, X : \mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end}\right) : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), \left(\left(\mathsf{module}\, X : \mathtt{sig}_Y\ \overline{D}\ \mathtt{end}\right), \sigma'\right)}$$

Finally, we add rules for sequence that combine the substitutions being built (in C-AWEF-SEQ).

C-AWEF-EMPTY

$$\frac{\Gamma : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma)}{\Gamma \overset{\text{can}}{\vdash}_Y \varepsilon : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), (\varepsilon, \emptyset)}$$

C-AWEF-SEQ

$$\frac{\Gamma \overset{\text{can}}{\vdash}_Y \exists \overline{\alpha}_1.\mathcal{D}_1 : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), (D_1, \sigma_1) \qquad \Gamma, \overline{\alpha}_1, Y.\mathcal{D}_1 \overset{\text{can}}{\vdash}_Y \exists \overline{\alpha}.\overline{\mathcal{D}} : \mathsf{wf}_a \hookleftarrow ((\Gamma_{\mathsf{src}}, Y.D_1), \sigma'), \left(\overline{D}, \sigma_2\right) \qquad \overline{\alpha}_1 \cap \overline{\alpha} = \emptyset}{\Gamma \overset{\text{can}}{\vdash}_Y \exists \overline{\alpha}_1 \overline{\alpha}. \left(\mathcal{D}_1, \overline{\mathcal{D}}\right) : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), \left(\left(D_1, \overline{D}\right), \sigma_1 \uplus \sigma_2\right)}$$

Anchored signatures The signatures rules wrap up the declaration ones. If a signature has existential types ($\exists \overline{\alpha}.\mathcal{R}$), the associated substitution σ translates them into paths inside the signature, as in rule C-AWEF-SIG. Functor signatures have an empty substitution (rule C-AWEF-FUNCT).

C-AWEF-SIG

$$\frac{\Gamma \overset{\text{can}}{\vdash}_Y \exists \overline{\alpha}.\overline{\mathcal{D}} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), \left(\overline{D}, \sigma\right)}{\Gamma \overset{\text{can}}{\vdash} \exists \overline{\alpha}.\mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), \left(\mathtt{sig}_Y\ \overline{D}\ \mathtt{end}, \sigma\right)}$$

C-AWEF-FUNCT

$$\frac{\Gamma \overset{\text{can}}{\vdash} \exists \overline{\alpha}.\mathcal{R} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), (S_1, \sigma_1) \qquad \Gamma, \overline{\alpha}, \mathsf{module}\, X : \mathcal{R} \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a \hookleftarrow ((\Gamma_{\mathsf{src}}, \mathsf{module}\, X : S_1), \sigma'), (S_2, \sigma_2) \qquad X \notin \Gamma \qquad \overline{\alpha} \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash} \forall \overline{\alpha}. (X : \mathcal{R}) \to \mathcal{S} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma_\Gamma), ((X : S_1) \to S_2, \emptyset)}$$

Anchored environments Whereas the key principle of anchored declaration was to *restrict existentials meaning*, the key principle of anchored environments rules is to *store existentials with a path*. When adding a field to the environment, we extend the associated substitution with the self reference, as in the rule C-AWEF-DEC. For functor arguments, we use the same distinction on the signature as in the anchored

declarations: a functor signature will not allow for type access, so the substitution is empty (rule C-AWEf-FctArgFct), whereas a structure signature will be associated with a substitution, that we extend with a prefix and combine with the current substitution (rule C-AWEf-FctArgAbs).

C-AWEf-Dec

$$\frac{\Gamma \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}.\overline{\mathcal{D}} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), (\overline{D}, \sigma') \qquad \overline{\alpha} \notin \Gamma \qquad \sigma'' = \beta \mapsto \begin{cases} Y.\sigma'(\beta) & \text{if } \beta \in \overline{\alpha} \\ \sigma(\beta) & \text{otherwise} \end{cases}}{\Gamma, \overline{\alpha}, \overline{Y.\mathcal{D}} : \mathsf{wf}_a \hookleftarrow ((\Gamma_{\mathsf{src}}, \overline{Y.D}), \sigma'')}$$

C-AWEf-FctArgFct

$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathcal{R} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), (S, \emptyset) \qquad X \notin \Gamma \qquad \overline{\alpha} \notin \Gamma}{\Gamma, \overline{\alpha}, (\mathsf{module}\, X : \mathcal{R}) : \mathsf{wf}_a \hookleftarrow ((\Gamma_{\mathsf{src}}, \mathsf{module}\, X : S), \emptyset)}$$

C-AWEf-FctArgAbs

$$\frac{\begin{array}{c}\Gamma \overset{\text{can}}{\vdash} \exists\overline{\alpha}.\mathtt{sig}_X\ \overline{\mathcal{D}}\ \mathtt{end} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), (\mathtt{sig}_X\ \overline{D}\ \mathtt{end}, \sigma') \\ X \notin \Gamma \qquad \overline{\alpha} \notin \Gamma \qquad \sigma'' = \beta \mapsto \begin{cases} X.\sigma'(\beta) & \text{if } \beta \in \overline{\alpha} \\ \sigma(\beta) & \text{otherwise} \end{cases}\end{array}}{\Gamma, \overline{\alpha}, (\mathsf{module}\, X : \mathtt{sig}_X\ \overline{\mathcal{D}}\ \mathtt{end}) : \mathsf{wf}_a \hookleftarrow ((\Gamma_{\mathsf{src}}, \mathsf{module}\, X : \mathtt{sig}_X\ \overline{D}\ \mathtt{end}), \sigma'')}$$

C-AWEf-Empty

$$\varepsilon : \mathsf{wf}_a \hookleftarrow (\varepsilon, \emptyset)$$

This completes the presentation of the anchored wellformedness. The full set of rules can be found in B.5. This judgment is more technical than the *anchorability*—which provided intuition and necessary conditions—but makes theorem statements and proof simpler, thanks to the invariants that it verifies.

4.2.2 Proof sketch

The proof presented here is not fully completed, and would benefit from being mechanized, which we plan to do in future work. We present some of the key lemma and proof insights. First, we formalize the link between anchorable and anchored wellformedness. Then, we give several auxiliary results for the judgments of the canonical system. We prove the main result by induction on the typing rules.

Lemma 1 (Anchorable and anchored wellformedness equivalences)**.** Given any environment Γ, signature $\mathcal{S}$, declaration $\mathcal{D}$, and type τ we have:

$$\Gamma : \mathsf{wf}_a \iff \exists\, \Gamma_s,\, \sigma_\Gamma,\qquad \Gamma : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma) \tag{9}$$

$$\Gamma \overset{\text{can}}{\vdash} \tau : \mathsf{wf}_a \iff \exists\, \Gamma_s,\, \sigma_\Gamma,\, T,\qquad \Gamma \overset{\text{can}}{\vdash} \tau : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma), T \tag{10}$$

$$\Gamma \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a \iff \exists\, \Gamma_s,\, \sigma_\Gamma,\, S,\, \sigma,\, \Gamma \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma), (S, \sigma) \tag{11}$$

$$\Gamma \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}.\mathcal{D} : \mathsf{wf}_a \iff \exists\, \Gamma_s,\, \sigma_\Gamma,\, D,\, \sigma, \Gamma \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}.\mathcal{D} : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma), (D, \sigma) \tag{12}$$

By induction on the rules, we can check that the source and substitution part of the anchored judgment are not more restrictive than the *anchorability* judgment.

Lemma 2 (Anchorability of lookup)**.** Given Γ, Γ_s, σ_Γ such that $\Gamma : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma)$, given a path P, a module type identifier A, and two signatures $\mathcal{R}$ and $\mathcal{S}$, we have:

$$\Gamma \vdash P \triangleright \mathcal{R} \implies \exists\, S,\, \sigma,\, \begin{cases} \Gamma \overset{\text{can}}{\vdash} \mathcal{R} : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma), (S, \sigma) \\ \Gamma_s \vdash P \triangleright S \end{cases} \tag{13}$$

$$\Gamma \vdash P.A \triangleright \mathcal{S} \implies \exists\, S,\, \sigma,\, \begin{cases} \Gamma \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma), (S, \sigma) \\ \Gamma_s \vdash P.A \triangleright S \end{cases} \tag{14}$$

Lemma 3 (Anchorability of canonification)**.** Given Γ, Γ_s, σ_Γ such that $\Gamma : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma)$, given a canonical signature $\mathcal{S}$ and a source signature S, we have:

$$\Gamma \overset{\text{can}}{\vdash} S \hookrightarrow \mathcal{S} \iff \exists \sigma, \Gamma \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma), (S, \sigma) \tag{15}$$

Lemma 4 (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, two substitutions σ_1 and σ_2, we have:

$$\left.\begin{array}{r} \Gamma \overset{\text{can}}{\vdash} \mathcal{S}_1 : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma), (S_1, \sigma_1) \\ \Gamma \overset{\text{can}}{\vdash} \mathcal{S}_2 : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma), (S_2, \sigma_2) \\ \Gamma \overset{\text{can}}{\vdash} \mathcal{S}_1 <: \mathcal{S}_2 \end{array}\right\} \implies \Gamma_s \overset{\text{src}}{\vdash} S_1 <: S_2 \tag{16}$$

The main theorem (Theorem 6) can now be proved by induction on the typing rules. We give the statement again:

$$\Gamma \overset{\text{can}}{\vdash}_a M : \mathcal{S} \implies \exists \Gamma_s, S, \begin{cases} \Gamma_s \hookrightarrow \Gamma \\ \Gamma \overset{\text{can}}{\vdash} S \hookrightarrow \mathcal{S} \\ \Gamma_s \overset{\text{src}}{\vdash} M : S \end{cases}$$

We reason by induction on the canonical typing rules. We show here two example cases:

- C-Typ-Var $\Gamma \overset{\text{can}}{\vdash}_a P : \mathcal{R}$
  1. We have an anchorable wellformed environment Γ, so, by the equivalence results (Lemma 1), there exists a source environment Γ_s and a substitution σ_Γ such that $\Gamma : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma)$.
  2. By anchorability of lookup, there exists a source signature S and a substitution σ such that $\Gamma_s \overset{\text{src}}{\vdash} P : S$ and $\Gamma \overset{\text{can}}{\vdash} \mathcal{R} : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma), (S, \sigma)$.
  3. The strengthened signature S' (satisfying $\Gamma_s \vdash P/S \twoheadrightarrow S'$) satisfies the theorems conditions (by the source typing rule S-Typ-Var).
- C-Typ-Sig $\Gamma \overset{\text{can}}{\vdash}_a (P : S) : \mathcal{S}$
  1. We have an anchorable wellformed environment Γ, so, there exists a source environment Γ_s and a substitution σ_Γ such that $\Gamma : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma)$.
  2. By anchorability of lookup, there exists a source signature S_0 and a substitution σ_0 such that $\Gamma_s \overset{\text{src}}{\vdash} P : S_0$ and $\Gamma \overset{\text{can}}{\vdash} \mathcal{R} : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma), (S_0, \sigma_0)$.
  3. By the anchorability of canonification, there exists σ_1 such that $\Gamma \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a \hookleftarrow (\Gamma_s, \sigma_\Gamma), (S, \sigma_1)$.
  4. By the anchorability of subtyping, we have $\Gamma_s \overset{\text{src}}{\vdash} S_0 <: S$
  5. We conclude by the source typing rule S-Typ-Sig.

This concludes the technical part of this section. The link between the canonical and source systems is now clarified. The next section will show the missing link between the canonical system and F^ω by a full elaboration of the canonical judgments into F^ω terms. Thus, the canonical system serves indeed as a middle point between the source and F^ω.

5 Formal guaranties through elaboration in F^ω

In this section we give formal foundations and guarantees for the canonical system through *elaboration*. Elaborating (translating) the terms and judgments of the canonical presentation into well-typed terms and judgments of F^ω has two interesting aspects. First, it gives a clear correspondence between some of the mechanisms of the canonical typing (mainly existential types) and standard mechanisms of F^ω, which serves the intuition. Second, it shows that the canonical system can be understood as a *particular mode of use* of F^ω, which means that the elaborated system enjoys the properties of soundness, progress and preservation of F^ω. As the main mechanism of the canonical system (existential types) is directly inspired by F^ω, the elaboration is mostly straightforward. In the first subsection, we give the syntax and rules of the version of F^ω (extended with existential types and records) that we use for the elaboration, and we present the F^ω terms and types that encode signatures, called *semantic* signatures. In the following subsection, we present the elaboration of the canonical judgments that compose the *elaborated system*. Finally, we state the correctness theorem for the elaboration and the link between the canonical and elaborated system.

This section is largely inspired by Rossberg et al. [2014]. The encoding of signatures into F^ω types and terms is very similar. The main difference (beside the grammar of the module expressions) comes from the fact that we elaborate judgments with both a canonical and F^ω result, whereas in Rossberg et al. [2014], typing and subtyping are defined *only* by elaboration (no signatures in the module syntax).

5.1 F-omega and encoded signatures

In this subsection we present the polymorphic lambda calculus that we use for the elaboration. It is a variant of F^ω extended with records and existential types. The grammar is given in Figure 16. The typing rules are standard, with the additional rules for record and existential types. They are given in Figure 17. All identifiers Z are implicitly associated with a record label l_Z.

$$\kappa ::= \Omega \mid \kappa \to \kappa \qquad \text{(kinds)}$$

$$\tau ::= \alpha \mid \tau \to \tau \mid \{\overline{l : \tau}\} \mid \forall\alpha : \kappa.\tau \mid \exists\alpha : \kappa.\tau \mid \lambda\alpha : \kappa.\tau \mid \tau\,\tau \qquad \text{(types)}$$

$$e ::= x \mid \lambda x : \tau.e \mid e\,e \mid \{\overline{l : e}\} \mid e.l \mid \lambda\alpha : \kappa.e \mid e\,\tau \mid \mathsf{pack}\,\langle\tau, e\rangle_\tau \mid \mathsf{unpack}\,\langle\alpha, x\rangle = e\,\mathsf{in}\,e \qquad \text{(terms)}$$

$$v ::= \lambda x : \tau.e \mid \{\overline{l : v}\} \mid \lambda\alpha : \kappa.e \mid \mathsf{pack}\,\langle\tau, v\rangle_\tau \qquad \text{(values)}$$

$$\Theta ::= \cdot \mid \Theta, \alpha : \kappa \mid \Theta, x : \tau \qquad \text{(environments)}$$

Figure 16: Syntax of F_ω

Encoded signatures In the elaboration, signatures are translated into types of F^ω using an encoding (see Figure 18). An implicit pattern-matching on terms and types allows us to distinguish between the constructs in the inference rules.

An example of encoding of a simple module is given in Figure 19, using the syntactic sugar of Figure 18.

5.2 The elaborated system

5.2.1 System structure

The *elaborated* system structure follows the canonical one, as presented in Figure 20. The judgments are extended to present both the canonical and F^ω terms. As we will see in the last subsection, adding the F^ω terms to the judgments is actually not restrictive, and every judgment satisfies invariants linking the canonical and F^ω parts together. We define all judgments over an extended, *hybrid* environment Δ that contains both canonical and F^ω terms. Its definition is given in Figure 21, where the notation $Y.Z : \mathcal{D} \rightsquigarrow \Sigma$

Environments

$$\cdot \overset{F_\omega}{\vdash} \square \qquad \frac{\Theta \overset{F_\omega}{\vdash} \square \qquad \alpha \notin \Theta}{\Theta\,,\alpha : \kappa \overset{F_\omega}{\vdash} \square} \qquad \cdot \overset{F_\omega}{\vdash} \square$$

Types

$$\frac{\Theta \overset{F_\omega}{\vdash} \tau_1 : \Omega \qquad \Theta \overset{F_\omega}{\vdash} \tau_2 : \Omega}{\Theta \overset{F_\omega}{\vdash} \tau_1 \to \tau_2 : \Omega} \qquad \frac{\overline{\Theta \overset{F_\omega}{\vdash} \tau : \Omega} \qquad \Theta \overset{F_\omega}{\vdash} \square}{\Gamma \overset{F_\omega}{\vdash} \{\overline{l : \tau}\} : \Omega} \qquad \frac{\Theta \overset{F_\omega}{\vdash} \square}{\Theta \overset{F_\omega}{\vdash} \alpha : \Theta(\alpha)} \qquad \frac{\Theta\,,\alpha : \kappa \overset{F_\omega}{\vdash} \tau : \Omega}{\Theta \overset{F_\omega}{\vdash} \forall\alpha : \kappa.\tau : \Omega}$$

$$\frac{\Theta\,,\alpha : \kappa \overset{F_\omega}{\vdash} \tau : \Omega}{\Theta \overset{F_\omega}{\vdash} \exists\alpha : \kappa.\tau : \Omega} \qquad \frac{\Theta\,,\alpha : \kappa \overset{F_\omega}{\vdash} \tau : \kappa'}{\Theta \overset{F_\omega}{\vdash} \lambda\alpha : \kappa.\tau : \kappa \to \kappa'} \qquad \frac{\Theta \overset{F_\omega}{\vdash} \tau_1 : \kappa' \to \kappa \qquad \Theta \overset{F_\omega}{\vdash} \tau_2 : \kappa'}{\Theta \overset{F_\omega}{\vdash} \tau_1\ \tau_2 : \kappa}$$

Terms

$$\frac{\Theta \overset{F_\omega}{\vdash} \square}{\Theta \overset{F_\omega}{\vdash} x : \Theta(x)} \qquad \frac{\Theta\,,x : \tau \overset{F_\omega}{\vdash} e : \tau'}{\Theta \overset{F_\omega}{\vdash} \lambda x : \tau.e : \tau \to \tau'} \qquad \frac{\Theta \overset{F_\omega}{\vdash} e_1 : \tau' \to \tau \qquad \Theta \overset{F_\omega}{\vdash} e_2 : \tau'}{\Theta \overset{F_\omega}{\vdash} e_1\ e_2 : \tau} \qquad \frac{\overline{\Theta \overset{F_\omega}{\vdash} e : \tau} \qquad \Theta \overset{F_\omega}{\vdash} \square}{\Theta \overset{F_\omega}{\vdash} \{\overline{l = e}\} : \{\overline{l : \tau}\}}$$

$$\frac{\Theta \overset{F_\omega}{\vdash} e : \{l : \tau, \overline{l' : \tau'}\}}{\Theta \overset{F_\omega}{\vdash} e.l : \tau} \qquad \frac{\Theta\,,\alpha : \kappa \overset{F_\omega}{\vdash} e : \tau}{\Theta \overset{F_\omega}{\vdash} \lambda\alpha : \kappa.e : \forall\alpha : \kappa.\tau} \qquad \frac{\Theta \overset{F_\omega}{\vdash} e : \forall\alpha : \kappa.\tau' \qquad \Theta \overset{F_\omega}{\vdash} \tau : \kappa}{\Theta \overset{F_\omega}{\vdash} e\ \tau : \tau'[\tau \mapsto \alpha]}$$

$$\frac{\Theta \overset{F_\omega}{\vdash} \tau : \kappa \qquad \Theta \overset{F_\omega}{\vdash} e : \tau'[\tau \mapsto \alpha] \qquad \Theta \overset{F_\omega}{\vdash} \exists\alpha : \kappa.\tau' : \Omega}{\Theta \overset{F_\omega}{\vdash} \underset{\exists}{\mathsf{pack}}\,\alpha : \kappa.\tau'\ \langle \tau, e \rangle : \exists\alpha : \kappa.\tau'}$$

$$\frac{\Theta \overset{F_\omega}{\vdash} e_1 : \exists\alpha : \kappa.\tau' \qquad \Theta\,,\alpha : \kappa\,,x : \tau' \overset{F_\omega}{\vdash} e_2 : \tau \qquad \Theta \overset{F_\omega}{\vdash} \tau : \Omega}{\Theta \overset{F_\omega}{\vdash} \mathsf{unpack}\ \langle \alpha, x \rangle = e_1\ \mathsf{in}\ e_2 : \tau}$$

Figure 17: Typing rules of F^ω

is a shortcut for :

$$\begin{aligned} Y.x &: (\mathsf{val} : (\approx \tau)) \rightsquigarrow [\tau] \\ Y.t &: (\mathsf{type} \approx \tau) \rightsquigarrow [= \tau : \star] \\ Y.X &: (\mathsf{module} : \mathcal{R}) \rightsquigarrow \Sigma \\ Y.A &: (\mathsf{module\ type} = \mathcal{S}) \rightsquigarrow [= \Pi] \end{aligned}$$

All judgments are extensions of their canonical counterparts, with additional conditions added to manage the F^ω terms. We give a brief overview of the lookup, wellformedness, and canonification judgments.

Lookup The elaborated lookup operation follows the canonical one. Functor arguments and declarations are saved in the environment with both their canonical and F^ω signatures which makes the rules like E-Lkp-FctArg and E-Lkp-Mod easy. For the projection rules, the fields are looked up in both the canonical and

Signatures

$$\Pi := \exists\overline{\alpha}.\Sigma \qquad \text{(abstract signatures)}$$

$$\Sigma := [\![\tau]\!] \mid [\![= \tau : \star]\!] \mid [\![= \Pi]\!] \mid \{\overline{l_X : \Sigma}\} \mid \forall\alpha.\Sigma \to \Pi \qquad \text{(concrete signatures)}$$

Types

$$[\![\tau]\!] := \{\mathsf{val} : \tau\} \qquad \text{(value)}$$

$$[\![= \tau : \star]\!] := \{\mathsf{typ} : \forall\beta : (\star \to \star)\,.\beta\ \tau \to \beta\ \tau\} \qquad \text{(type)}$$

$$[\![= \Pi]\!] := \{\mathsf{sig} : \Pi \to \Pi\} \qquad \text{(module type)}$$

Terms

$$[\![e]\!] := \{\mathsf{val} = e\} \qquad \text{(value)}$$

$$[\![\tau : \star]\!] := \{\mathsf{typ} = \lambda\tau : (\star \to \star)\,.\lambda x : \tau\ x\}\ type \qquad (.)$$

$$[\![\Pi]\!] := \{\mathsf{sig} = \lambda x : \Pi.x\} \qquad \text{(module type)}$$

Figure 18: Encoded signatures (as F_ω types) with the corresponding terms. Three special keywords are reserved (`val`, `typ`, `sig`).

Encoded signature

$$\Pi = \exists\alpha \left\{ \begin{array}{l} \ell_X : \left\{ \begin{array}{l} \ell_t : [\![= \alpha : \star]\!] \\ \ell_v : [\![\alpha]\!] \end{array} \right\} \\ \ell_u : [\![= (\alpha \times \mathtt{bool}) : \star]\!] \end{array} \right\}$$

Source Code

```
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 module

$$e = \left(\begin{array}{l} \mathsf{unpack}\langle\alpha_1, y_1\rangle = \mathsf{pack}\,\langle \mathtt{int}, \{\ell_X = \{\ell_t = [\![\mathtt{int} : \star]\!], \ell_v = [\![42]\!]\}\}\rangle\ \mathsf{in} \\ \mathsf{unpack}\langle\varnothing, y_2\rangle = (\mathsf{let}\, X.t, X.v = y_1.\ell_t, y_1.\ell_v\ \mathsf{in}\ \{\ell_u = [\![(\alpha_1 \times \mathtt{bool}) : \star]\!]\})\ \mathsf{in} \\ \mathsf{pack}\,\langle\alpha_1, \{\ell_X = (y_1.\ell_X), \ell_u = (y_2.\ell_u)\}\rangle \end{array} \right)$$

Figure 19: 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. Regarding the typing rules of F^ω, we have $\varnothing \overset{F_\omega}{\vdash} e : \Pi$

F^ω signatures, as in the rule C-Lkp-Proj-Mod. The full set of rules can be found in Figure 55.

E-Lkp-FctArg
$$\frac{(\mathsf{module}\, X : \mathcal{R} \rightsquigarrow \Sigma) \in \Gamma}{\Gamma \vdash X \triangleright \mathcal{R} \rightsquigarrow \Sigma}$$

E-Lkp-Mod
$$\frac{(Y.X : \mathsf{module} : \mathcal{R} \rightsquigarrow \Sigma) \in \Gamma}{\Gamma \vdash Y.X \triangleright \mathcal{R} \rightsquigarrow \Sigma}$$

E-Lkp-Proj-Mod
$$\frac{\Gamma \vdash P \triangleright \mathtt{sig}\ \overline{\mathcal{D}}\ \mathtt{end} \rightsquigarrow \Sigma \qquad (\mathsf{module}\, X : \mathcal{R}) \in D \qquad \Sigma.l_X = \Sigma'}{\Gamma \vdash P.X \triangleright \mathcal{R} \rightsquigarrow \Sigma'}$$

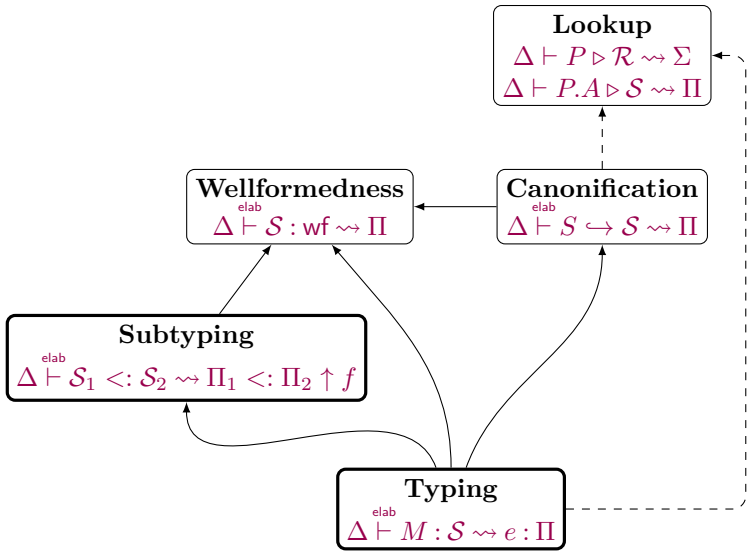


Figure 20: Structure of judgments for the elaborated system

$$
\begin{aligned}
\Delta ::= {} & \varepsilon && \text{(Empty)}\\
| {} & \Delta\,, \overline{\alpha} && \text{(Abstract types)}\\
| {} & \Delta\,, (\mathsf{module}\, X : \mathcal{R} \rightsquigarrow \Sigma) && \text{(Functor Argument)}\\
| {} & \Delta\,, (Y.Z : \mathcal{D} \rightsquigarrow \Sigma) && \text{(Declaration)}
\end{aligned}
$$

Figure 21: Syntax of the hybrid environments. The existential identifiers are common for the canonical and elaborated parts.

Wellformedness Again, the elaborated wellformedness rules follow closely the canonical ones. The main difference comes from the fact that encoded signatures do not distinguish between signatures and declarations: each declaration is encoded as a single-field signature and lists of declarations are encoded as signatures. This can be seen in the rules E-Wf-Val or E-Wf-Type and in the sequence rule E-Wf-Seq where F^ω signatures are merged. We show here only example rules for the wellformedness of declarations, the full set of rules can be found in Appendix A.2.

E-Wf-Val

$$\frac{\Delta \overset{\text{elab}}{\vdash} \tau : \mathsf{wf} \qquad Y.x \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y \mathsf{val}\, x : (\approx \tau) : \mathsf{wf} \rightsquigarrow \{l_x : [\tau]\}}$$

E-Wf-Type

$$\frac{\Delta \overset{\text{elab}}{\vdash} \tau : \mathsf{wf} \qquad Y.t \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y \mathsf{type}\, t \approx \tau : \mathsf{wf} \rightsquigarrow \{l_t : [= \tau : \star]\}}$$

E-Wf-Mod

$$\frac{\Delta \overset{\text{elab}}{\vdash} \mathcal{R} : \mathsf{wf} \rightsquigarrow \Sigma \qquad Y.X \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y (\mathsf{module}\, X : \mathcal{R}) : \mathsf{wf} \rightsquigarrow \{l_X : \Sigma\}}$$

E-Wf-Seq

$$\frac{\Delta \overset{\text{elab}}{\vdash}_Y \mathcal{D}_1 : \mathsf{wf} \rightsquigarrow \{l_{Z_1} : \Sigma_1\} \qquad \Delta\,, (Y.Z_1 : \mathcal{D}_1 \rightsquigarrow \Sigma_1) \overset{\text{elab}}{\vdash}_Y \overline{\mathcal{D}} : \mathsf{wf} \rightsquigarrow \{\overline{l_Z : \Sigma}\}}{\Delta \overset{\text{elab}}{\vdash}_Y (\mathcal{D}_1\,, \overline{\mathcal{D}}) : \mathsf{wf} \rightsquigarrow \{l_{Z_1} : \Sigma_1, \overline{l_Z : \Sigma}\}}$$

Canonification As the canonical types are represented as F^ω types, the elaborated canonification of types $\Delta \overset{\text{elab}}{\vdash} T \hookrightarrow \tau$ is the same as the canonical one. For declarations and signatures, the canonification produces encoded declarations and signatures that correspond directly to their canonical counterparts. As they are F^ω *types* (and not terms), the existentials can be manipulated directly (whereas terms would need the explicit `pack`/`unpack` constructs). This is for instance the case in the rules E-Cf-Mod and E-Cf-TypeAbs shown below, whereas E-Cf-Type depends on the canonification of types.

E-Cf-Type

$$\frac{\Delta \overset{\text{elab}}{\vdash} T \hookrightarrow \tau \qquad Y.t \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y \mathsf{type}\, t = T \hookrightarrow \mathsf{type}\, t \approx \tau \rightsquigarrow \{l_t : [= \tau : \star]\}}$$

E-Cf-TypeAbs

$$\frac{\Delta : \mathsf{wf} \qquad Y.t \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y \mathsf{type}\, t \hookrightarrow \exists \alpha.\, \mathsf{type}\, t (\approx \alpha) \rightsquigarrow \exists \alpha.\, \{l_t : [= \alpha : \star]\}}$$

E-Cf-Mod

$$\frac{\Delta \overset{\text{elab}}{\vdash} S \hookrightarrow \exists \overline{\alpha}.\mathcal{R} \rightsquigarrow \exists \overline{\alpha}.\Sigma \qquad Y.X \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y (\mathsf{module}\, X : S) \hookrightarrow \exists \overline{\alpha}.\, (\mathsf{module}\, X : \mathcal{R}) \rightsquigarrow \exists \overline{\alpha}.\, \{l_X : \Sigma\}}$$

The full sets of rules can be found in Appendix C.3.

5.3 Subtyping

As we use a version of F^ω without subtyping, the subtyping judgment $\Delta \overset{\text{elab}}{\vdash} \mathcal{S}_1 <: \mathcal{S}_2 \rightsquigarrow \Pi_1 <: \Pi_2 \uparrow f$ produces an *subtyping function* f: a conversion function from the type of the first term to the type of the second ($\Pi_1 \to \Pi_2$). The nature of this subtyping function has interesting impacts on the compilation of modules. Indeed, when we consider abstraction-only subtyping, as the memory representation of modules (the terms) are the same (they are stripped of types), we expect this function to be η-β-convertible to the identity. When it is not the case (as with reordering or deletion of fields), the subtyping function is not *code-free*, and this would correspond to a specific action from the compiler (copy, access table, etc.).

The subtyping rules are made longer by the construction of the subtyping function, but mostly follow the canonical ones. Contrary to the wellformedness and canonification judgments, the subtyping on declarations is not done with single-field signatures, but directly with the encoding of the field ($[\![\tau]\!]$, $[\![= \tau : \star]\!]$, $[\![= \Pi]\!]$, etc.). The identifier is not needed in the elaborated part (as F^ω records are not dependent, fields do not refer to each other), so we stripped it for simplicity. This can be seen in the rules for declarations below.

E-Sub-Val

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

E-Sub-Type

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

E-Sub-Mod

$$\frac{\Delta \overset{\text{elab}}{\vdash} \mathcal{R} <: \mathcal{R}' \rightsquigarrow \Sigma <: \Sigma' \uparrow f}{\Delta \overset{\text{elab}}{\vdash} (\mathsf{module}\, X : \mathcal{R}) <: (\mathsf{module}\, X : \mathcal{R}') \rightsquigarrow \Sigma <: \Sigma' \uparrow f}$$

E-Sub-ModType

$$\frac{\Delta \overset{\text{elab}}{\vdash} \mathcal{S} <: \mathcal{S}' \rightsquigarrow \Pi <: \Pi' \uparrow f \qquad \Delta \overset{\text{elab}}{\vdash} \mathcal{S}' <: \mathcal{S} \rightsquigarrow \Pi' <: \Pi \uparrow f'}{\Delta \overset{\text{elab}}{\vdash} (\mathsf{module\ type}\, A = \mathcal{S}) <: (\mathsf{module\ type}\, A = \mathcal{S}') \rightsquigarrow \Pi <: \Pi' \uparrow \lambda\, (x : [= \Pi])\, .\, [\Pi']}$$

Two of the subtyping rules for signatures are given below. The way existentials are dealt with in E-Sub-Abs is similar in the canonical and existential parts, with the subtyping function using the construction $\mathsf{unpack}\,\langle\cdot,\cdot\rangle = \cdot\ \mathsf{in}\ \mathsf{pack}\,\langle\cdot,\cdot\rangle$ to reseal the signature with the correct list of existentials. For E-Sub-Sig, the encoded signatures are split and reconstructed using the declaration subtyping functions.

E-Sub-Abs

$$\frac{\Delta\,,\overline{\alpha} \overset{\text{elab}}{\vdash} \mathcal{R} <: \mathcal{R}'[\overline{\alpha}' \mapsto \overline{\tau}] \rightsquigarrow \Sigma <: \Sigma'[\overline{\alpha}' \mapsto \overline{\tau}] \uparrow f}{\Delta \overset{\text{elab}}{\vdash} \exists\overline{\alpha}.\mathcal{R} <: \exists\overline{\alpha}'.\mathcal{R}' \rightsquigarrow \exists\overline{\alpha}.\Sigma <: \exists\overline{\alpha}'.\Sigma' \uparrow \lambda\,(x : \exists\overline{\alpha}.\Sigma)\,.\ \mathsf{unpack}\,\langle\overline{\alpha}, y\rangle = x\ \mathsf{in}\ \mathsf{pack}\,\langle\overline{\tau}, f\ y\rangle}$$

E-Sub-Sig

$$\frac{\overline{\mathcal{D}_0} \subseteq \overline{\mathcal{D}} \qquad \Delta \overset{\text{elab}}{\vdash} \overline{\mathcal{D}} : \mathsf{wf} \rightsquigarrow \left\{\overline{l_{Z'} : \Sigma_0}, \overline{l_Z : \Sigma}\right\} \qquad \Delta \overset{\text{elab}}{\vdash} \overline{\mathcal{D}'} : \mathsf{wf} \rightsquigarrow \left\{\overline{l_{Z'} : \Sigma'}\right\} \qquad \frac{}{\Delta \overset{\text{elab}}{\vdash} \mathcal{D}_0 <: \mathcal{D}' \rightsquigarrow \Sigma_0 <: \Sigma' \uparrow f}}{\Delta \overset{\text{elab}}{\vdash} \mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} <: \mathtt{sig}_Y\ \overline{\mathcal{D}'}\ \mathtt{end} \rightsquigarrow \left\{\overline{l_{Z'} : \Sigma_0}, \overline{l_Z : \Sigma}\right\} <: \left\{\overline{l_{Z'} : \Sigma'}\right\} \uparrow \lambda\left(x : \left\{\overline{l_{Z'} : \Sigma_0}, \overline{l_Z : \Sigma}\right\}\right).\ \left\{\overline{l_{Z'} = f(x.l_{Z'})}\right\}}$$

The full set of subtyping rules can be found in Appendix C.4.

5.4 Typing

In the typing rules, explicit terms of F^ω are produced for the module expressions and types for signatures. To represent module variables, we introduce a translation of paths into F^ω identifiers:

$$\begin{aligned}
\lfloor Y.Z \rfloor &::= \mathtt{Y_Z} \\
\lfloor X \rfloor &::= \mathtt{X} \\
\lfloor P.X \rfloor &::= \lfloor P \rfloor\mathtt{_X} \\
\lfloor P.A \rfloor &::= \lfloor P \rfloor\mathtt{_A}
\end{aligned}$$

Binding typing rules As for wellformedness and canonification, we encode an individual binder using a single-field declaration (record). The extrusion of existential types requires the $\mathsf{unpack}\,\langle\cdot,\cdot\rangle = \cdot\ \mathsf{in}\ \cdot$ and $\mathsf{pack}\,\langle\cdot,\cdot\rangle$ constructs, as in E-Typ-Mod. In the sequence rule E-Typ-Seq, the two subdeclarations are merged by a substitution for renaming: an identifier of the form $\lfloor Y.Z_1 \rfloor$ used in e_2 correspond to the fields l_{Z_1} of the

record y_1.

E-Typ-Let

$$\frac{\Delta \overset{\text{elab}}{\vdash} E : T \rightsquigarrow e : \tau \qquad Y.x \notin \Delta}{\Delta \overset{\text{elab}}{\vdash} (\mathsf{let}\, x = E) : (\mathsf{val}\, x : (\approx \tau)) \rightsquigarrow \{l_x = e\} : \{l_x : [\tau]\}}$$

E-Typ-Type

$$\frac{\Delta \overset{\text{elab}}{\vdash} T \hookrightarrow \tau \qquad Y.t \notin \Delta}{\Delta \overset{\text{elab}}{\vdash} \mathsf{type}\, t = T : \mathsf{type}\, t \approx \tau \rightsquigarrow \{l_t = [\tau : \star]\} : \{l_t : [= \tau : \star]\}}$$

E-Typ-Mod

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

E-Typ-Seq

$$\frac{\Delta \overset{\text{elab}}{\vdash} B_1 : \exists\overline{\alpha}_1.\overline{\mathcal{D}_1} \rightsquigarrow e_1 : \exists\overline{\alpha}_1.\, \{\overline{l_{Z_1} : \Sigma_1}\} \qquad \Delta\,, \overline{\alpha}_1\,, \overline{Y.\mathcal{D}_1 : l_{Z_1} \rightsquigarrow \Sigma_1} \overset{\text{elab}}{\vdash} B_2 : \exists\overline{\alpha}_2.\overline{\mathcal{D}_2} \rightsquigarrow e_2 : \exists\overline{\alpha}_2.\, \{\overline{l_{Z_2} : \Sigma_2}\}}{\Delta \overset{\text{elab}}{\vdash} B_1; B_2 : \exists\overline{\alpha}_1\overline{\alpha}_2.\overline{\mathcal{D}_1} +\!\!+ \overline{\mathcal{D}_2} \rightsquigarrow \rightsquigarrow : \begin{array}{l} \mathsf{unpack}\, \langle \overline{\alpha}_1, y_1\rangle = e_1 \;\mathsf{in} \\ \mathsf{unpack}\, \langle \overline{\alpha}_2, y_2\rangle = \\ \quad \mathsf{let}\, \overline{\lfloor Y.Z_1 \rfloor = y_1.l_{Z_1}} \;\mathsf{in}\; e_2 \;\; \mathsf{in} \\ \mathsf{pack}\, \langle \overline{\alpha}_1\overline{\alpha}_2, \{\overline{l_{Z_1} : y_1.l_{Z_1}}, \overline{l_{Z_2} : y_2}\}\rangle \end{array} \quad : \exists\overline{\alpha}_1\overline{\alpha}_2.\, \{\overline{l_{Z_1} : \Sigma_1}, \overline{l_{Z_2} : \Sigma_2}\}}$$

Module expressions typing We present bellow some of the typing rules for module expressions. The rule E-Typ-Var uses the conversion from paths to identifiers. As the typing of binding produces encoded signatures (and not encoded *list of declarations*), the rule E-Typ-Struct is made very simple. In the rule E-Typ-Sig (signature ascription), the subtyping function is used to convert the type of the variable $\lfloor P \rfloor$ from Σ_2 to $\Sigma_2[\overline{\alpha} \mapsto \overline{\tau}]$ and the result is resealed with the existentials $\overline{\alpha}$. The rule E-Typ-Fct show how polymorphic signatures $\forall\overline{\alpha} : \Sigma \to \Pi.$ appear naturally from the covariant position of existential signatures.

Finally, the rule E-Typ-Proj shows the unsealing and resealing of existentials.

$$\text{E-Typ-Var}\quad \frac{\Delta : \mathsf{wf} \qquad \Delta \vdash P \triangleright \mathcal{R} \rightsquigarrow \Sigma}{\Delta \overset{\text{elab}}{\vdash} P : \mathcal{R} \rightsquigarrow \lfloor P \rfloor : \Sigma}$$

$$\text{E-Typ-Sig}\quad \frac{\Delta \overset{\text{elab}}{\vdash} S \hookrightarrow \exists\overline{\alpha}.\mathcal{R}_1 \rightsquigarrow \exists\overline{\alpha}.\Sigma_1 \qquad \Delta \overset{\text{elab}}{\vdash} P : \mathcal{R}_2 \rightsquigarrow \lfloor P \rfloor : \Sigma_2 \qquad \Delta \overset{\text{elab}}{\vdash} \mathcal{R}_2 <: \mathcal{R}_1[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow \Sigma_2 <: \Sigma_1[\overline{\alpha} \mapsto \overline{\tau}] \uparrow f}{\Delta \overset{\text{elab}}{\vdash} (P : S) : \mathcal{S} \rightsquigarrow \mathsf{pack}\,\langle \overline{\tau}, f\lfloor P \rfloor \rangle : \exists\overline{\alpha}.\Sigma_1}$$

$$\text{E-Typ-Struct}\quad \frac{Y \notin \Delta \qquad \Delta \overset{\text{elab}}{\vdash} B : \exists\overline{\alpha}.\overline{\mathcal{D}} \rightsquigarrow e : \exists\overline{\alpha}.\Sigma}{\Delta \overset{\text{elab}}{\vdash} \mathtt{struct}_Y\ B\ \mathsf{end} : \exists\overline{\alpha}.\mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathsf{end} \rightsquigarrow e : \exists\overline{\alpha}.\Sigma}$$

$$\text{E-Typ-Fct}\quad \frac{X \notin \Delta \qquad \Delta \overset{\text{elab}}{\vdash} S_a \hookrightarrow \exists\overline{\alpha}.\mathcal{R} \rightsquigarrow \exists\overline{\alpha}.\Sigma \qquad \Delta\,,\overline{\alpha}\,,\mathsf{module}\,X : \mathcal{R} \rightsquigarrow \Sigma \overset{\text{elab}}{\vdash} M : \mathcal{S} \rightsquigarrow e : \Pi}{\Delta \overset{\text{elab}}{\vdash} ((X : S_a) \to M) : \forall\overline{\alpha}.\,(X : \mathcal{R}) \to \mathcal{S} \rightsquigarrow \lambda\overline{\alpha}.\lambda\,(x : \Sigma)\,.\,e : \forall\overline{\alpha}.\Sigma \to \Pi}$$

$$\text{E-Typ-Proj}\quad \frac{\Delta \overset{\text{elab}}{\vdash} M : \exists\overline{\alpha}.\mathtt{sig}_Y\ \overline{\mathcal{D}}_1\,,\mathsf{module}\,X : \mathcal{R}\,,\overline{\mathcal{D}}_2\ \mathsf{end} \rightsquigarrow e : \exists\overline{\alpha}.\left\{\overline{l_{Z_1} : \Sigma_1}, l_X : \Sigma, \overline{l_{Z_2} : \Sigma_2}\right\} \qquad \Delta \overset{\text{elab}}{\vdash} \exists\overline{\alpha}.\mathcal{R} : \mathsf{wf} \rightsquigarrow \exists\overline{\alpha}.\Sigma}{\Delta \overset{\text{elab}}{\vdash} M.X : \exists\overline{\alpha}.\mathcal{R} \rightsquigarrow \mathsf{unpack}\,\langle \overline{\alpha}, y\rangle = e\ \mathsf{in}\ \mathsf{pack}\,\langle \overline{\alpha}, y.l_X\rangle : \exists\overline{\alpha}.\Sigma}$$

The full set of rules is given in Appendix C.5. This completes the presentation of the *elaborated* system. We finish the section by presenting some key results.

5.5 Correctness and link with the canonical system

In this section we present the correctness result of the elaboration (generated terms are well-typed) and the link with the canonical system (the elaboration is correct for the canonical rules and (more importantly) not restrictive). This requires some technical extensions, but the proofs are mostly straightforward by induction.

5.5.1 Correctness

The key idea of the correctness result is the following: when typing a module expression $\Delta \overset{\text{elab}}{\vdash} M : \mathcal{S} \rightsquigarrow e : \Pi$, the term e has indeed the type Π. The technical issue only comes from the fact that Δ is not an F^ω environment, we need to convert it first. We introduce the transformation of environments:

$$\begin{aligned}
\omega(\varepsilon) &= \varepsilon \\
\omega(\Delta\,,\overline{\alpha}) &= \omega(\Delta)\,,\overline{\alpha} \\
\omega(\Delta\,,Y.Z : \mathcal{D} \rightsquigarrow \Sigma) &= \omega(\Delta)\,,\lfloor Y.Z \rfloor : \Sigma \\
\omega(\Delta\,,\mathsf{module}\,X : \mathcal{R} \rightsquigarrow \Sigma) &= \omega(\Delta)\,,\lfloor X \rfloor : \Sigma
\end{aligned}$$

Then we state the correctness of subtyping and typing. Both results are proved by a straightforward induction.

Lemma 5 (Correctness of subtyping)**.** Given an environment Δ, two signatures $\mathcal{S}$ and $\mathcal{S}'$, two encoded signatures Π and Π', and a subtyping function f, we have:

$$\Delta \overset{\text{elab}}{\vdash} \mathcal{S} <: \mathcal{S}' \rightsquigarrow \Pi <: \Pi' \uparrow f \implies \omega(\Delta) \overset{F_\omega}{\vdash} f : \Pi \to \Pi' \tag{17}$$

Theorem 7 (Correctness of elaboration)**.** Given an environment Δ, a module expression M, a signature $\mathcal{S}$, a term e and a encoded signature Π, we have:

$$\Delta \overset{\text{elab}}{\vdash} M : \mathcal{S} \rightsquigarrow e : \Pi \implies \omega(\Delta) \overset{F_\omega}{\vdash} e : \Pi. \tag{18}$$

5.5.2 Link with the canonical system, backwards

In this subsection we show that the elaborated system is indeed an subset of the canonical system: all typing derivations in the elaborated system can be expressed in the canonical one by stripping the F^ω parts. We first define an simple *stripping* operator for elaborated environments:

$$\begin{aligned}
\mu(\varepsilon) &= \varepsilon \\
\mu(\Delta\,, \overline{\alpha}) &= \mu(\Delta)\,, \overline{\alpha} \\
\mu(\Delta\,, Y.Z : \mathcal{D} \rightsquigarrow \Sigma) &= \mu(\Delta)\,, Y.Z : \mathcal{D} \\
\mu(\Delta\,, \mathsf{module}\, X : \mathcal{R} \rightsquigarrow \Sigma) &= \mu(\Delta)\,, X : \mathcal{R}
\end{aligned}$$

By an easy induction, we have the following result:

Theorem 8 (From elaborated to canonical typing)**.** Given an environment Δ, a module expression M, a signature $\mathcal{S}$, a term e and a encoded signature Π, we have:

$$\Delta \overset{\text{elab}}{\vdash} M : \mathcal{S} \rightsquigarrow e : \Pi \implies \mu(\Delta) \overset{\text{can}}{\vdash} M : \mathcal{S}. \tag{19}$$

5.5.3 Link with the canonical system, forward

In this subsection we show that the elaboration is actually not *restrictive*: all derivations in the canonical system can be expressed in the elaborated one. We first define an *elaboration* operator for canonical signatures:

$$\begin{aligned}
\mathsf{e}\,(\exists\overline{\alpha}.\mathcal{R}) &= \exists\overline{\alpha}.\mathsf{e}\,(\mathcal{R}) \\
\mathsf{e}\,(\forall\overline{\alpha}.\,(X : \mathcal{R}) \to \mathcal{S}) &= \forall\overline{\alpha}.\,(X : \mathsf{e}\,(\mathcal{R})) \to \mathsf{e}\,(\mathcal{S}) \\
\mathsf{e}\,\big(\mathtt{sig}\ \overline{\mathcal{D}}\ \mathtt{end}\big) &= \{\overline{\mathsf{e}\,(\mathcal{D})}\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{e}\,(\mathsf{val}\, x : (\approx \tau)) &= \{l_x : [\tau]\} \\
\mathsf{e}\,(\mathsf{type}\, t \approx \tau) &= \{l_t : [= \tau : \star]\} \\
\mathsf{e}\,(\mathsf{module}\, X : \mathcal{R}) &= \{l_X : \mathsf{e}\,(\mathcal{R})\} \\
\mathsf{e}\,(\mathsf{module\ type}\, A = \mathcal{S}) &= \{l_A : [= \mathsf{e}\,(\mathcal{S})]\}
\end{aligned}$$

For a given signature $\mathcal{S}$, we will write $\Pi_{\mathcal{S}}$ for $\mathsf{e}\,(\mathcal{S})$ and $\Sigma_{\mathcal{R}}$ for $\mathsf{e}\,(\mathcal{R})$. We extend the operator to environments:

$$\begin{aligned}
\mathsf{e}\,(\varepsilon) &= \varepsilon \\
\mathsf{e}\,(\Gamma\,, \overline{\alpha}) &= \mathsf{e}\,(\Gamma)\,, \overline{\alpha} \\
\mathsf{e}\,(\Gamma\,, \mathsf{module}\, X : \mathcal{R}) &= \mathsf{e}\,(\Gamma)\,, (\mathsf{module}\, X : \mathcal{R} \rightsquigarrow \mathsf{e}\,(\mathcal{R})) \\
\mathsf{e}\,(\Gamma\,, Y.Z : \mathcal{D}) &= \mathsf{e}\,(\Gamma)\,, (Y.Z : \mathcal{D} \rightsquigarrow \mathsf{e}\,(\mathcal{D}))
\end{aligned}$$

Again, will we write Δ_Γ for $\mathsf{e}\,(\Gamma)$. We have the following results:

$$\begin{aligned}
(\mathsf{module}\, X : \mathcal{R}) \in \overline{\mathcal{D}} &\implies \mathsf{e}\,\big(\mathtt{sig}\ \overline{\mathcal{D}}\ \mathtt{end}\big) = \{\ldots, l_X : \mathsf{e}\,(\mathcal{R}), \ldots\} \\
(\mathsf{module\ type}\, A = \mathcal{S}) \in \overline{\mathcal{D}} &\implies \mathsf{e}\,\big(\mathtt{sig}\ \overline{\mathcal{D}}\ \mathtt{end}\big) = \{\ldots, l_A : [= \mathsf{e}\,(\mathcal{R})], \ldots\} \\
\forall \alpha,\ \alpha \in \Gamma &\iff \alpha \in \Delta_\Gamma \\
\forall Z,\ Z \in \Gamma &\iff Z \in \Delta_\Gamma
\end{aligned}$$

Then we state the intermediary lemmas. All results can be shown by induction.

Lemma 6 (Elaboration of lookup)**.** Given an environment Γ, a path P, a module type identifier A, and a signature $\mathcal{R}$, we have:

$$\Gamma \vdash P \triangleright \mathcal{R} \implies \Delta_\Gamma \vdash P \triangleright \mathcal{R} \rightsquigarrow \Sigma_\mathcal{R} \tag{20}$$

$$\Gamma \vdash P.A \triangleright \mathcal{S} \implies \Delta_\Gamma \vdash P.A \triangleright \mathcal{S} \rightsquigarrow \Pi_\mathcal{S} \tag{21}$$

Lemma 7 (Elaboration of wellformedness)**.** Given an environment Γ and a signature $\mathcal{S}$, we have:

$$\Gamma : \mathsf{wf} \implies \Delta_\Gamma : \mathsf{wf} \tag{22}$$

$$\Gamma \overset{\mathsf{can}}{\vdash} \mathcal{S} : \mathsf{wf} \implies \Delta_\Gamma \overset{\mathsf{elab}}{\vdash} \mathcal{S} : \mathsf{wf} \rightsquigarrow \Pi_\mathcal{S} \tag{23}$$

Lemma 8 (Elaboration of canonification)**.** Given an environment Γ, a source signature S, and a signature $\mathcal{S}$, we have:

$$\Gamma \overset{\mathsf{can}}{\vdash} S \hookrightarrow \mathcal{S} \implies \Delta_\Gamma \overset{\mathsf{elab}}{\vdash} S \hookrightarrow \mathcal{S} \rightsquigarrow \Pi_\mathcal{S} \tag{24}$$

Lemma 9 (Elaboration of subtyping)**.** Given an environment Γ and two signatures $\mathcal{S}$ and $\mathcal{S}'$, we have:

$$\Gamma \overset{\mathsf{can}}{\vdash} \mathcal{S}_1 <: \mathcal{S}_2 \implies \exists f,\ \Delta_\Gamma \overset{\mathsf{elab}}{\vdash} \mathcal{S}_1 <: \mathcal{S}_2 \rightsquigarrow \Pi_{\mathcal{S}_1} <: \Pi_{\mathcal{S}_2} \uparrow f \tag{25}$$

Theorem 9 (Elaboration of typing)**.** Given an environment Γ, a module expression M, and a signature $\mathcal{S}$, we have:

$$\Gamma \overset{\mathsf{can}}{\vdash} M : \mathcal{S} \implies \exists e,\ \Delta_\Gamma \overset{\mathsf{elab}}{\vdash} M : \mathcal{S} \rightsquigarrow e : \Pi_\mathcal{S} \tag{26}$$

This completes the presentation of the elaborated system. More than only a formal support for the canonical one, the elaboration was the key design inspiration, notably for existential types.

Conclusion

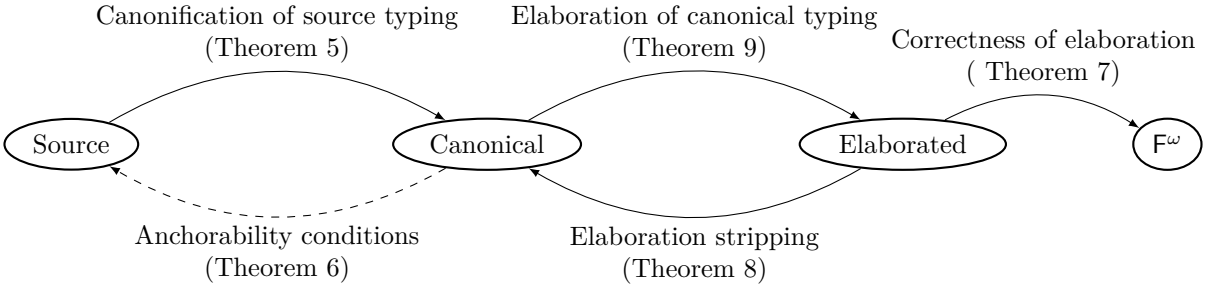


Figure 22: Links between the source, canonical and elaborated systems

ML-Module systems are known for being a well-studied but complex topic. The battle-hardened, path-based, OCaml approach as proven to be successful, 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. Those limitations are at the heart of the need for *ad-hoc* and complex fixes (strengthening, equivalence). We introduced the canonical system as a more expressive yet simpler language, equipped with the right construction (existential types) to distinguish between existence and identity and solve the main issues of the source system. While still being close to the OCaml source, the canonical presentation is very easy to elaborate in F^ω, which provides formal guarantees. It also allows us to give a clear description of the limitations of the source presentation through the anchorability conditions. As a middle-point between usability and formalism, the canonical system is both a comprehensive description and a framework for building new features and improve the algorithms (specifically for the *solvable* cases of signature avoidance) of the current OCaml typechecker.

Future work We plan to extend the scope of this work in several directions. First, we want to make the module system applicative (tracking equality of types through functor application) by introducing higher-kinds existentials. The missing link of Figure 22, from F^ω to the elaborated system should be investigated to ensure the canonical system enjoys the subreduction property. We hope that the canonical system can be extended to support other features (module aliases, transparent ascription, first class modules, abstract signatures, etc.) which can be hard to describe in the source presentation. As the number of inference rules grows, the need to mechanize the proofs of our presentation will become clearer and clearer. Along the way, improvement to the current OCaml typechecking algorithms can be made, inspired by the canonical approach. If it turns out to be expressive enough, it could serve as a basis for a redesign of a part of the typechecker, which might be a crucial step to provide a clean basis to introduce new features such as modular implicits.

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. *J. Funct. Program.*, 17(4–5):433–471, July 2007. ISSN 0956-7968. doi: 10.1017/S0956796807006429. URL `https://doi.org/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.

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.

A OCaml source system

A.1 Lookup

$$\textsc{S-Lkp-FctArg}\quad \frac{(\mathsf{module}\, X : S) \in \Gamma}{\Gamma \vdash X \triangleright S}$$

$$\textsc{S-Lkp-Mod}\quad \frac{Y.(\mathsf{module}\, X : S) \in \Gamma}{\Gamma \vdash Y.X \triangleright S}$$

$$\textsc{S-Lkp-Sig}\quad \frac{Y.(\mathsf{module\ type}\, A = S) \in \Gamma}{\Gamma \vdash Y.A \triangleright S}$$

$$\textsc{S-Lkp-Proj-Mod}\quad \frac{\Gamma \vdash P \triangleright \mathtt{sig}_Y\ \overline{D}\ \mathtt{end} \qquad (\mathsf{module}\, X : S) \in \overline{D}}{\Gamma \vdash P.X \triangleright S[Y \mapsto P]}$$

$$\textsc{S-Lkp-Proj-Sig}\quad \frac{\Gamma \vdash P \triangleright \mathtt{sig}_Y\ \overline{D}\ \mathtt{end} \qquad (\mathsf{module\ type}\, A = S) \in \overline{D}}{\Gamma \vdash P.A \triangleright S[Y \mapsto P]}$$

$$\textsc{S-Lkp-Alias}\quad \frac{\Gamma \vdash P \triangleright P'.A \qquad \Gamma \vdash P'.A \triangleright S}{\Gamma \vdash P \triangleright S}$$

Figure 23: $\Gamma \vdash P \triangleright S$ – Lookup rules
Canonical: Figure 37, Elaboration: Figure 55

Invariants

$$\neg(\Gamma \vdash P \triangleright \mathcal{S} \implies \Gamma \overset{\mathsf{src}}{\vdash} \mathcal{S} : \mathsf{wf})$$

A.2 Well-formedness

$$\textsc{S-Wf-Empty}\quad \varepsilon : \mathsf{wf}$$

$$\textsc{S-Wf-FctArg}\quad \frac{\Gamma \overset{\mathsf{src}}{\vdash} S : \mathsf{wf} \qquad X \notin \Gamma}{(\Gamma\,, \mathsf{module}\, X : S) : \mathsf{wf}}$$

Figure 24: $\Gamma : \mathsf{wf}$ – Wellformedness of environments
Canonical: Figure 38, Elaboration: Figure 56

Invariants

$$\Gamma \overset{\mathsf{src}}{\vdash} T : \mathsf{wf} \implies \Gamma : \mathsf{wf}$$

$$\Gamma \overset{\mathsf{src}}{\vdash} S : \mathsf{wf} \implies \Gamma : \mathsf{wf}$$

$$\Gamma \overset{\mathsf{src}}{\vdash}_Y D : \mathsf{wf} \implies \Gamma : \mathsf{wf}$$

$$\Gamma : \mathsf{wf} \wedge \Gamma \vdash P \triangleright S \implies \Gamma \overset{\mathsf{src}}{\vdash} S : \mathsf{wf}$$

$$\Gamma : \mathsf{wf} \wedge \Gamma \vdash P.A \triangleright S \implies \Gamma \overset{\mathsf{src}}{\vdash} S : \mathsf{wf}$$

S-Wf-Sig
$$\frac{\Gamma \overset{\text{src}}{\vdash}_Y \overline{D} : \mathsf{wf}}{\Gamma \overset{\text{src}}{\vdash} \mathtt{sig}_Y\ \overline{D}\ \mathsf{end} : \mathsf{wf}}$$

S-Wf-Funct
$$\frac{\Gamma \overset{\text{src}}{\vdash} S_1 : \mathsf{wf} \qquad \Gamma, \mathsf{module}\, X : S_1 \overset{\text{src}}{\vdash} S_2 : \mathsf{wf} \qquad X \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash} (X : S_1) \to S_2 : \mathsf{wf}}$$

S-Wf-Path
$$\frac{\Gamma \vdash P.A \triangleright S \qquad \Gamma \overset{\text{src}}{\vdash} S : \mathsf{wf}}{\Gamma \overset{\text{src}}{\vdash} P.A : \mathsf{wf}}$$

Figure 25: $\Gamma \overset{\text{src}}{\vdash} S : \mathsf{wf}$ – Wellformedness of signatures
Canonical: Figure 39, Elaboration: Figure 57

S-Wf-Val
$$\frac{\Gamma \overset{\text{src}}{\vdash} T : \mathsf{wf} \qquad Y.x \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_Y \mathsf{val}\, x : T : \mathsf{wf}}$$

S-Wf-Type
$$\frac{\Gamma \overset{\text{src}}{\vdash} T : \mathsf{wf} \qquad Y.t \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_Y \mathsf{type}\, t = T : \mathsf{wf}}$$

S-Wf-TypeAbs
$$\frac{\Gamma : \mathsf{wf} \qquad Y.t \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_Y \mathsf{type}\, t : \mathsf{wf}}$$

S-Wf-Mod
$$\frac{\Gamma \overset{\text{src}}{\vdash} S : \mathsf{wf} \qquad Y.X \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_Y (\mathsf{module}\, X : S) : \mathsf{wf}}$$

S-Wf-ModType
$$\frac{\Gamma \overset{\text{src}}{\vdash} S : \mathsf{wf} \qquad Y.A \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash}_Y (\mathsf{module\ type}\, A = S) : \mathsf{wf}}$$

S-Wf-Empty
$$\frac{\Gamma : \mathsf{wf}}{\Gamma \overset{\text{src}}{\vdash}_Y \varepsilon : \mathsf{wf}}$$

S-Wf-Seq
$$\frac{\Gamma \overset{\text{src}}{\vdash}_Y D_1 : \mathsf{wf} \qquad \Gamma, Y.D_1 \overset{\text{src}}{\vdash}_Y \overline{D} : \mathsf{wf}}{\Gamma \overset{\text{src}}{\vdash}_Y (D_1, \overline{D}) : \mathsf{wf}}$$

Figure 26: $\Gamma \overset{\text{src}}{\vdash}_Y D : \mathsf{wf}$ – Wellformedness of declarations
Canonical: Figure 40, Elaboration: Figure 58

A.3 Equivalence

S-Eqv-Env
$$\frac{Y.(\mathsf{type}\, t = T) \in \Gamma}{\Gamma \vdash Y.t \approx T}$$

S-Eqv-Lkp
$$\frac{\Gamma \vdash P \triangleright \mathtt{sig}_Y\ \overline{D}\ \mathsf{end} \qquad (\mathsf{type}\, t = T) \in D}{\Gamma \vdash P.t \approx T[Y \mapsto P]}$$

S-Eqv-Trans
$$\frac{\Gamma \vdash T_1 \approx T_2 \qquad \Gamma \vdash T_2 \approx T_3}{\Gamma \vdash T_1 \approx T_3}$$

S-Eqv-Sym
$$\frac{\Gamma \vdash T' \approx T}{\Gamma \vdash T \approx T'}$$

S-Eqv-Refl
$$\Gamma \vdash T \approx T$$

S-Eqv-Cgr
$$\frac{\forall i \in [\![0, n]\!],\ \Gamma \vdash T_i \approx T'_i}{\Gamma \vdash Op(T_0, \ldots, T_n) \approx Op(T'_0, \ldots, T'_n)}$$

Figure 27: $\Gamma \vdash T \approx T'$ – Type equivalence

S-Eqv-Funct
$$\frac{\Gamma \vdash S'_a \approx S_a \qquad \Gamma, \mathsf{module}\, X : S'_a \vdash S_r \approx S'_r}{\Gamma \vdash (X : S_a) \to S_r \approx (X : S'_a) \to S'_r}$$

S-Eqv-Lkp
$$\frac{\Gamma \vdash P.A \triangleright S}{\Gamma \vdash P.A \approx S}$$

S-Eqv-Trans
$$\frac{\Gamma \vdash S_1 \approx S_2 \qquad \Gamma \vdash S_2 \approx S_3}{\Gamma \vdash S_1 \approx S_3}$$

S-Eqv-Refl
$$\Gamma \vdash S \approx S$$

S-Eqv-Sym
$$\frac{\Gamma \vdash S' \approx S}{\Gamma \vdash S \approx S'}$$

S-Eqv-Cgr
$$\frac{\Gamma \vdash_Y \overline{D} \approx \overline{D'}}{\Gamma \vdash \mathtt{sig}_Y\ \overline{D}\ \mathtt{end} \approx \mathtt{sig}_Y\ \overline{D'}\ \mathtt{end}}$$

Figure 28: $\Gamma \vdash S \approx S'$ – Signature equivalence

S-Eqv-Trans
$$\frac{\Gamma \vdash_Y D_1 \approx D_2 \qquad \Gamma \vdash_Y D_2 \approx D_3}{\Gamma \vdash_Y D_1 \approx D_3}$$

S-Eqv-Refl
$$\Gamma \vdash_Y D \approx D$$

S-Eqv-Sym
$$\frac{\Gamma \vdash_Y D' \approx D}{\Gamma \vdash_Y D \approx D'}$$

S-Eqv-Mod
$$\frac{\Gamma \vdash S \approx S'}{\Gamma \vdash_Y (\mathsf{module}\, X : S) \approx (\mathsf{module}\, X : S')}$$

S-Eqv-ModType
$$\frac{\Gamma \vdash S \approx S'}{\Gamma \vdash_Y (\mathsf{module\ type}\, A = S) \approx (\mathsf{module\ type}\, A = S')}$$

S-Eqv-Val
$$\frac{\Gamma \vdash T_1 \approx T_2}{\Gamma \vdash_Y (\mathsf{val}\, x : T_1) \approx (\mathsf{val}\, x : T_2)}$$

S-Eqv-TypeAbs
$$\Gamma \vdash_Y (\mathsf{type}\, t) \approx (\mathsf{type}\, t)$$

S-Eqv-Type
$$\frac{\Gamma \vdash T_1 \approx T_2}{\Gamma \vdash_Y (\mathsf{type}\, t = T_1) \approx (\mathsf{type}\, t = T_2)}$$

S-Eqv-Empty
$$\Gamma \vdash \varepsilon \approx \varepsilon$$

S-Eqv-Seq
$$\frac{\Gamma \vdash_Y D_1 \approx D'_1 \qquad \Gamma, Y.D_1 \vdash_Y \overline{D} \approx \overline{D'}}{\Gamma \vdash_Y D_1, \overline{D} \approx D'_1, \overline{D'}}$$

Figure 29: $\Gamma \vdash_Y D \approx D'$ – Declaration equivalence

Invariants

$$\Gamma \overset{\mathsf{src}}{\vdash} T : \mathsf{wf} \wedge \Gamma \vdash T \approx T' \implies \Gamma \overset{\mathsf{src}}{\vdash} T' : \mathsf{wf}$$
$$\Gamma \overset{\mathsf{src}}{\vdash} S : \mathsf{wf} \wedge \Gamma \vdash S \approx S' \implies \Gamma \overset{\mathsf{src}}{\vdash} S' : \mathsf{wf}$$
$$\Gamma \overset{\mathsf{src}}{\vdash}_Y D : \mathsf{wf} \wedge \Gamma \vdash_Y D \approx D' \implies \Gamma \overset{\mathsf{src}}{\vdash}_Y D' : \mathsf{wf}$$

A.4 Strengthening

S-Str-Path
$$\frac{\Gamma \vdash P'.A \triangleright S' \qquad \Gamma \vdash S'/P \rightarrowtriangle S}{\Gamma \vdash P'.A/P \rightarrowtriangle S}$$

S-Str-Sig
$$\frac{\Gamma \vdash \overline{D}/P \rightarrowtriangle \overline{D'}}{\Gamma \vdash \mathtt{sig}_Y\ \overline{D}\ \mathtt{end}/P \rightarrowtriangle \mathtt{sig}_Y\ \overline{D'}\ \mathtt{end}}$$

Figure 30: $\Gamma \vdash P/S \rightarrowtriangle S'$ – Signature strengthening, defined only if $\Gamma \vdash P \triangleright S$

S-Str-Val
$$\Gamma \vdash_Y (\mathsf{val}\, x : T)\, /P \rightarrowtriangle \mathsf{val}\, x : T$$

S-Str-Type
$$\Gamma \vdash_Y (\mathsf{type}\, t = T)\, /P \rightarrowtriangle \mathsf{type}\, t = T$$

S-Str-TypeAbs
$$\Gamma \vdash_Y (\mathsf{type}\, t)\, /P \rightarrowtriangle \mathsf{type}\, t = P.t$$

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

S-Str-ModType
$$\Gamma \vdash_Y (\mathsf{module\ type}\, X = S)\, /P \rightarrowtriangle \mathsf{module\ type}\, X = S$$

S-Str-None
$$\Gamma \vdash_Y D/P \rightarrowtriangle D$$

S-Str-Empty
$$\Gamma \vdash_Y \varepsilon/P \rightarrowtriangle \varepsilon$$

S-Str-Seq
$$\frac{\Gamma \vdash_Y D_1/P \rightarrowtriangle D_1' \qquad \Gamma\,, Y.D_1 \vdash_Y \overline{D}/P \rightarrowtriangle \overline{D'}}{\Gamma \vdash_Y \left(D_1\,, \overline{D}\right)/P \rightarrowtriangle D_1'\,, \overline{D'}}$$

Figure 31: $\Gamma \vdash_Y P/D \rightarrowtriangle D'$ – Declaration strengthening

It's equivalent to push strengthen or un-strengthen declarations in the context, as their only use is to get stored module types, that are left un-strengthen anyway.

A.5 Subtyping

S-Sub-Equiv
$$\frac{\Gamma \vdash T \approx T'}{\Gamma \overset{\mathsf{src}}{\vdash} T <: T'}$$

Figure 32: $\Gamma \overset{\mathsf{src}}{\vdash} T <: T'$ – Types subtyping rules

Invariants

$$\Gamma \overset{\mathsf{src}}{\vdash} S_1 : \mathsf{wf} \wedge \Gamma \overset{\mathsf{src}}{\vdash} S_1 <: S_2 \implies \Gamma \overset{\mathsf{src}}{\vdash} S_2 : \mathsf{wf}$$
$$\Gamma \overset{\mathsf{src}}{\vdash} \overline{D}_1 : \mathsf{wf} \wedge \Gamma \overset{\mathsf{src}}{\vdash} \overline{D}_1 <: \overline{D}_2 \implies \Gamma \overset{\mathsf{src}}{\vdash} \overline{D}_2 : \mathsf{wf}$$
$$\Gamma \overset{\mathsf{src}}{\vdash} S : \mathsf{wf} \implies \Gamma \overset{\mathsf{src}}{\vdash} S <: S$$
$$\Gamma \overset{\mathsf{src}}{\vdash}_Y \overline{D} : \mathsf{wf} \implies \Gamma \overset{\mathsf{src}}{\vdash} \overline{D} <: \overline{D}$$
$$\Gamma \vdash S \approx S' \iff \left(\Gamma \overset{\mathsf{src}}{\vdash} S <: S' \wedge \Gamma \overset{\mathsf{src}}{\vdash} S' <: S\right)$$

S-Sub-Lookup-Left

$$\frac{\Gamma \vdash P.A \triangleright S' \qquad \Gamma \overset{\text{src}}{\vdash} S' <: S}{\Gamma \overset{\text{src}}{\vdash} P.A <: S}$$

S-Sub-Lookup-Right

$$\frac{\Gamma \vdash P.A \triangleright S' \qquad \Gamma \overset{\text{src}}{\vdash} S <: S'}{\Gamma \overset{\text{src}}{\vdash} S <: P.A}$$

S-Sub-Fct

$$\frac{\Gamma \overset{\text{src}}{\vdash} S'_a <: S_a \qquad \Gamma, \mathsf{module}\, X : S_a \overset{\text{src}}{\vdash} S_r <: S'_r \qquad X \notin \Gamma}{\Gamma \overset{\text{src}}{\vdash} (X : S_a) \to S_r <: (X : S'_a) \to S'_r}$$

S-Sub-Sig

$$\frac{\Gamma \overset{\text{src}}{\vdash}_Y \overline{D} : \mathsf{wf} \qquad \overline{D}_0 \subseteq \overline{D} \qquad \overline{\Gamma, \overline{D} \overset{\text{src}}{\vdash}_Y D_0 <: D'} \qquad \Gamma \overset{\text{src}}{\vdash}_Y \overline{D'} : \mathsf{wf}}{\Gamma \overset{\text{src}}{\vdash} \mathtt{sig}_Y\ \overline{D}\ \mathtt{end} <: \mathtt{sig}_Y\ \overline{D'}\ \mathtt{end}}$$

S-Sub-Sig-Eq

$$\frac{\Gamma \overset{\text{src}}{\vdash}_Y \overline{D} : \mathsf{wf} \qquad \overline{\Gamma, \overline{D} \overset{\text{src}}{\vdash}_Y D \prec: D'} \qquad \Gamma \overset{\text{src}}{\vdash}_Y \overline{D'} : \mathsf{wf}}{\Gamma \overset{\text{src}}{\vdash} \mathtt{sig}_Y\ \overline{D}\ \mathtt{end} \prec: \mathtt{sig}_Y\ \overline{D'}\ \mathtt{end}}$$

Figure 33: $\Gamma \overset{\text{src}}{\vdash} S <: S'$ – Signature subtyping rules
Canonical: Figure 51, Elaboration: Figure 62

D_0 is not wellformed in general, as it might use declarations of D, but we have: $\overline{\Gamma, \overline{D} \overset{\text{src}}{\vdash}_Y D_0 : \mathsf{wf}}$ (every declaration is wellformed, not the set of them). Wellformedness of $\overline{\mathcal{D}}'$ ensures that $\overline{\mathcal{D}}_0$ does not duplicate fields.

S-Sub-Mod

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

S-Sub-ModType

$$\frac{\Gamma \overset{\text{src}}{\vdash} S <: S' \qquad \Gamma \overset{\text{src}}{\vdash} S' <: S}{\Gamma \overset{\text{src}}{\vdash}_Y (\mathsf{module\ type}\, A = S) <: (\mathsf{module\ type}\, A = S')}$$

S-Sub-Val

$$\frac{\Gamma \overset{\text{src}}{\vdash} T_1 <: T_2}{\Gamma \overset{\text{src}}{\vdash}_Y (\mathsf{val}\, x : T_1) <: (\mathsf{val}\, x : T_2)}$$

S-Sub-TypeAbs

$$\Gamma \overset{\text{src}}{\vdash}_Y (\mathsf{type}\, t) <: (\mathsf{type}\, t)$$

S-Sub-Type

$$\frac{\Gamma \overset{\text{src}}{\vdash} T_1 <: T_2}{\Gamma \overset{\text{src}}{\vdash}_Y (\mathsf{type}\, t = T_1) <: (\mathsf{type}\, t = T_2)}$$

S-Sub-TypeToAbs

$$\frac{\Gamma \overset{\text{src}}{\vdash} T : \mathsf{wf}}{\Gamma \overset{\text{src}}{\vdash}_Y (\mathsf{type}\, t = T) <: (\mathsf{type}\, t)}$$

Figure 34: $\Gamma \overset{\text{src}}{\vdash}_Y D <: id'$ – Declaration subtyping rules
Canonical: Figure 52, Elaboration: Figure 63

S-Typ-Var
$$\frac{\Gamma \vdash P \triangleright S \qquad \Gamma : \mathsf{wf}}{\Gamma \overset{\mathsf{src}}{\vdash} P : S}$$

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

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

S-Typ-Sig
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash} P : S' \qquad \Gamma \overset{\mathsf{src}}{\vdash} S' <: S}{\Gamma \overset{\mathsf{src}}{\vdash} (P : S) : S}$$

S-Typ-App
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash} P_f : (X : S_a) \to S_r \qquad \Gamma \overset{\mathsf{src}}{\vdash} P : S \qquad \Gamma \overset{\mathsf{src}}{\vdash} S <: S_a}{\Gamma \overset{\mathsf{src}}{\vdash} P_f(P) : S_r[X \mapsto P]}$$

S-Typ-Fct
$$\frac{X \notin \Gamma \qquad \Gamma; \mathsf{module}\, X : S_a \overset{\mathsf{src}}{\vdash} M : S_r}{\Gamma \overset{\mathsf{src}}{\vdash} ((X : S_a) \to M) : (X : S_a) \to S_r}$$

S-Typ-Struct
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash}_Y B : \overline{D} \qquad Y \notin \Gamma}{\Gamma \overset{\mathsf{src}}{\vdash} \mathtt{struct}_Y\ B\ \mathtt{end} : \mathtt{sig}_Y\ \overline{D}\ \mathtt{end}}$$

S-Typ-Proj
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash} M : \mathtt{sig}_Y\ \overline{D}_1\,, \mathsf{module}\, X : S\,, \overline{D}_2\ \mathtt{end} \qquad \Gamma\,, \overline{D}_1 \overset{\mathsf{src}}{\vdash} S \prec: S' \qquad \Gamma \overset{\mathsf{src}}{\vdash} S' : \mathsf{wf}}{\Gamma \overset{\mathsf{src}}{\vdash} M.X : S}$$

Figure 35: $\Gamma \overset{\mathsf{src}}{\vdash} M : S$ – Module typing rules
Canonical: Figure 53, Elaboration: Figure 64

S-Typ-Let
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash} E : T \qquad Y.x \notin \Gamma}{\Gamma \overset{\mathsf{src}}{\vdash}_Y (\mathsf{let}\, x = E) : (\mathsf{val}\, x : T)}$$

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

S-Typ-Type
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash} T : \mathsf{wf} \qquad Y.t \notin \Gamma}{\Gamma \overset{\mathsf{src}}{\vdash}_Y (\mathsf{type}\, t = T) : (\mathsf{type}\, t = T)}$$

S-Typ-ModType
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash} S : \mathsf{wf} \qquad Y.A \notin \Gamma}{\Gamma \overset{\mathsf{src}}{\vdash}_Y (\mathsf{module\ type}\, A = S) : (\mathsf{module\ type}\, A = S)}$$

S-Typ-Empty
$$\frac{\Gamma : \mathsf{wf}}{\Gamma \overset{\mathsf{src}}{\vdash}_Y \varepsilon : \varepsilon}$$

S-Typ-Seq
$$\frac{\Gamma \overset{\mathsf{src}}{\vdash}_Y B_1 : \overline{D}_1 \qquad \Gamma\,, Y.\overline{D}_1 \overset{\mathsf{src}}{\vdash}_Y B_2 : \overline{D}_2}{\Gamma \overset{\mathsf{src}}{\vdash}_Y B_1; B_2 : \overline{D}_1 \mathbin{+\!\!+} \overline{D}_2}$$

Figure 36: $\Gamma \overset{\mathsf{src}}{\vdash}_Y B : \overline{D}$ – Bindings typing rules
Canonical: Figure 54, Elaboration: Figure 65

A.6 Typing

Invariants

$$\Gamma \overset{\mathsf{src}}{\vdash} M : S \implies \Gamma \overset{\mathsf{src}}{\vdash} S : \mathsf{wf}$$

B Canonical system

B.1 Lookup

$$\textsc{C-Lkp-FctArg}\quad \frac{(\mathsf{module}\, X : \mathcal{R}) \in \Gamma}{\Gamma \vdash X \triangleright \mathcal{R}} \qquad \textsc{C-Lkp-Mod}\quad \frac{Y.(\mathsf{module}\, X : \mathcal{R}) \in \Gamma}{\Gamma \vdash Y.X \triangleright \mathcal{R}} \qquad \textsc{C-Lkp-ModType}\quad \frac{Y.(\mathsf{module\ type}\, A = \mathcal{S}) \in \Gamma}{\Gamma \vdash Y.A \triangleright \mathcal{S}}$$

$$\textsc{C-Lkp-Proj-Mod}\quad \frac{\Gamma \vdash P \triangleright \mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} \qquad (\mathsf{module}\, X : \mathcal{R}) \in D}{\Gamma \vdash P.X \triangleright \mathcal{R}[Y \mapsto P]} \qquad \textsc{C-Lkp-Proj-Sig}\quad \frac{\Gamma \vdash P \triangleright \mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} \qquad (\mathsf{module\ type}\, A = \mathcal{S}) \in D}{\Gamma \vdash P.A \triangleright \mathcal{S}[Y \mapsto P]}$$

Figure 37: $\Gamma \vdash P \triangleright \mathcal{R}$ and $\Gamma \vdash P.A \triangleright \mathcal{S}$– Lookup rules
Source: Figure 23, Elaboration: Figure 55

Invariants

$$\neg\, (\Gamma \vdash P \triangleright \mathcal{R} \implies \Gamma : \mathsf{wf})$$

B.2 Well-formedness

$$\textsc{C-Wf-Empty}\quad \varepsilon : \mathsf{wf} \qquad \textsc{C-Wf-AbsTypes}\quad \frac{\Gamma : \mathsf{wf} \qquad \overline{\alpha} \notin \Gamma}{\Gamma, \overline{\alpha} : \mathsf{wf}} \qquad \textsc{C-Wf-FctArg}\quad \frac{\Gamma \overset{\mathsf{can}}{\vdash} \mathcal{R} : \mathsf{wf} \qquad X \notin \Gamma}{\Gamma, (\mathsf{module}\, X : \mathcal{R}) : \mathsf{wf}} \qquad \textsc{C-Wf-Dec}\quad \frac{\Gamma \overset{\mathsf{can}}{\vdash}_Y \mathcal{D} : \mathsf{wf}}{\Gamma, Y.\mathcal{D} : \mathsf{wf}}$$

Figure 38: $\Gamma : \mathsf{wf}$ – Wellformedness of environments
Source: Figure 24, Elaboration: Figure 56

$$\textsc{C-Wf-Abs}\quad \frac{\Gamma, \overline{\alpha} \overset{\mathsf{can}}{\vdash} \mathcal{R} : \mathsf{wf}}{\Gamma \overset{\mathsf{can}}{\vdash} \exists \overline{\alpha}.\mathcal{R} : \mathsf{wf}} \qquad \textsc{C-Wf-Sig}\quad \frac{\Gamma \overset{\mathsf{can}}{\vdash}_Y \overline{\mathcal{D}} : \mathsf{wf}}{\Gamma \overset{\mathsf{can}}{\vdash} \mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} : \mathsf{wf}} \qquad \textsc{C-Wf-Funct}\quad \frac{\Gamma, \overline{\alpha} \overset{\mathsf{can}}{\vdash} \mathcal{R} : \mathsf{wf} \qquad \Gamma, \overline{\alpha}, \mathsf{module}\, X : \mathcal{R} \overset{\mathsf{can}}{\vdash} \mathcal{S} : \mathsf{wf} \qquad X \notin \Gamma}{\Gamma \overset{\mathsf{can}}{\vdash} \forall \overline{\alpha}.\, (X : \mathcal{R}) \to \mathcal{S} : \mathsf{wf}}$$

Figure 39: $\Gamma \overset{\mathsf{can}}{\vdash} \mathcal{S} : \mathsf{wf}$ – Wellformedness of NF-signatures
Source: Figure 25, Elaboration: Figure 57

C-Wf-Val
$$\frac{\Gamma \overset{\text{can}}{\vdash} \tau : \mathsf{wf} \qquad Y.x \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \mathsf{val}\, x : (\approx \tau) : \mathsf{wf}}$$

C-Wf-Type
$$\frac{\Gamma \overset{\text{can}}{\vdash} \tau : \mathsf{wf} \qquad Y.t \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \mathsf{type}\, t \approx \tau : \mathsf{wf}}$$

C-Wf-Mod
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathcal{R} : \mathsf{wf} \qquad Y.X \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y (\mathsf{module}\, X : \mathcal{R}) : \mathsf{wf}}$$

C-Wf-ModType
$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf} \qquad Y.A \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y (\mathsf{module\ type}\, A = \mathcal{S}) : \mathsf{wf}}$$

C-Wf-Empty
$$\frac{\Gamma : \mathsf{wf}}{\Gamma \overset{\text{can}}{\vdash}_Y \varepsilon : \mathsf{wf}}$$

C-Wf-Seq
$$\frac{\Gamma \overset{\text{can}}{\vdash}_Y \mathcal{D}_1 : \mathsf{wf} \qquad \Gamma, Y.\mathcal{D}_1 \overset{\text{can}}{\vdash}_Y \overline{\mathcal{D}} : \mathsf{wf}}{\Gamma \overset{\text{can}}{\vdash}_Y (\mathcal{D}_1, \overline{\mathcal{D}}) : \mathsf{wf}}$$

Figure 40: $\Gamma \overset{\text{can}}{\vdash}_Y \mathcal{D} : \mathsf{wf}$ – Wellformedness of declarations
Source: Figure 26, Elaboration: Figure 58

B.3 Canonification

C-Cf-LocalType
$$\frac{\Gamma : \mathsf{wf} \qquad \mathsf{type}\, Y.t \approx \tau \in \Gamma}{\Gamma \overset{\text{can}}{\vdash} Y.t \hookrightarrow \tau}$$

C-Cf-Type
$$\frac{\Gamma : \mathsf{wf} \qquad \Gamma \vdash P \triangleright \mathtt{sig}_Y\, \overline{\mathcal{D}}\ \mathtt{end} \qquad (\mathsf{type}\, t \approx \tau) \in \overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\vdash} P.t \hookrightarrow \tau}$$

C-Cf-Cgr
$$\frac{\Gamma : \mathsf{wf} \qquad \forall i \in [\![0, n]\!],\ \Gamma \overset{\text{can}}{\vdash} T_i \hookrightarrow \tau_i}{\Gamma \overset{\text{can}}{\vdash} Op(T_0, \ldots, T_n) \hookrightarrow Op(\tau_0, \ldots, \tau_n)}$$

Figure 41: $\Gamma \overset{\text{can}}{\vdash} T \hookrightarrow \tau$ – CF-conversion of types
Elaboration: Figure 59

C-Cf-Val
$$\frac{\Gamma \overset{\text{can}}{\vdash} T \hookrightarrow \tau \qquad Y.x \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \mathsf{val}\, x : T \hookrightarrow \mathsf{val}\, x : (\approx \tau)}$$

C-Cf-Type
$$\frac{\Gamma \overset{\text{can}}{\vdash} T \hookrightarrow \tau \qquad Y.t \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \mathsf{type}\, t = T \hookrightarrow \mathsf{type}\, t \approx \tau}$$

C-Cf-TypeAbs
$$\frac{\Gamma : \mathsf{wf} \qquad Y.t \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \mathsf{type}\, t \hookrightarrow \exists \alpha.\, \mathsf{type}\, t \approx \alpha}$$

C-Cf-Mod
$$\frac{\Gamma \overset{\text{can}}{\vdash} S \hookrightarrow \exists \overline{\alpha}.\mathcal{R} \qquad Y.X \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y (\mathsf{module}\, X : S) \hookrightarrow \exists \overline{\alpha}.\, (\mathsf{module}\, X : \mathcal{R})}$$

C-Cf-ModType
$$\frac{\Gamma \overset{\text{can}}{\vdash} S \hookrightarrow \mathcal{S} \qquad Y.A \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y (\mathsf{module\ type}\, A = S) \hookrightarrow (\mathsf{module\ type}\, A = \mathcal{S})}$$

C-Cf-Empty
$$\frac{\Gamma : \mathsf{wf}}{\Gamma \overset{\text{can}}{\vdash}_Y \varepsilon \hookrightarrow \varepsilon}$$

C-Cf-Seq
$$\frac{\Gamma \overset{\text{can}}{\vdash}_Y D_1 \hookrightarrow \exists \overline{\alpha}_1.\mathcal{D}_1 \qquad \Gamma, \overline{\alpha}_1, Y.\mathcal{D}_1 \overset{\text{can}}{\vdash}_Y \overline{D} \hookrightarrow \exists \overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\vdash}_Y (D_1, \overline{D}) \hookrightarrow \exists \overline{\alpha}_1 \overline{\alpha}.\, (\mathcal{D}_1, \overline{\mathcal{D}})}$$

Figure 42: $\Gamma \overset{\text{can}}{\vdash}_Y D \hookrightarrow \exists \overline{\alpha}.\mathcal{D}$ – NF-conversion of declarations
Elaboration: Figure 61

C-Cf-Sig
$$\frac{\Gamma \overset{\text{can}}{\vdash}_Y \overline{D} \hookrightarrow \exists \overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\vdash} \mathtt{sig}_Y\, \overline{D}\ \mathsf{end} \hookrightarrow \exists \overline{\alpha}.\mathtt{sig}_Y\, \overline{\mathcal{D}}\ \mathsf{end}}$$

C-Cf-ModTypeVar
$$\frac{\Gamma : \mathsf{wf} \qquad \Gamma \vdash P.A \triangleright \mathcal{S}}{\Gamma \overset{\text{can}}{\vdash} P.A \hookrightarrow \mathcal{S}}$$

C-Cf-Funct
$$\frac{\Gamma \overset{\text{can}}{\vdash} S_a \hookrightarrow \exists \overline{\alpha}.\mathcal{R}_a \qquad \Gamma, \overline{\alpha}, \mathsf{module}\, X : \mathcal{R}_a \overset{\text{can}}{\vdash} S \hookrightarrow \mathcal{S} \qquad X \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash} (X : S_a) \to S \hookrightarrow \forall \overline{\alpha}.\, (X : \mathcal{R}) \to \mathcal{S}}$$

Figure 43: $\Gamma \overset{\text{can}}{\vdash} S \hookrightarrow \mathcal{S}$ – CF-conversion of signatures
Elaboration: Figure 60

B.4 Anchorable well-formedness

C-AWf-Empty
$$\varepsilon : \mathsf{wf}_a$$

C-AWf-FctArg
$$\frac{\Gamma \overset{\text{can}}{\vdash} \exists \overline{\alpha}.\mathcal{R} : \mathsf{wf}_a \qquad X \notin \Gamma \qquad \overline{\alpha} \notin \Gamma}{\Gamma, \overline{\alpha}, (\mathsf{module}\, X : \mathcal{R}) : \mathsf{wf}_a}$$

C-AWf-Dec
$$\frac{\Gamma \overset{\text{can}}{\vdash}_Y \exists \overline{\alpha}.\mathcal{D} : \mathsf{wf}_a \qquad \overline{\alpha} \notin \Gamma}{\Gamma, \overline{\alpha}, Y.\mathcal{D} : \mathsf{wf}_a}$$

Figure 44: $\Gamma : \mathsf{wf}_a$ – Wellformedness of environments
Source: Figure 24, Elaboration: Figure 56

$$\textsc{C-AWf-Sig}\quad \frac{\Gamma \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}.\overline{\mathcal{D}} : \mathsf{wf}_a}{\Gamma \overset{\text{can}}{\vdash} \exists\overline{\alpha}.\mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} : \mathsf{wf}_a}$$

$$\textsc{C-AWf-Funct}\quad \frac{\Gamma \overset{\text{can}}{\vdash} \exists\overline{\alpha}.\mathcal{R} : \mathsf{wf}_a \qquad \Gamma\,,\overline{\alpha}\,,\mathsf{module}\,X : \mathcal{R} \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a \qquad X \notin \Gamma \qquad \overline{\alpha} \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash} \forall\overline{\alpha}.\,(X : \mathcal{R}) \to \mathcal{S} : \mathsf{wf}_a}$$

Figure 45: $\Gamma \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a$ – Wellformedness of NF-signatures
Source: Figure 25, Elaboration: Figure 57

$$\Gamma \overset{\text{can}}{\vdash} \exists\overline{\alpha}.\mathcal{R} : \mathsf{wf}_a \implies \Gamma\,,\overline{\alpha} \overset{\text{can}}{\vdash} \mathcal{R} : \mathsf{wf}$$

$$\textsc{C-AWf-Val}\quad \frac{\Gamma \overset{\text{can}}{\vdash} \tau : \mathsf{wf}_a \qquad Y.x \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \mathsf{val}\,x : (\approx \tau) : \mathsf{wf}_a}$$

$$\textsc{C-AWf-Type}\quad \frac{\Gamma \overset{\text{can}}{\vdash} \tau : \mathsf{wf}_a \qquad Y.t \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \mathsf{type}\,t \approx \tau : \mathsf{wf}_a}$$

$$\textsc{C-AWf-TypeAbs}\quad \frac{\Gamma : \mathsf{wf}_a \qquad Y.t \notin \Gamma \qquad \alpha \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \exists\alpha.\,(\mathsf{type}\,t \approx \alpha) : \mathsf{wf}_a}$$

$$\textsc{C-AWf-Mod}\quad \frac{\Gamma \overset{\text{can}}{\vdash} \exists\overline{\alpha}.\mathcal{R} : \mathsf{wf}_a \qquad Y.X \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}.\,(\mathsf{module}\,X : \mathcal{R}) : \mathsf{wf}_a}$$

$$\textsc{C-AWf-ModType}\quad \frac{\Gamma \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a \qquad Y.A \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y (\mathsf{module\ type}\,A = \mathcal{S}) : \mathsf{wf}_a}$$

$$\textsc{C-AWf-Empty}\quad \frac{\Gamma : \mathsf{wf}_a}{\Gamma \overset{\text{can}}{\vdash}_Y \varepsilon : \mathsf{wf}_a}$$

$$\textsc{C-AWf-Seq}\quad \frac{\Gamma \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}_1.\mathcal{D}_1 : \mathsf{wf}_a \qquad \Gamma\,,\overline{\alpha}_1\,,Y.\mathcal{D}_1 \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}.\overline{\mathcal{D}} : \mathsf{wf}_a \qquad \overline{\alpha} \cap \overline{\alpha}_1 = \emptyset}{\Gamma \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}_1\overline{\alpha}.\,(\mathcal{D}_1\,,\overline{\mathcal{D}}) : \mathsf{wf}_a}$$

Figure 46: $\Gamma \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}.\mathcal{D} : \mathsf{wf}_a$ – Wellformedness of declarations
Source: Figure 26, Elaboration: Figure 58

The bottom-up approach of anchored-wellformedness is visible here. When we have $\Gamma \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}.\overline{\mathcal{D}} : \mathsf{wf}_a$, it means that the declarations $\overline{\mathcal{D}}$ are defining the existential types $\overline{\alpha}$ with explicit abstract type definition $(\mathsf{type}\,t \approx \alpha)$, and are wellformed.

$$\Gamma \overset{\text{can}}{\vdash}_Y \exists\overline{\alpha}.\overline{\mathcal{D}} : \mathsf{wf}_a \implies \Gamma\,,\overline{\alpha} \overset{\text{can}}{\vdash}_Y \overline{\mathcal{D}} : \mathsf{wf}$$

B.5 Anchored well-formedness

C-AWEf-Empty

$$\varepsilon : \mathsf{wf}_a \hookleftarrow (\varepsilon, \emptyset)$$

C-AWEf-FctArgAbs

$$\frac{\Gamma \overset{\mathsf{can}}{\vdash} \exists\overline{\alpha}.\mathtt{sig}_X\ \overline{\mathcal{D}}\ \mathtt{end} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma)\,, \left(\mathtt{sig}_X\ \overline{D}\ \mathtt{end}, \sigma'\right) \qquad X \notin \Gamma \qquad \overline{\alpha} \notin \Gamma \qquad \sigma'' = \beta \mapsto \begin{cases} X.\sigma'(\beta) & \text{if } \beta \in \overline{\alpha} \\ \sigma(\beta) & \text{otherwise} \end{cases}}{\Gamma\,, \overline{\alpha}\,, \left(\mathsf{module}\, X : \mathtt{sig}_X\ \overline{\mathcal{D}}\ \mathtt{end}\right) : \mathsf{wf}_a \hookleftarrow \left(\left(\Gamma_{\mathsf{src}}\,, \mathsf{module}\, X : \mathtt{sig}_X\ \overline{D}\ \mathtt{end}\right), \sigma''\right)}$$

C-AWEf-FctArgFct

$$\frac{\Gamma \overset{\mathsf{can}}{\vdash} \mathcal{R} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma)\,, (S, \emptyset) \qquad X \notin \Gamma \qquad \overline{\alpha} \notin \Gamma}{\Gamma\,, \overline{\alpha}\,, (\mathsf{module}\, X : \mathcal{R}) : \mathsf{wf}_a \hookleftarrow ((\Gamma_{\mathsf{src}}\,, \mathsf{module}\, X : S)\,, \emptyset)}$$

C-AWEf-Dec

$$\frac{\Gamma \overset{\mathsf{can}}{\vdash}_Y \exists\overline{\alpha}.\overline{\mathcal{D}} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma)\,, \left(\overline{D}, \sigma'\right) \qquad \overline{\alpha} \notin \Gamma \qquad \sigma'' = \beta \mapsto \begin{cases} Y.\sigma'(\beta) & \text{if } \beta \in \overline{\alpha} \\ \sigma(\beta) & \text{otherwise} \end{cases}}{\Gamma\,, \overline{\alpha}\,, \overline{Y.\mathcal{D}} : \mathsf{wf}_a \hookleftarrow \left(\left(\Gamma_{\mathsf{src}}\,, \overline{Y.D}\right), \sigma''\right)}$$

Figure 47: $\Gamma : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma)$ – Anchored wellformedness of environments

$$\sigma : \alpha \mapsto T$$

$$\forall \alpha \in \Gamma, \Gamma_{\mathsf{src}} \overset{\mathsf{src}}{\vdash} \sigma(\alpha) : \mathsf{wf}$$

$$\Gamma : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma) \implies \Gamma_{\mathsf{src}} : \mathsf{wf}$$

$$\Gamma : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma) \implies \Gamma_{\mathsf{src}} \hookrightarrow \Gamma$$

C-AWEf-Sig

$$\frac{\Gamma \overset{\mathsf{can}}{\vdash}_Y \exists\overline{\alpha}.\overline{\mathcal{D}} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma)\,, \left(\overline{D}, \sigma'\right)}{\Gamma \overset{\mathsf{can}}{\vdash} \exists\overline{\alpha}.\mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma)\,, \left(\mathtt{sig}_Y\ \overline{D}\ \mathtt{end}, \sigma'\right)}$$

C-AWEf-Funct

$$\frac{\Gamma \overset{\mathsf{can}}{\vdash} \exists\overline{\alpha}.\mathcal{R} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma)\,, (S_a, \sigma_a) \qquad \Gamma\,, \overline{\alpha}\,, \mathsf{module}\, X : \mathcal{R} \overset{\mathsf{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a \hookleftarrow ((\Gamma_{\mathsf{src}}\,, \mathsf{module}\, X : S_a)\,, \sigma')\,, (S_r, \sigma_r) \qquad X \notin \Gamma \qquad \overline{\alpha} \notin \Gamma}{\Gamma \overset{\mathsf{can}}{\vdash} \forall\overline{\alpha}.\, (X : \mathcal{R}) \to \mathcal{S} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma)\,, ((X : S_a) \to S_r, \emptyset)}$$

Figure 48: $\Gamma \overset{\mathsf{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma)\,, (S, \sigma_s)$ – Anchored wellformedness of canonical signatures

$$\Gamma \overset{\mathsf{can}}{\vdash} \mathcal{R} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma)\,, (S, \emptyset)$$

$$\Gamma \overset{\mathsf{can}}{\vdash} \exists\overline{\alpha}.\mathcal{R} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma)\,, (S, \sigma') \implies \sigma' : \overline{\alpha} \to T$$

C-AWEf-Val

$$\frac{\Gamma \overset{\text{can}}{\vdash} \tau : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), T \qquad Y.x \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \mathsf{val}\, x : (\approx \tau) : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), (\mathsf{val}\, x : T, \emptyset)}$$

C-AWEf-Type

$$\frac{\Gamma \overset{\text{can}}{\vdash} \tau : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), T \qquad Y.t \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \mathsf{type}\, t \approx \tau : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), (\mathsf{type}\, t = T, \emptyset)}$$

C-AWEf-TypeAbs

$$\frac{\Gamma : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma) \qquad Y.t \notin \Gamma \qquad \alpha \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y \exists \alpha. (\mathsf{type}\, t \approx \alpha) : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), (\mathsf{type}\, t, \alpha \mapsto Y.t)}$$

C-AWEf-ModAbs

$$\frac{\Gamma \overset{\text{can}}{\vdash} \exists \overline{\alpha}. \left(\mathtt{sig}_X\ \overline{\mathcal{D}}\ \mathtt{end}\right) : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), \left(\mathtt{sig}_X\ \overline{D}\ \mathtt{end}, \sigma'\right) \qquad Y.X \notin \Gamma \qquad \sigma'' = \alpha \mapsto Y.\sigma'(\alpha)}{\Gamma \overset{\text{can}}{\vdash}_Y \exists \overline{\alpha}. \left(\mathsf{module}\, X : \mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end}\right) : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), \left(\left(\mathsf{module}\, X : \mathtt{sig}_Y\ \overline{D}\ \mathtt{end}\right), \sigma''\right)}$$

C-AWEf-ModFct

$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathcal{R} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), (S, \emptyset) \qquad Y.X \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y (\mathsf{module}\, X : \mathcal{R}) : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), ((\mathsf{module}\, X : S), \emptyset)}$$

C-AWEf-ModType

$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), (S, \sigma') \qquad Y.A \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y (\mathsf{module\ type}\, A = \mathcal{S}) : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), ((\mathsf{module\ type}\, X = S), \emptyset)}$$

C-AWEf-Empty

$$\frac{\Gamma : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma)}{\Gamma \overset{\text{can}}{\vdash}_Y \varepsilon : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), (\varepsilon, \emptyset)}$$

C-AWEf-Seq

$$\frac{\begin{array}{c}\Gamma \overset{\text{can}}{\vdash}_Y \exists \overline{\alpha}_1.\mathcal{D}_1 : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), (D_1, \sigma_1) \\ \Gamma, \overline{\alpha}_1, Y.\mathcal{D}_1 \overset{\text{can}}{\vdash}_Y \exists \overline{\alpha}.\overline{\mathcal{D}} : \mathsf{wf}_a \hookleftarrow ((\Gamma_{\mathsf{src}}, Y.D_1), \sigma'), \left(\overline{D}, \sigma_2\right) \qquad \overline{\alpha} \cap \overline{\alpha}_1 = \emptyset\end{array}}{\Gamma \overset{\text{can}}{\vdash}_Y \exists \overline{\alpha}_1 \overline{\alpha}. \left(\mathcal{D}_1, \overline{\mathcal{D}}\right) : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), \left(\left(D_1, \overline{D}\right), \sigma_1 \uplus \sigma_2\right)}$$

Figure 49: $\Gamma \overset{\text{can}}{\vdash} \exists \overline{\alpha}.\overline{\mathcal{D}} : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), \left(\overline{D}, \sigma'\right)$ – Anchored wellformedness of declarations

C-AWEf-Existential

$$\frac{\Gamma : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma)}{\Gamma \overset{\text{can}}{\vdash} \alpha : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), \sigma(\alpha)}$$

C-AWEf-Congruence

$$\frac{\forall i \in [\![1, n]\!], \Gamma \overset{\text{can}}{\vdash} \tau_i : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), T_i}{\Gamma \overset{\text{can}}{\vdash} Op\,(\tau_1, \ldots, \tau_n) : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), Op\,(T_1, \ldots, T_n)}$$

Figure 50: $\Gamma \overset{\text{can}}{\vdash} \tau : \mathsf{wf}_a \hookleftarrow (\Gamma_{\mathsf{src}}, \sigma), T$ – Anchored wellformedness of types

C-Sub-Abs

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

C-Sub-Funct

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

C-Sub-Sig-Erase

$$\frac{\overline{\mathcal{D}_0} \subseteq \overline{\mathcal{D}} \qquad \Gamma \overset{\text{can}}{\vdash}_Y \overline{\mathcal{D}} : \mathsf{wf} \qquad \overline{\Gamma, \overline{\mathcal{D}} \overset{\text{can}}{\vdash} \mathcal{D}_0 <: \mathcal{D}'} \qquad \Gamma \overset{\text{can}}{\vdash}_Y \overline{\mathcal{D}'} : \mathsf{wf}}{\Gamma \overset{\text{can}}{\vdash} \mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} <: \mathtt{sig}_Y\ \overline{\mathcal{D}'}\ \mathtt{end}}$$

C-Sub-Sig-Abs

$$\frac{\Gamma \overset{\text{can}}{\vdash}_Y \overline{\mathcal{D}} : \mathsf{wf} \qquad \overline{\Gamma, \overline{\mathcal{D}} \overset{\text{can}}{\vdash} \mathcal{D} \prec: \mathcal{D}'} \qquad \Gamma \overset{\text{can}}{\vdash}_Y \overline{\mathcal{D}'} : \mathsf{wf}}{\Gamma \overset{\text{can}}{\vdash} \mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} \prec: \mathtt{sig}_Y\ \overline{\mathcal{D}'}\ \mathtt{end}}$$

Figure 51: $\Gamma \vdash \mathcal{S} <: \mathcal{S}'$ – Canonical signatures subtyping rules
Source: Figure 33, Elaboration: Figure 62

C-Sub-Mod

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

C-Sub-ModType

$$\frac{\Gamma \overset{\text{can}}{\vdash} \mathcal{S} <: \mathcal{S}' \qquad \Gamma \overset{\text{can}}{\vdash} \mathcal{S}' <: \mathcal{S}}{\Gamma \overset{\text{can}}{\vdash} (\mathsf{module\ type}\, A = \mathcal{S}) <: (\mathsf{module\ type}\, A = \mathcal{S}')}$$

C-Sub-Val

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

C-Sub-Type

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

Figure 52: $\Gamma \overset{\text{can}}{\vdash} \mathcal{D} <: \mathcal{D}'$ – Canonical declarations subtyping rules
Source: Figure 34, Elaboration: Figure 63

B.6 Subtyping

Invariants

$$\Gamma \overset{\text{can}}{\vdash} S_1 <: S_2 \implies \Gamma \overset{\text{can}}{\vdash} S_1 : \mathsf{wf} \wedge \Gamma \overset{\text{can}}{\vdash} S_2 : \mathsf{wf}$$

$$\Gamma \overset{\text{can}}{\vdash} S : \mathsf{wf} \implies \Gamma \overset{\text{can}}{\vdash} S <: S$$

$$\Gamma \overset{\text{can}}{\vdash}_Y \overline{D} : \mathsf{wf} \implies \Gamma \overset{\text{can}}{\vdash} \overline{D} <: \overline{D}$$

Theorems

$$\Gamma \overset{\text{can}}{\vdash} S_1 \prec: S_2 \implies \Gamma \overset{\text{can}}{\vdash} S_1 <: S_2$$

B.7 Typing

C-Typ-Var
$$\frac{\Gamma : \mathsf{wf} \qquad \Gamma \vdash P \triangleright \mathcal{R}}{\Gamma \overset{\text{can}}{\vdash} P : \mathcal{R}}$$

C-Typ-Sig
$$\frac{\Gamma \overset{\text{can}}{\vdash} S \hookrightarrow \mathcal{S} \qquad \Gamma \overset{\text{can}}{\vdash} P : \mathcal{R} \qquad \Gamma \overset{\text{can}}{\vdash} \mathcal{R} <: \mathcal{S}}{\Gamma \overset{\text{can}}{\vdash} (P : S) : \mathcal{S}}$$

C-Typ-App
$$\frac{\Gamma \overset{\text{can}}{\vdash} P_f : \forall \overline{\alpha}. (X : \mathcal{R}_a) \to \mathcal{S}_r \qquad \Gamma \overset{\text{can}}{\vdash} P : \mathcal{R} \qquad \Gamma \overset{\text{can}}{\vdash} \mathcal{R} <: \mathcal{R}_a[\overline{\alpha} \mapsto \overline{\tau}]}{\Gamma \overset{\text{can}}{\vdash} P_f(P) : \mathcal{S}_r[\overline{\alpha} \mapsto \overline{\tau}]}$$

C-Typ-Fct
$$\frac{X \notin \Gamma \qquad \Gamma \overset{\text{can}}{\vdash} S_a \hookrightarrow \exists \overline{\alpha}.\mathcal{R} \qquad \Gamma, \overline{\alpha}, \mathsf{module}\, X : \mathcal{R} \overset{\text{can}}{\vdash} M : \mathcal{S}}{\Gamma \overset{\text{can}}{\vdash} ((X : S_a) \to M) : \forall \overline{\alpha}. (X : \mathcal{R}) \to \mathcal{S}}$$

C-Typ-Struct
$$\frac{Y \notin \Gamma \qquad \Gamma \overset{\text{can}}{\vdash}_Y B : \exists \overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \overset{\text{can}}{\vdash} \mathtt{struct}_Y\ B\ \mathtt{end} : \exists \overline{\alpha}.\mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end}}$$

C-Typ-Proj
$$\frac{\Gamma \overset{\text{can}}{\vdash} M : \exists \overline{\alpha}.\mathtt{sig}_Y\ \overline{\mathcal{D}}_1, \mathsf{module}\, X : \mathcal{R}, \overline{\mathcal{D}}_2\ \mathtt{end} \qquad \Gamma \overset{\text{can}}{\vdash} \exists \overline{\alpha}.\mathcal{R} : \mathsf{wf}}{\Gamma \overset{\text{can}}{\vdash} M.X : \exists \overline{\alpha}.\mathcal{R}}$$

Figure 53: $\Gamma \overset{\text{can}}{\vdash} M : \mathcal{S}$ – Module typing rules
Source: Figure 35, Elaboration: Figure 64

C-Typ-Let
$$\frac{\Gamma \overset{\text{can}}{\vdash} E : T \qquad Y.x \notin \Gamma \qquad \Gamma \overset{\text{can}}{\vdash} T \hookrightarrow \tau}{\Gamma \overset{\text{can}}{\vdash}_Y (\mathsf{let}\, x = E) : (\mathsf{val}\, x : (\approx \tau))}$$

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

C-Typ-Type
$$\frac{\Gamma \overset{\text{can}}{\vdash} T \hookrightarrow \tau \qquad Y.t \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y (\mathsf{type}\, t = T) : (\mathsf{type}\, t \approx \tau)}$$

C-Typ-ModType
$$\frac{\Gamma \overset{\text{can}}{\vdash} S \hookrightarrow \mathcal{S} \qquad Y.A \notin \Gamma}{\Gamma \overset{\text{can}}{\vdash}_Y (\mathsf{module\ type}\, A = S) : (\mathsf{module\ type}\, A = \mathcal{S})}$$

C-Typ-Empty
$$\frac{\Gamma : \mathsf{wf}}{\Gamma \overset{\text{can}}{\vdash}_Y \varepsilon : \varepsilon}$$

C-Typ-Seq
$$\frac{\Gamma \overset{\text{can}}{\vdash}_Y B_1 : \exists \overline{\alpha}_1.\overline{\mathcal{D}_1} \qquad \Gamma, \overline{\alpha}_1, \overline{Y.\mathcal{D}_1} \overset{\text{can}}{\vdash}_Y B_2 : \exists \overline{\alpha}_2.\overline{\mathcal{D}_2}}{\Gamma \overset{\text{can}}{\vdash}_Y B_1; B_2 : \exists \overline{\alpha}_1 \overline{\alpha}_2.\overline{\mathcal{D}_1} +\!\!+ \overline{\mathcal{D}_2}}$$

Figure 54: $\Gamma \overset{\text{can}}{\vdash}_Y B : \exists \overline{\alpha}.\mathcal{D}$ – Bindings typing rules
Source: Figure 36, Elaboration: Figure 65

Invariants

$$\Gamma \overset{\text{can}}{\vdash} M : \mathcal{S} \implies \Gamma \overset{\text{can}}{\vdash} \mathcal{S} : \mathsf{wf} \wedge \Gamma : \mathsf{wf}$$

C Elaboration

C.1 Lookup rules

$$\textsc{E-Lkp-FctArg}\quad \frac{(\mathsf{module}\, X : \mathcal{R} \rightsquigarrow \Sigma) \in \Gamma}{\Gamma \vdash X \triangleright \mathcal{R} \rightsquigarrow \Sigma}$$

$$\textsc{E-Lkp-Mod}\quad \frac{(Y.X : \mathsf{module} : \mathcal{R} \rightsquigarrow \Sigma) \in \Gamma}{\Gamma \vdash Y.X \triangleright \mathcal{R} \rightsquigarrow \Sigma}$$

$$\textsc{E-Lkp-ModType}\quad \frac{(Y.A : \mathsf{module\ type} = \mathcal{S} \rightsquigarrow \Pi) \in \Gamma}{\Gamma \vdash Y.A \triangleright \mathcal{S} \rightsquigarrow \Pi}$$

$$\textsc{C-Lkp-Proj-Mod}\quad \frac{\Gamma \vdash P \triangleright \mathtt{sig}\ \overline{\mathcal{D}}\ \mathtt{end} \rightsquigarrow \Sigma \qquad (\mathsf{module}\, X : \mathcal{R}) \in D \qquad \Sigma = \{\ldots, l_X : \Sigma', \ldots\}}{\Gamma \vdash P.X \triangleright \mathcal{R} \rightsquigarrow \Sigma'}$$

$$\textsc{C-Lkp-Proj-Sig}\quad \frac{\Gamma \vdash P \triangleright \mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} \rightsquigarrow \Sigma \qquad (\mathsf{module\ type}\, A = \mathcal{S}) \in D \qquad \Sigma = \{\ldots, l_A : [= \Pi], \ldots\}}{\Gamma \vdash P.A \triangleright \mathcal{S} \rightsquigarrow \Pi}$$

Figure 55: $\Delta \vdash P \triangleright S \rightsquigarrow \Sigma$ – Path lookup rules
Source: Figure 23, Canonical: Figure 37

C.2 Wellformedness

$$\textsc{E-Wf-Empty}\quad \varepsilon : \mathsf{wf}$$

$$\textsc{E-Wf-AbsTypes}\quad \frac{\Delta : \mathsf{wf} \qquad \alpha \notin \Delta}{(\Delta, \overline{\alpha}) : \mathsf{wf}}$$

$$\textsc{E-Wf-FctArg}\quad \frac{\Delta \overset{\mathsf{elab}}{\vdash} \mathcal{R} : \mathsf{wf} \rightsquigarrow \Sigma \qquad X \notin \Delta}{\Delta, (\mathsf{module}\, X : \mathcal{R} \rightsquigarrow \Sigma) : \mathsf{wf}}$$

$$\textsc{E-Wf-Dec}\quad \frac{\Delta \overset{\mathsf{elab}}{\vdash}_Y \mathcal{D} : \mathsf{wf} \rightsquigarrow \Sigma \qquad Z \notin D}{(\Delta, Y.Z : \mathcal{D} \rightsquigarrow \Sigma) : \mathsf{wf}}$$

Figure 56: $\Delta : \mathsf{wf}$ – Wellformedness of environments
Source: Figure 24, Canonical: Figure 38

E-Wf-Abs

$$\frac{\Delta\,,\overline{\alpha} \overset{\text{elab}}{\vdash} \mathcal{R} : \mathsf{wf} \rightsquigarrow \Sigma}{\Delta \overset{\text{elab}}{\vdash} \exists\overline{\alpha}.\mathcal{R} : \mathsf{wf} \rightsquigarrow \exists\overline{\alpha}.\Sigma}$$

E-Wf-Sig

$$\frac{\Delta \overset{\text{elab}}{\vdash}_Y \overline{\mathcal{D}} : \mathsf{wf} \rightsquigarrow \Sigma}{\Delta \overset{\text{elab}}{\vdash} \mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} : \mathsf{wf} \rightsquigarrow \Sigma}$$

E-Wf-Funct

$$\frac{\Delta \overset{\text{elab}}{\vdash} \exists\overline{\alpha}.\mathcal{R} : \mathsf{wf} \rightsquigarrow \exists\overline{\alpha}.\Sigma \qquad \Delta\,,\overline{\alpha}\,,(\mathsf{module}\,X : \mathcal{R} \rightsquigarrow \Sigma) \overset{\text{elab}}{\vdash} \mathcal{S} : \mathsf{wf} \rightsquigarrow \Pi \qquad X \notin \Delta}{\Delta \overset{\text{elab}}{\vdash} \forall\overline{\alpha}.\,(X : \mathcal{R}) \to \mathcal{S} : \mathsf{wf} \rightsquigarrow \forall\overline{\alpha}.\Sigma \to \Pi}$$

Figure 57: $\Delta \overset{\text{elab}}{\vdash} \mathcal{S} : \mathsf{wf} \rightsquigarrow \Pi$ – Wellformedness of NF-signatures
Source: Figure 25, Canonical: Figure 39

E-Wf-Val

$$\frac{\Delta \overset{\text{elab}}{\vdash} \tau : \mathsf{wf} \qquad Y.x \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y \mathsf{val}\,x : (\approx \tau) : \mathsf{wf} \rightsquigarrow \{l_x : [\tau]\}}$$

E-Wf-Type

$$\frac{\Delta \overset{\text{elab}}{\vdash} \tau : \mathsf{wf} \qquad Y.t \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y \mathsf{type}\,t \approx \tau : \mathsf{wf} \rightsquigarrow \{l_t : [= \tau : \star]\}}$$

E-Wf-Mod

$$\frac{\Delta \overset{\text{elab}}{\vdash} \mathcal{R} : \mathsf{wf} \rightsquigarrow \Sigma \qquad Y.X \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y (\mathsf{module}\,X : \mathcal{R}) : \mathsf{wf} \rightsquigarrow \{l_X : \Sigma\}}$$

E-Wf-ModType

$$\frac{\Delta \overset{\text{elab}}{\vdash} \mathcal{S} : \mathsf{wf} \rightsquigarrow \Pi \qquad Y.A \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y (\mathsf{module\ type}\,A = \mathcal{S}) : \mathsf{wf} \rightsquigarrow \{l_A : [= \Pi]\}}$$

E-Wf-Empty

$$\frac{\Delta : \mathsf{wf}}{\Delta \overset{\text{elab}}{\vdash}_Y \varepsilon : \mathsf{wf} \rightsquigarrow \{\}}$$

E-Wf-Seq

$$\frac{\Delta \overset{\text{elab}}{\vdash}_Y \mathcal{D}_1 : \mathsf{wf} \rightsquigarrow \{l_{Z_1} : \Sigma_1\} \qquad \Delta\,,(Y.Z_1 : \mathcal{D}_1 \rightsquigarrow \Sigma_1) \overset{\text{elab}}{\vdash}_Y \overline{\mathcal{D}} : \mathsf{wf} \rightsquigarrow \{\overline{l_Z : \Sigma}\}}{\Delta \overset{\text{elab}}{\vdash}_Y (\mathcal{D}_1\,,\overline{\mathcal{D}}) : \mathsf{wf} \rightsquigarrow \{l_{Z_1} : \Sigma_1, \overline{l_Z : \Sigma}\}}$$

Figure 58: $\Delta \overset{\text{elab}}{\vdash}_Y \mathcal{D} : \mathsf{wf} \rightsquigarrow \Sigma$ – Wellformedness of declarations
Source: Figure 26, Canonical: Figure 40

C.3 Canonification

E-Cf-LocalType

$$\frac{\Delta : \mathsf{wf} \qquad (\mathsf{type}\, Y.t \approx \tau \rightsquigarrow [= \tau : \star]) \in \Delta}{\Delta \overset{\mathsf{elab}}{\vdash} Y.t \hookrightarrow \tau}$$

E-Cf-Type

$$\frac{\Delta : \mathsf{wf} \qquad \Delta \vdash P \triangleright \mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} \rightsquigarrow \Sigma \qquad (\mathsf{type}\, t \approx \tau) \in \overline{\mathcal{D}} \qquad \Sigma.l_t = [= \tau : \star]}{\Delta \overset{\mathsf{elab}}{\vdash} P.t \hookrightarrow \tau}$$

E-Cf-Cgr

$$\frac{\Delta : \mathsf{wf} \qquad \forall i \in [\![0, n]\!],\ \Delta \overset{\mathsf{elab}}{\vdash} T_i \hookrightarrow \tau_i}{\Delta \overset{\mathsf{elab}}{\vdash} Op(T_0, \ldots, T_n) \hookrightarrow Op(\tau_0, \ldots, \tau_n)}$$

Figure 59: $\Delta \overset{\mathsf{elab}}{\vdash} T \hookrightarrow \tau$ – CF-conversion of types
Canonical: Figure 41

E-Cf-Sig

$$\frac{\Delta \overset{\mathsf{elab}}{\vdash}_Y \overline{D} \hookrightarrow \exists\overline{\alpha}.\overline{\mathcal{D}} \rightsquigarrow \exists\overline{\alpha}.\Sigma}{\Delta \overset{\mathsf{elab}}{\vdash} \mathtt{sig}_Y\ \overline{D}\ \mathtt{end} \hookrightarrow \exists\overline{\alpha}.\mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} \rightsquigarrow \exists\overline{\alpha}.\Sigma}$$

E-Cf-Funct

$$\frac{\Delta \overset{\mathsf{elab}}{\vdash} S_1 \hookrightarrow \exists\overline{\alpha}.\mathcal{R} \rightsquigarrow \exists\overline{\alpha}.\Sigma \qquad \Delta\,, \overline{\alpha}\,, (\mathsf{module}\, X : \mathcal{R} \rightsquigarrow \Sigma) \overset{\mathsf{elab}}{\vdash} S \hookrightarrow \mathcal{S} \rightsquigarrow \Pi \qquad X \notin \Delta}{\Delta \overset{\mathsf{elab}}{\vdash} ((X : S_1) \to S_2) \hookrightarrow (\forall\overline{\alpha}.\,(X : \mathcal{R}) \to \mathcal{S}) \rightsquigarrow (\forall\overline{\alpha}.\Sigma \to \Pi)}$$

E-Cf-ModType

$$\frac{\Delta : \mathsf{wf} \qquad \Delta \vdash P.A \triangleright \mathcal{S} \rightsquigarrow \Pi}{\Delta \overset{\mathsf{elab}}{\vdash} P.A \hookrightarrow \mathcal{S} \rightsquigarrow \Pi}$$

$$\Delta \overset{\mathsf{elab}}{\vdash} S \hookrightarrow \mathcal{S} \rightsquigarrow \Pi \implies \Delta \overset{\mathsf{elab}}{\vdash}_Y \mathcal{S} : \mathsf{wf} \rightsquigarrow \Pi$$

Figure 60: $\Delta \overset{\mathsf{elab}}{\vdash} S \hookrightarrow \mathcal{S} \rightsquigarrow \Pi$ – NF-conversion of signatures
Canonical: Figure 43

E-Cf-Val

$$\frac{\Delta \overset{\text{elab}}{\vdash} T \hookrightarrow \tau \qquad Y.x \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y \mathsf{val}\, x : T \hookrightarrow \mathsf{val}\, x : (\approx \tau) \rightsquigarrow \{l_x : [\tau]\}}$$

E-Cf-Type

$$\frac{\Delta \overset{\text{elab}}{\vdash} T \hookrightarrow \tau \qquad Y.t \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y \mathsf{type}\, t = T \hookrightarrow \mathsf{type}\, t \approx \tau \rightsquigarrow \{l_t : [= \tau : \star]\}}$$

E-Cf-TypeAbs

$$\frac{\Delta : \mathsf{wf} \qquad Y.t \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y \mathsf{type}\, t \hookrightarrow \exists \alpha.\, \mathsf{type}\, t (\approx \alpha) \rightsquigarrow \exists \alpha.\, \{l_t : [= \alpha : \star]\}}$$

E-Cf-Mod

$$\frac{\Delta \overset{\text{elab}}{\vdash} S \hookrightarrow \exists \overline{\alpha}.\mathcal{R} \rightsquigarrow \exists \overline{\alpha}.\Sigma \qquad Y.X \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y (\mathsf{module}\, X : S) \hookrightarrow \exists \overline{\alpha}.\, (\mathsf{module}\, X : \mathcal{R}) \rightsquigarrow \exists \overline{\alpha}.\, \{l_X : \Sigma\}}$$

E-Cf-ModType

$$\frac{\Delta \overset{\text{elab}}{\vdash} S \hookrightarrow \mathcal{S} \rightsquigarrow \Pi \qquad Y.A \notin \Delta}{\Delta \overset{\text{elab}}{\vdash}_Y (\mathsf{module\ type}\, A = S) \hookrightarrow (\mathsf{module\ type}\, A = \mathcal{S}) \rightsquigarrow \{l_A : [= \Pi]\}}$$

E-Cf-Empty

$$\frac{\Delta : \mathsf{wf}}{\Delta \overset{\text{elab}}{\vdash}_Y \varepsilon \hookrightarrow \varepsilon \rightsquigarrow \{\}}$$

E-Cf-Seq

$$\frac{\Delta \overset{\text{elab}}{\vdash}_Y D_1 \hookrightarrow \exists \overline{\alpha}_1.\mathcal{D}_1 \rightsquigarrow \exists \overline{\alpha}_1.\, \{l_{Z_1} : \Sigma_1\} \qquad \Delta\,, \overline{\alpha}_1\,, (Y.Z_1 : \mathcal{D}_1 \rightsquigarrow \Sigma_1) \overset{\text{elab}}{\vdash}_Y \overline{D} \hookrightarrow \exists \overline{\alpha}.\overline{\mathcal{D}} \rightsquigarrow \exists \overline{\alpha}.\, \{\overline{l_Z : \Sigma}\}}{\Delta \overset{\text{elab}}{\vdash}_Y \left(D_1\,, \overline{D}\right) \hookrightarrow \exists \overline{\alpha}_1 \overline{\alpha}.\, \left(\mathcal{D}_1\,, \overline{\mathcal{D}}\right) \rightsquigarrow \exists \overline{\alpha}_1 \overline{\alpha}.\, \left\{l_{Z_1} : \Sigma_1, \overline{l_Z : \Sigma}\right\}}$$

$$\Delta \overset{\text{elab}}{\vdash}_Y \overline{D} \hookrightarrow \exists \overline{\alpha}.\overline{\mathcal{D}} \rightsquigarrow \exists \overline{\alpha}.\Sigma \implies \Delta\,, \overline{\alpha} \overset{\text{elab}}{\vdash}_Y \overline{\mathcal{D}} : \mathsf{wf} \rightsquigarrow \Sigma$$

Figure 61: $\Delta \overset{\text{elab}}{\vdash}_Y D \hookrightarrow \exists \overline{\alpha}.\mathcal{D} \rightsquigarrow \exists \overline{\alpha}.\Sigma$ – NF-conversion of declarations
Canonical: Figure 42

C.4 Subtyping

E-Sub-Abs

$$\frac{\Delta\,,\overline{\alpha} \overset{\mathsf{elab}}{\vdash} \mathcal{R} <: \mathcal{R}'[\overline{\alpha}' \mapsto \overline{\tau}] \rightsquigarrow \Sigma <: \Sigma'[\overline{\alpha}' \mapsto \overline{\tau}] \uparrow f}{\Delta \overset{\mathsf{elab}}{\vdash} \exists\overline{\alpha}.\mathcal{R} <: \exists\overline{\alpha}'.\mathcal{R}' \rightsquigarrow \exists\overline{\alpha}.\Sigma <: \exists\overline{\alpha}'.\Sigma' \uparrow \lambda\,(x : \exists\overline{\alpha}.\Sigma)\,.\ \mathsf{unpack}\,\langle\overline{\alpha}, y\rangle = x\ \mathsf{in}\ \mathsf{pack}\,\langle\overline{\tau}, f\ y\rangle}$$

E-Sub-Funct

$$\frac{\begin{array}{c}\Delta\,,\overline{\alpha}' \overset{\mathsf{elab}}{\vdash} \mathcal{R}' <: \mathcal{R}[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow \Sigma' <: \Sigma[\overline{\alpha} \mapsto \overline{\tau}] \uparrow f_1 \\ \Delta\,,\overline{\alpha}'\,,(\mathsf{module}\,X : \mathcal{R}' \rightsquigarrow \Sigma') \overset{\mathsf{elab}}{\vdash} \mathcal{S}[\overline{\alpha} \mapsto \overline{\tau}] <: \mathcal{S}' \rightsquigarrow \Pi[\overline{\alpha} \mapsto \overline{\tau}] <: \Pi' \uparrow f_2 \qquad X \notin \Delta\end{array}}{\Delta \overset{\mathsf{elab}}{\vdash} \forall\overline{\alpha}.\,(X : \mathcal{R}) \to \mathcal{S} <: \forall\overline{\alpha}'.\,(X : \mathcal{R}') \to \mathcal{S}' \rightsquigarrow (\forall\overline{\alpha}.\Sigma \to \Pi) <: (\forall\overline{\alpha}'.\Sigma' \to \Pi') \uparrow \begin{array}{l}\lambda\,(f : (\forall\overline{\alpha}.\Sigma \to \Pi))\,. \\ \lambda\overline{\alpha}'. \\ \lambda\,(x : \Sigma')\,.\,f_2\,(f\ \overline{\tau}\ (f_1\ x))\end{array}}$$

E-Sub-Sig

$$\frac{\begin{array}{c}\overline{\mathcal{D}_0} \subseteq \overline{\mathcal{D}} \\ \Delta \overset{\mathsf{elab}}{\vdash} \overline{\mathcal{D}} : \mathsf{wf} \rightsquigarrow \left\{\overline{l_{Z'} : \Sigma_0}, \overline{l_Z : \Sigma}\right\} \qquad \Delta \overset{\mathsf{elab}}{\vdash} \overline{\mathcal{D}'} : \mathsf{wf} \rightsquigarrow \left\{\overline{l_{Z'} : \Sigma'}\right\} \qquad \dfrac{}{\Delta \overset{\mathsf{elab}}{\vdash} \mathcal{D}_0 <: \mathcal{D}' \rightsquigarrow \Sigma_0 <: \Sigma' \uparrow f}\end{array}}{\Delta \overset{\mathsf{elab}}{\vdash} \mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} <: \mathtt{sig}_Y\ \overline{\mathcal{D}'}\ \mathtt{end} \rightsquigarrow \left\{\overline{l_{Z'} : \Sigma_0}, \overline{l_Z : \Sigma}\right\} <: \left\{\overline{l_{Z'} : \Sigma'}\right\} \uparrow \lambda\left(x : \left\{\overline{l_{Z'} : \Sigma_0}, \overline{l_Z : \Sigma}\right\}\right). \left\{\overline{l_{Z'} = f(x.l_{Z'})}\right\}}$$

Figure 62: $\Delta \overset{\mathsf{elab}}{\vdash} \mathcal{S} <: \mathcal{S}' \rightsquigarrow a <: b \uparrow f$ – NF-Signature subtyping rules
Source: Figure 33, Canonical: Figure 51

$$\left(\Delta \overset{\mathsf{elab}}{\vdash} \mathcal{S} <: \mathcal{S}' \rightsquigarrow \Pi <: \Pi' \uparrow f\right) \implies f : \Pi \to \Pi'$$

E-Sub-Val

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

E-Sub-Type

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

E-Sub-Mod

$$\frac{\Delta \overset{\text{elab}}{\vdash} \mathcal{R} <: \mathcal{R}' \rightsquigarrow \Sigma <: \Sigma' \uparrow f}{\Delta \overset{\text{elab}}{\vdash} (\mathsf{module}\, X : \mathcal{R}) <: (\mathsf{module}\, X : \mathcal{R}') \rightsquigarrow \Sigma <: \Sigma' \uparrow f}$$

E-Sub-ModType

$$\frac{\Delta \overset{\text{elab}}{\vdash} \mathcal{S} <: \mathcal{S}' \rightsquigarrow \Pi <: \Pi' \uparrow f \qquad \Delta \overset{\text{elab}}{\vdash} \mathcal{S}' <: \mathcal{S} \rightsquigarrow \Pi' <: \Pi \uparrow f'}{\Delta \overset{\text{elab}}{\vdash} (\mathsf{module\ type}\, A = \mathcal{S}) <: (\mathsf{module\ type}\, A = \mathcal{S}') \rightsquigarrow \Pi <: \Pi' \uparrow \lambda\,(x : [= \Pi])\,.\,[\Pi']}$$

Figure 63: $\Delta \overset{\text{elab}}{\vdash} \mathcal{D} <: \mathcal{D}' \rightsquigarrow \Sigma <: \Sigma' \uparrow f$ – Declaration subtyping rules
Source: Figure 34, Canonical: Figure 52

$$\Delta \overset{\text{elab}}{\vdash} \overline{\mathcal{D}} <: \overline{\mathcal{D}'} \rightsquigarrow \overline{\Sigma} <: \overline{\Sigma'} \uparrow \overline{f} \implies \begin{cases} \overline{f : \Sigma \to \Sigma'} \\ \Delta \overset{\text{elab}}{\vdash} \overline{\mathcal{D}} : \mathsf{wf} \rightsquigarrow \{\overline{l_Z : \Sigma}\} \\ \Delta \overset{\text{elab}}{\vdash} \overline{\mathcal{D}'} : \mathsf{wf} \rightsquigarrow \{\overline{l_Z : \Sigma'}\} \end{cases}$$

$$\Delta \overset{\text{elab}}{\vdash}_Y \overline{\mathcal{D}} : \mathsf{wf} \rightsquigarrow \{\overline{l_Z : \Sigma}\} \implies \Delta \overset{\text{elab}}{\vdash} \overline{\mathcal{D}} <: \overline{\mathcal{D}} \rightsquigarrow \{\overline{l_Z : \Sigma}\} <: \{\overline{l_Z : \Sigma}\} \uparrow \overline{\lambda\,(x : \Sigma)\,.\,x}$$

C.5 Typing

E-Typ-Var

$$\frac{\Delta : \mathsf{wf} \qquad \Delta \vdash P \triangleright \mathcal{R} \rightsquigarrow \Sigma}{\Delta \overset{\mathsf{elab}}{\vdash} P : \mathcal{R} \rightsquigarrow \lfloor P \rfloor : \Sigma}$$

E-Typ-Sig

$$\frac{\Delta \overset{\mathsf{elab}}{\vdash} S \hookrightarrow \exists\overline{\alpha}.\mathcal{R}_1 \rightsquigarrow \exists\overline{\alpha}.\Sigma_1 \qquad \Delta \overset{\mathsf{elab}}{\vdash} P : \mathcal{R}_2 \rightsquigarrow \lfloor P \rfloor : \Sigma_2 \qquad \Delta \overset{\mathsf{elab}}{\vdash} \mathcal{R}_2 <: \mathcal{R}_1[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow \Sigma_2 <: \Sigma_1[\overline{\alpha} \mapsto \overline{\tau}] \uparrow f}{\Delta \overset{\mathsf{elab}}{\vdash} (P : S) : \mathcal{S} \rightsquigarrow \mathsf{pack}\,\langle \overline{\tau}, f\lfloor P \rfloor \rangle : \exists\overline{\alpha}.\Sigma_1}$$

E-Typ-App

$$\frac{\Delta \overset{\mathsf{elab}}{\vdash} P_f : \forall\overline{\alpha}.\,(X : \mathcal{R}_a) \to \mathcal{S}_r \rightsquigarrow \lfloor P_f \rfloor : \forall\overline{\alpha}.\Sigma_a \to \Pi_r \qquad \Delta \overset{\mathsf{elab}}{\vdash} P : \mathcal{R} \rightsquigarrow \lfloor P \rfloor : \Sigma \qquad \Delta \overset{\mathsf{elab}}{\vdash} \mathcal{R} <: \mathcal{R}_a[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow \Sigma <: \Sigma_a[\overline{\alpha} \mapsto \overline{\tau}] \uparrow f}{\Delta \overset{\mathsf{elab}}{\vdash} P_f(P) : \mathcal{S}_r[\overline{\alpha} \mapsto \overline{\tau}][X \mapsto P] \rightsquigarrow \lfloor P_f \rfloor\, \overline{\tau}\,(f\lfloor P \rfloor) : \Pi_r[\overline{\alpha} \mapsto \overline{\tau}]}$$

E-Typ-Fct

$$\frac{X \notin \Delta \qquad \Delta \overset{\mathsf{elab}}{\vdash} S_a \hookrightarrow \exists\overline{\alpha}.\mathcal{R} \rightsquigarrow \exists\overline{\alpha}.\Sigma \qquad \Delta\,,\overline{\alpha}\,,\mathsf{module}\,X : \mathcal{R} \rightsquigarrow \Sigma \overset{\mathsf{elab}}{\vdash} M : \mathcal{S} \rightsquigarrow e : \Pi}{\Delta \overset{\mathsf{elab}}{\vdash} ((X : S_a) \to M) : \forall\overline{\alpha}.\,(X : \mathcal{R}) \to \mathcal{S} \rightsquigarrow \lambda\overline{\alpha}.\lambda\,(x : \Sigma)\,.\,e : \forall\overline{\alpha}.\Sigma \to \Pi}$$

E-Typ-Struct

$$\frac{Y \notin \Delta \qquad \Delta \overset{\mathsf{elab}}{\vdash} B : \exists\overline{\alpha}.\overline{\mathcal{D}} \rightsquigarrow e : \exists\overline{\alpha}.\Sigma}{\Delta \overset{\mathsf{elab}}{\vdash} \mathtt{struct}_Y\ B\ \mathtt{end} : \exists\overline{\alpha}.\mathtt{sig}_Y\ \overline{\mathcal{D}}\ \mathtt{end} \rightsquigarrow e : \exists\overline{\alpha}.\Sigma}$$

E-Typ-Proj

$$\frac{\Delta \overset{\mathsf{elab}}{\vdash} M : \exists\overline{\alpha}.\mathtt{sig}_Y\ \overline{\mathcal{D}}_1\,,\mathsf{module}\,X : \mathcal{R}\,,\overline{\mathcal{D}}_2\ \mathtt{end} \rightsquigarrow e : \exists\overline{\alpha}.\,\left\{\overline{l_{Z_1} : \Sigma_1}, l_X : \Sigma, \overline{l_{Z_2} : \Sigma_2}\right\}}{\Delta \overset{\mathsf{elab}}{\vdash} M.X : \exists\overline{\alpha}.\mathcal{R} \rightsquigarrow \mathsf{unpack}\,\langle \overline{\alpha}, y \rangle = e\ \mathsf{in}\ \mathsf{pack}\,\langle \overline{\alpha}, y.l_X \rangle : \exists\overline{\alpha}.\Sigma}$$

Figure 64: $\Delta \overset{\mathsf{elab}}{\vdash} M : \mathcal{S} \rightsquigarrow e : \Pi$ – Module typing rules
Source: Figure 35, Canonical: Figure 53

E-Typ-Let

$$\frac{\Delta \overset{\text{elab}}{\vdash} E : T \rightsquigarrow e : \tau \qquad Y.x \notin \Delta}{\Delta \overset{\text{elab}}{\vdash} (\mathsf{let}\, x = E) : (\mathsf{val}\, x : (\approx \tau)) \rightsquigarrow \{l_x = e\} : \{l_x : [\tau]\}}$$

E-Typ-Type

$$\frac{\Delta \overset{\text{elab}}{\vdash} T \hookrightarrow \tau \qquad Y.t \notin \Delta}{\Delta \overset{\text{elab}}{\vdash} \mathsf{type}\, t = T : \mathsf{type}\, t \approx \tau \rightsquigarrow \{l_t = [\tau : \star]\} : \{l_t : [= \tau : \star]\}}$$

E-Typ-Mod

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

E-Typ-ModType

$$\frac{\Delta \overset{\text{elab}}{\vdash} S \hookrightarrow \mathcal{S} \rightsquigarrow \Pi \qquad Y.A \notin \Delta}{\Delta \overset{\text{elab}}{\vdash} (\mathsf{module\ type}\, A = S) : (\mathsf{module\ type}\, A = \mathcal{S}) \rightsquigarrow \{l_A = [\Pi]\} : \{l_A : [= \Pi]\}}$$

E-Typ-Empty

$$\frac{\Delta : \mathsf{wf}}{\Delta \overset{\text{elab}}{\vdash} \varepsilon : \varepsilon \rightsquigarrow \{\} : \{\}}$$

E-Typ-Seq

$$\frac{\Delta \overset{\text{elab}}{\vdash} B_1 : \exists\overline{\alpha}_1.\overline{\mathcal{D}_1} \rightsquigarrow e_1 : \exists\overline{\alpha}_1.\, \{\overline{l_{Z_1} : \Sigma_1}\} \qquad \Delta\,, \overline{\alpha}_1\,, \overline{Y.\mathcal{D}_1 : l_{Z_1} \rightsquigarrow \Sigma_1} \overset{\text{elab}}{\vdash} B_2 : \exists\overline{\alpha}_2.\overline{\mathcal{D}_2} \rightsquigarrow e_2 : \exists\overline{\alpha}_2.\, \{\overline{l_{Z_2} : \Sigma_2}\}}{\begin{array}{l} \Delta \overset{\text{elab}}{\vdash} B_1; B_2 : \exists\overline{\alpha}_1\overline{\alpha}_2.\overline{\mathcal{D}_1} +\!\!+ \overline{\mathcal{D}_2} \rightsquigarrow \rightsquigarrow : \mathsf{unpack}\, \langle\overline{\alpha}_1, y_1\rangle = e_1\ \mathsf{in} \qquad : \exists\overline{\alpha}_1\overline{\alpha}_2.\, \{\overline{l_{Z_1} : \Sigma_1}, \overline{l_{Z_2} : \Sigma_2}\} \\ \qquad \mathsf{unpack}\, \langle\overline{\alpha}_2, y_2\rangle = \\ \qquad\quad \mathsf{let}\, \lfloor \overline{Y.Z_1} \rfloor = y_1.\overline{l_{Z_1}}\ \mathsf{in}\, e_2\ \ \mathsf{in} \\ \qquad \mathsf{pack}\, \langle\overline{\alpha}_1\overline{\alpha}_2, \{\overline{l_{Z_1} : y_1.\overline{l_{Z_1}}}, \overline{l_{Z_2} : y_2}\}\rangle \end{array}}$$

Figure 65: $\Delta \overset{\text{elab}}{\vdash} B : \exists\overline{\alpha}.\mathcal{D} \rightsquigarrow e : \exists\overline{\alpha}.\Sigma$ – Bindings typing rules
Source: Figure 36, Canonical: Figure 54