Université Paris Cité

École Doctorale 386 — Sciences Mathématiques de Paris Centre
Inria

Retrofitting ML modules

(working version as of September 23, 2024)

Résumé

WIP

Résumé en français

WIP

Abstract

WIP

Acknowledgments

WIP

Résumé en français (long)

Contents

List of Figures

Introduction

WIP

- Programming languages intro
- Modularity for complex systems

Practical problems : namespaces, separate compilation
Theoretical problems : reusability, structure, abstraction

Overview

Chapter 1

Features and challenges of a modern module system

The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise

- Edsger Dijkstra

In this chapter, we introduce the design space of ML-modules informally. We present the key features, the strength and weaknesses of modularity *à la ML*. While the type theory is developed more thoroughly in Chapter 3 and Chapter 4, we gathered the key design insights and present them here. This chapter serves both as an introduction, a summary and a plan for what the future of ML modules might look like.

Features Throughout the chapter, we present both language *mechanisms*, language *properties* and language *constructs* – we refer to all three as *features*. In each section, we summarize the features using the following marks: M for mechanisms, P for properties and C for constructs. As we do both an overview of existing features and a *whish-list* of what would constitute a fully-accomplished ML-module system, we use symbols to indicate the status of each feature:

- ✓ represents largely supported and well understood features
- ● represents partially supported features, for which the theoretical background might be missing or the implementation incomplete
- ✚ represents new features that are not supported in current implementations, but for which we advocate

Unless stated otherwise, the status symbol applies to OCaml, but comparisons with SML, Moscow ML, or experimental languages such as 1ML are made. We sum up those features in Section 1.8.

Code inserts We display code inserts with the following convention:

```
This is an OCaml (or OCaml-like) code insert
```

```
This is the result of typechecking when it succeeds
```

```
This is an error that occurred during typechecking
```

Contributions In this chapter, we provide a complete, unified and updated overview of all features of ML-module systems. We present language-design solutions as well as some new features, especially regarding tools for *abstraction*.

Overview After presenting the basic building blocks in Section 1.1, we focus on the key mechanism of type abstraction.

In Section 1.2, we explore the interaction between abstraction and functors. It leads to the distinction between *generativity* and *applicativity*, and the *module equivalence* problem. We discuss an extension of the syntactic criterion for applicativity via the new feature of *transparent signatures*.

In Section 1.3, we focus on the interaction between abstraction and projection, which poses the (infamous) *signature avoidance problem*. We present the main approaches to this problem, as well as the solution followed by the current OCaml implementation. We discuss the need for a *type-preserving* solution, and for a mechanism to *delay* avoidance.

In Section 1.4, we move from type-level abstraction to module-level abstraction. We present the current state of the feature in OCaml, as well as a proposed restriction of *simple abstract signatures*.

In Section 1.5, we discuss the *composition* between modules, i.e., the way modules can be combined together. After presenting hierarchical and flat composition, we briefly discuss recursive modules and open recursion, both from a typing and semantic point of view.

In Section 1.6, we present other advanced features, starting with those enabling a better interaction between the module language and the core language, namely *first-class modules* and *modular explicits/implicits*. Then, we move on to the features for signature manipulation and miscellaneous ones.

In Section 1.7, we briefly compare ML-style modules with other approaches of modularity from other languages, mainly object-oriented programming and type-classes.

Finally, in Section 1.8, we sum up all the features presented in this chapter to give an overview and a vision for a fully-accomplished, modern, and powerful module system.

```
(* Cartesian implementation *)
module Complex = struct
  type t = float * float
  let one = (1., 0.)
  let zero = (0., 0.)
  let add (a,b) (c,d) =
    (a +. c, b +. d)
  let mul (a,b) (c,d) =
    (a*.c -. b*.d, b*.c +. a*.d)
  let imaginary_part (a,_) = a
  let complex_part (_,b) = b
end
```

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

```
module CRing = (Complex : Ring)
```

```
module Polynomials =
 functor (R : Ring) →
  struct
    type t = R.t list
    let zero = []
    let one = [R.one]
    let rec add p1 p2 = match p1,p2 with
      | x1::p1, x2::p2 →
         (R.add x1 x2)::(add p1 p2)
      | [], _ → p2
      | _ , [] → p1
    let rec mul p1 p2 = match p1 with
      | [] → []
      | x1::p1 →
         add (List.map (R.mul x1) p2)
           (mul p1 (R.zero::p2))
  end
```

```
module CX = Polynomials(Complex)
module PolynomialsXY (R : Ring) =
    Polynomials(Polynomials(R))
```

Figure 1: Basic modularity example (in OCaml).

1.1 Basic modularity

We start with the basic constructs of ML modules. An introductory example is given in Figure 1 and developed in this section.

Module layer The key design choice of ML is that modularity is provided by a *separate* language layer, the *module layer*, that uses different keywords an mechanisms from the rest of language, called the *core-language*. As a consequence, both languages can be designed with different trade-offs, adapted for programming *in the small* and *in the large*. For instance, in ML, the core-language uses a first-order type language, but is equipped with a powerful inference engine that makes is mostly implicitly typed (and yet predictable). By contrast, the module-layer has powerful type language, but requires explicit type annotations.

Structures The basic building block of ML modules is the *structure*, obtained with the keywords **`struct/end`**. Definitions inside a structure are called *bindings* Inside a structure, the user can bind values and define new types, each associated with a *name*. Bindings have an *open scope*: the name they define is accessible for the rest of the enclosing structure. This is especially visible for value bindings: they are made with the **`let`** keyword, but there are no corresponding **`in`**: the scope is open.

Type definitions Type definitions are made with the **`type`** keyword. OCaml offers a rich variety of type definitions: algebraic data-types (ADT), generalized algebraic data-types (GADTs)Garrigue and Rémy [2012], objectsRémy and Vouillon [1997], polymorphic variants, extensible types, parametric types, type annotations (variance, unboxing, etc.), private types, type constraints, etc. New proposals are often made to extend the type definition mechanism: the latest being (at the time of writing) modal memory management Lorenzen et al. [2024]. The type definition mechanism sits a bit in-between the core and module layers: it is a construct of the module language but is technically supported by the core. The module language is more or less agnostic of the structure of type declarations.

Manifest types OCaml also supports *manifest types* Leroy [1994]: as done in the module `Complex`, one can give a name to any *already-existing* type expression (here, `float*float`). In practice, this feature is essential, as it allows to use short, local names for type expressions that might be big and clutter the program. It also acts as a light form of encapsulation. Type manifests are handled by the module layer.

Signatures The type of a module—obtained by inference or written by the user—is called a *signature*. The signature of structures are called *structural signatures* and uses the keyword **`sig/end`**. Definitions inside a structural signature are called *declarations* [1]. Signatures can be named via a *module-type* binding, as illustrated by `Ring`. This is especially useful, as signature can grow very big and clutter the output if no short names are used. While the nomenclature in the literature can vary, in this thesis, we refer to the type of modules as signatures, while module-types only refer to the binding/declaration.

Type Abstraction The killer feature of ML modules, that is also the crux from a formalization point of view is *type abstraction*. Type declarations can be left abstract, as the one at line 14. This allows the definition of *new types* that are only compatible with themselves. Abstract types provide a powerful and flexible mechanism for *encapsulation*, that we will be the heart of the discussion throughout the rest of this chapter.

Controlling the outside view of a module Signatures can be used to control interactions between modules in two ways. First, the *outside view* of a module can be restricted to protect internal invariants by an explicit *ascription* to a given module type (line 1). This basically an up-cast, requiring a subtyping check between the inferred signature of the module expression (here, the inferred signature of `Complex`) and the user-provided one (here `Ring`). This check is *structural*: there is no need to mention the `Ring` signature at the definition point of `Complex`, it is sufficient to have the appropriate fields. By contrast, with nominal subtyping, the subtyping relations must be made explicit up front, which is less flexible and imposes an order in the definitions of module and interfaces.

Ascriptions can be used to (1) *hide* fields (making them inaccessible from the outside), (2) *reorder* fields (regardless of the original order of definitions), and (3) *abstract* type components, which hides the underlying implementation while keeping the name visible. The basic form of ascription is also called *opaque ascription*—we will later see *transparent ascription* in Section 1.2.4. Here, `CRing.t` is an available type, but its implementation as a pair of floats is hidden: one cannot coerce a pair of floats into a `CRing.t` value.

Controlling dependencies of a module Signatures can also be used to restrict how a given module depends on (i.e., uses) other modules. This is achieved by turning the module into a *functor*. Here, `Polynomials` is a functor that can take any implementation `R` satisfying the `Ring` interface and that returns an implementation of the ring of polynomials over `R`. The *body* of the functor is *polymorphic* with respect to the abstract type fields of its argument, and thus, does not depend on their actual implementations. Functors can then be called and composed: `Polynomials` can be applied to modules satisfying `Ring` such as `Complex` and `CRing` (lines 19), but also the output of `Polynomials` itself (line 21). There is a subtyping check at functor application, which induces a run-time coercion.

[1]The terminology is not fully stabilized in the literature regarding the distinction between *bindings* for fields of structures and *declarations* for fields of structural signatures. In some works like Rossberg and Dreyer [2013], bindings are called *definitions*, and declarations are called *specifications*.

Hierarchy Modules can be nested inside other modules as *sub-modules*. Correspondingly, submodules can be extracted by *projection*: `M.X` extracts the submodule `X` from `M`.

Higher order functors As expected by users of functional languages, functors can be *higher order*: they can take another functor as input. Historically, higher order-functors have been a technical challenge, especially before the introduction of applicative functors (see Shan [2004]; Dreyer et al. [2003]; Leroy [1995]; Biswas [1995]; Harper and Lillibridge [1994]; Harper et al. [1989], among others).

Overall, we have the following basic features

> ✓ M **Basic modularity**: structures, signatures, value bindings/declarations, submodules, type and module-type bindings/declarations.
> ✓ C **Type manifests**: creating new names for already-existing types
> ✓ M **Abstract types**
> ✓ M **Interface control**: ascription (interface of exports) and functors (interface of imports)
> ✓ M **Higher-order functors**

Regularity and robustness A good property for languages, especially in the context of modularity, is its *regularity*: the user should be able to arbitrarily nest and use the features, without restrictions like being *at top-level*. It is partially the case in OCaml: submodules can be arbitrarily nested, ascription can be used at any point, functors applications can be written with an anonymous functor and anonymous argument, functors can be higher-order. However, other constructs, such as projections, are syntactically restricted to module identifiers (we expand on the technical reasons behind this choice in Section 1.3). This is detrimental for the usability of the system, as users might run into hard-to-predict unsupported edge-cases when trying to have an advanced use of the language.

Another good qualitative property is the *robustness*: the the predictability As an example, the language support neither *let-binding* nor *inlining of definitions*! Let-binding can break the syntactic criterion of applicativity, which we discuss in Section 1.2.3—it can be fixed with *transparent signatures*. Inlining the definition of a submodule (or *let-inlining*) can cause signature avoidance, as discussed in Section 1.3. This breaks the *subject reduction* property of the intuitive call-by-value semantics.

> ● P **Robustness**

Summary At an high-level, ML modules can be understood as a small separate calculus built on top of a source language, with standard features re-branded with ML nomenclature. *Structures* are basically records: *bindings* are record-field expressions and *declarations* are record-field types. *Projection* is just record access. *Ascriptions* are explicit type castings (only *up-casts*), and *functors* are just functions. However, in addition, signatures contain abstract types. The crux of ML modules is the interaction between abstraction and other features, which we discuss in the following sections.

Other introductions to ML-modularity can be found in Dreyer et al. [2005]; Russo [2004]; Leroy [2000] (among others) or in Pierce [2004] (Chapter 8).

```
module Tokens () = (struct
    type t = int
    let x = ref 0
    let fresh () = let n = !x in x := n + 1 ; n
    let eq x y = (x = y)
  end : sig
    type t
    val fresh : unit → t
    val eq : t → t → bool
  end)
module PublicTokens = Tokens()
module PrivateTokens = Tokens()
(** PublicTokens.t =/= PrivateTokens.t *)
```

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

```
module OrderedSet (E:Ordered) = (struct
    type t = E.t list
    let empty : t = []
    let rec add x s = match s with
    | [] → [x]
    | [y::s] → if (E.lt x y) then (x::y::s)
               else (y::(add y s))
  end : sig
    type t
    val empty : t
    val add : E.t → t → t
  end)
module S1 = OrderedSet(Integers)
module S2 = OrderedSet(Integers)
(** S1.t === S2.t *)
```

(b) An applicative functor — Functors are applicative by default in OCaml. Here, `OrderedSets(E)` is a module implementing (ordered) sets of elements of type `E.t`. Applicative functors can be used in paths directly, leading to `S1.t = OrderedSets(Integer).t`.

Figure 2: Examples of generative and applicative functors.

1.2 Functors and abstraction

1.2.1 Applicative and Generative Functors

Both modules and functors can be used to either *structure* the code base or to build *reusable components*. In the latter case, several instances of a functor application may be available in the context when combining different pieces of code. This is typically the case for modules providing common data-structures such as lists, hash-tables, sets, etc. When such functor produces abstract types, a question arises: should every instance of the same application produce incompatible abstract types, i.e., types not considered equal by the typechecker? This question leads to the distinction between *applicative* and *generative* functors, which have different semantics and correspond to different use cases. Both are supported by OCaml and illustrated in Figure 2. If two instances have *equal* abstract types, there are effectively *compatible* and the functions and values from each module can be used together. We say that two instances are *incompatible* when they have different abstract types.

Generative functors Applying a *generative functor* twice *generates* two incompatible modules, with incompatible abstract types. The body of such functor might be stateful, emits effects, or dynamically choose the implementations of its abstract types (using *first-class modules*). Generativity can also be used by programmers as a strong abstraction barrier to force incompatibility between otherwise pure and compatible data-structures that represent different objects in the program. OCaml *syntactically* distinguishes generative functors from applicative ones by requiring the last argument to be a special unit argument "`()`". Therefore, generative and applicative functors are *syntactically* distinguished: the unit argument () is not the same as an empty structure. We find this effect-suggesting syntax of taking a unit argument fitting, as calling a generative functor can indeed produce effects.

Applicative functors Conversely, applying an *applicative functor* twice with *the same argument* produces compatible modules, with the same abstract types. The body of such a functor must be pure and have a static implementation of its abstract types. In OCaml, it is left to the user's responsibility to mark impure functors as generative, the typechecker does not track effects, only preventing unpacking of first-class modules and calling generative functors inside the body of applicative ones. Applicativity acts as a weaker abstraction barrier, making several instances of the same structure compatible. This is especially useful to provide generic functionalities (such as hash-maps[2], sets, lists, etc.) that may appear in several places and yet be compatible. Applicative functors are the default in OCaml.

One could think that applicative functors are not *necessary*: the programmer could always have factorized in advance the functor applications that happen to be duplicated and only use generative functors. In practice, this would be very restrictive, as it requires a non-local program-wide change and prevents many concise patterns. In practice, applicative functors where introduced by Leroy [1995] and quickly became one of the killer features of OCaml, greatly contributing to the success of the language. Generative functors, the default in SML, were added to OCaml only latter on.

- ✓ C **Generative functors**: necessary for effectful modules or dynamic choice of implementation (using first-class modules, see Section 1.6.1)
- ✓ C **Applicative functors**: crucial for sharing of pure functors without *a priori* factorization of similar applications
- ✚ M **Effect tracking at the core level**: applicative functors bodies should be pure. As of now, it is left to the user responsibility, whereas a type-system check of purity would provide a much stronger guarantee.

1.2.2 Abstraction Safety and Granularity of Applicativity

A key design point is the *granularity* of applicative functors: under what criterion should two applications of a functor produce compatible modules? This problem was coined the *module equivalence problem* by Dreyer [2007a]. We say that two modules are *equivalent* when applying the same functor to both yields compatible abstract types.

An option, used in Moscow ML, is to consider modules to be equivalent when they have the same type fields (same names and same definitions[3]). This criterion is called *static equivalence* Shao [1999]; Russo [2000]; Dreyer et al. [2003]. It is *type-safe*: in the absence of first-class modules, which are forbidden in applicative functor for that very reason, the actual implementation of the abstract types produced by the functor can only depend statically

[2] `Hashtbl.Make` is actually pure, as it does not produce a new hash table itself, even though it contains impure functions.

[3] This relies on some equality of types. For instance, it could be the cloture of syntactic equality and applicativity – which might be problematic for recursive signatures with applicative functors.

```
module S1 = Set.Make(struct type t = int let lt = (>) end)
module S2 = Set.Make(struct type t = int let lt = (<) end)

let s = S1.(empty |> add 1 |> add 3 |> add 2) (* [1,2,3] *)
let s' = S2.add 2 s1 (* [2,1,2,3] *)
let test = S1.mem 2 s' (* false *)
```

Figure 3: A simple example of a breach of abstraction safety. With the static equivalence criterion, this code type-checks but mis-behaves. Specifically, the implementation of sets as lists assumes that the list is ordered with respect to the comparison function. However, `S1` and `S2` do not have the same comparison function, but still have compatible abstract types.

on its parameters, thus only on its *statically known* type fields. Therefore, requiring static equivalence is sufficient to ensure that the abstract types produced by the functor are the same.

However, the static equivalence criterion can make two functor applications compatible while they actually have different internal invariants. In the example on Figure 3, the lists used to represent sets are ordered with respect to the comparison function of the argument of the functor. Therefore, a typechecker implementing the static equivalence criterion would allow a user to mix sets ordered with different ordering functions, which would produce wrong results—but not crash.

Yet, developers often expect a stronger property called *abstraction safety*: abstract types should protect arbitrary local invariants that may also depend on values. In the example on Figure 3, applications of `OrderedSets` should produce compatible abstract types only when both the type `E.t` and the values, in particular the comparison function `E.lt`, are the same. Crucially, abstraction can protect invariants that cannot be expressed in the type-language[4].

To preserve abstraction safety, we need *dynamic equivalence*, i.e., the runtime equality of values. Unfortunately, it is undecidable in general. Besides, tracking even an approximation of the equality of value fields would be too fine-grained and cumbersome, as modules may have numerous value fields. To enforce abstraction safety while remaining practical, OCaml follows a *coarse-grain approach*: tracking equalities only at the module level (not at the value level). This was originally introduced as a *syntactic criterion* by Leroy [1995]: two functor applications produce the same abstract types when they are syntactically identical. The syntactic criterion therefore acts as a gross approximation of dynamic equivalence, preserving abstraction safety.

> ✓ P **Abstraction safety**: type abstraction should protect arbitrary invariants. While easy to verify for the rest of the language, applicative functor need a special treatment: their module equivalence criterion should be a subset of dynamic equivalence.

As a summary, applicativity of functors relies on a criterion for *module equivalence*. We distinguish two notions of equivalence:

- *Static equivalence*: the two modules have the same type fields (for some notion of type equality). Using static equivalence for applicativity yields *type-level applicativity*.
- *Dynamic equivalence*: the two modules have the values at runtime. A subset of dynamic equivalence can be tracked statically by the type-system, either (1) by tracking equality between value fields, yielding *value-level applicativity*, or (2) by tracking equality between modules, yielding *module-level applicativity*.

[4]This is a key point when comparing the module system of OCaml and proof assistants like Coq. In the latter, all the properties of objects can be expressed directly as lemmas (or using dependent types) and preserved by the type-system. Abstraction only serves to limit the reach of internal details.

1.2.3 Module level equalities and aliasing

The *syntactic criterion* is however somewhat fragile, as, for example, it does not support naming sub-expressions. In this subsection we present the OCaml extension of *module aliases* that extends this criterion in a somewhat restricted way.

Two paths with functor applications are considered equal when they are syntactically equal:

```
module F (_:sig end) = struct end
module G (_:sig end) = struct type t end
module X = struct end
let f : G(F(X)).t → G(F(X)).t = fun x → x (* typechecks *)
```

However, naming a sub-expression breaks the syntactic equality:

```
module FX = F(X)
```

```
let f': G(FX).t → G(F(X)).t = fun x → x
```

```
Error: This expression has type G(FX).t but an expression was expected of type
         G(F(X)).t
```

Here, the type of module `FX` should manifest its module-level equality with `F(X)`. Interestingly, the OCaml feature of *module aliases* partially covers this need, even-though it was historically introduced for unrelated reasons.

Module aliases To allow for a better management of namespaces, *type-level module aliases* were added to OCaml: the signature language was extended with the *alias declaration* construct `module` $X = P$ to express that a module is a *statically known alias* of the module at the path P. The main motivation behind this extension was to provide a mechanism to give short names to modules without actually duplicating the module at runtime. It is especially useful for accessing the standard library:

```
module M = struct
  module A = StdLib.Array (* example of short name with module aliases *)
  (* ... *)
end
```

Internally, the compiler maintains a flag to decide if the submodule can be removed from the structure at runtime (*absent alias*, also called *static alias*) or should be kept (*present alias*, also called *dynamic alias*). As it was designed as a namespace feature, it made sense to try to prevent a dynamic copy of the aliased module if not necessary. However, it effectively merged in the same syntax two very different features, and the user cannot easily control if the aliased module will be present or absent in the end.

Yet, it opened the door to module-level equalities, somewhat reviving the structure sharing mechanism of SML'90. In particular, the syntactic criterion was extended to include module aliases (both static and dynamic), making the following code typecheck:

```
module F (_:sig end) = struct type t end
module X = struct end
module X' = X
let f : F(X).t → F(X').t = fun x → x
```

Restrictions However, the motivating example at the beginning of this section would not typecheck: the aliasing of OCaml was designed for static aliases and is therefore quite limited. In practice, for a declaration **`module`** `M = P`, the path `P` has the following restrictions (extract of the OCaml documentation Leroy et al. [2024])[5]:

[5] https://ocaml.org/manual/5.2/modulealias.html

1. it should be of the form `M0.M1...Mn` (i.e. without functor applications);
2. inside the body of a functor, `M0` should not be one of the functor parameters;
3. inside a recursive module definition, M0 should not be one of the recursively defined modules.

The restrictions 1 and 3 were put in place to ensure that the aliased module is already dynamically present, to make the static redirection meaningful. But those are not necessary for dynamic aliases! (Restriction 2 is discussed in the next subsection)

Overall, we advocate for a separation of the two mechanisms for aliasing. The simpler static aliasing is well suited to remain a *declaration* made inside an enclosing module, and the aliased module cannot contain functor application, other recursive modules or functor parameters. It should have a different syntax than dynamic aliases (discussed in the next subsection).

> C **Static aliases**: to manage name-spaces (provide short names for modules), the user should be able to define *static aliases* to other, already defined, modules. Static aliases do not induce a copy of the aliased module, they are removed at runtime.

In the rest of this thesis, "aliases" refer to dynamic aliases by default.

1.2.4 Aliases and transparent ascription

In this subsection we present a technical issue with aliases, solved by the introduction of a new feature of *transparent signatures* that provides a robust extension of the syntactic criterion of applicativity.

Unlike static aliases, dynamic aliases don't have to be restricted to paths without applications and could contain other recursive modules. Besides, the aliasing does not have to be at the declaration level, but could very well be at the signature level. However, paths containing functor parameters are an issue[6]. As an example, let us consider the following code:

```
module X : T = (* ... *)
module F (Y:S) = Y (* reexport *)
module X' = F(X)
```

The module `X'` cannot be given the expected alias signature `module X' : (= X)`, which contradicts the substitution-based intuition. Indeed, if the type system were to maintain module aliases through functor calls, it would impose strong constraints on the compilation of structures and functors that would drastically affect the performance trade-offs of modules.

To see why, one can consider the compilation of the body of the functor `F`. In order for the body of the functor to be an alias of the argument, it should contain *all* of the fields of its parameter, *possibly more* than the ones indicated in the signature `S`, and in an *order that might be different.* Therefore, the static view of `Y` is a potentially strict supertype of the dynamic structure given at functor application—while the static view is used to perform direct accesses inside `Y`. Overall, this would force structures to have a dynamic representation where they can always be seen as a subtype of their actual type, i.e., a representation with *code-free* subtyping. Typically, using dictionaries with a *dynamic dispatch* through an access-table provides a code-free subtyping representation of structures, at the cost of slower accesses (due to the indirection).

However, in all ML-module systems, including SML and OCaml, structures are compiled via *static dispatch*: accesses inside a structure are made with fixed offsets to be fast. In such setting, subtyping is not code free, and explicit coercions are inserted at functor calls, which break module aliases. Therefore, the type system does not follow the naive substitution semantics for aliases inside functors, but instead uses a set of syntactically-based restrictions

[6]This issue stalled the pull request OCaml #10435 that aimed at extending the alias mechanism

to prevent them in functor signatures. Those restrictions are not stable under substitution and can be bypassed in some edge-cases[7].

> ✓ M **Static dispatch**: "*modules are often accessed but rarely reordered*", the type system should be compatible with a static dispatch compilation scheme that favor fast access over code-free subtyping.

Transparent ascription to the rescue Interestingly, *transparent ascription*, originally introduced as a module expression written `(M : S)` in SML, helps lifting this restriction. It restricts the outside view of a module `M` to the fields present in the signature `S` while preserving all type equalities. However, this feature does not increase expressiveness in SML as a similar result could be obtained via a *usual* (opaque) ascription with a signature where all type equalities have been made explicit. A proposal for OCaml[8] is to add transparent ascription as an extension not only of the module language, but of the *signature* language, writing `(= P < S)` for the signature of a module that is an alias of `P` but restricted to the fields of `S`, which we call a *transparent signature*. A module with such a signature has a *module-level equality* with `P` and the content `S`. We say that it shares the same *identity* as `P`.

Transparent signatures provide a generalization of aliasing, storing both the aliasing information *and* the actual signature (hence, the memory representation). The transparent ascription *expression à la* SML `(P : S)`, is then just syntactic sugar for an opaque ascription with a transparent signature `(P :> (= P < S))` (in SML syntax).

Thanks to transparent signatures, aliasing information can be preserved through the implicit ascription at functor calls. As OCaml features applicative functors (unlike SML), this would increase the expressiveness of the signature language. Besides, concrete signatures are compatible with static dispatch and copying at function calls allows deletion and reordering of fields while keeping type equalities. The motivating example of the beginning of this section would give :

```
module X : T = (* ... *)
module F (Y:S) = Y (* reexport *)
module X' = F(X)
```

```
module X' : (= X < S)
```

Another, more realistic example of code pattern where transparent signatures are necessary is given in Figure 4. This pattern arises when combining a functor (`Make3D`) that reexports its argument (`K`) with an applicative functor called twice (`Set`), once on the argument and once on the reexported argument so that the modules resulting from both applications may interact. Here, the fixed interface `VectorSpace` could not be *functorized* to explicit the dependency with the underlying field, as not all vector spaces are functors over a field. Besides, type-level sharing is not sufficient to obtain the right type equalities when calling the `Set` functor.

> ✚ C **Transparent signatures (dynamic aliases)**: to extend the syntactic criterion of applicativity in a manner that composes well with functors, we propose to store both the module level equality and the dynamic signature using transparent signatures. This would lead to a much more robust notion of module equivalence that extends the syntactic criterion.
>
> M **Robust let-binding with submodules**: using transparent signatures, the user should always be able to bind module expressions as submodules without breaking

[7]See the following issues: OCaml #7818, OCaml #2051, OCaml #10435, OCaml #10612 and OCaml #11441.

[8]OCaml #10612. Transparent ascription is written `(P :> S)` in OCaml #10612, the opposite of the SML convention.

```
1  (* Interface definitions *)
2  module type Field = ...
3  module type VectorSpace = sig
4    module Scalar : Field
5    ... (** more fields *)
6  end
7
8  (* Basic data-structure *)
9  module Set(Y:...) = ...
10
11 (* Extension of a vector space
12    with additional features *)
13 module LinAlgebra(V:VectorSpace) =
14 struct
15   ...
16   module SSet = Set(V.Scalar)
17   ...
18 end
```

```
19 (* Special case of vector space
20    built directly from a field *)
21 module Make3D(K:Field) = (struct
22   module Scalar = K
23   ... (** built from K *)
24 end : sig
25   module Scalar : (= K < Field)  ...
26 end)

27 (* Implementation of real numbers ℝ *)
28 module Reals = ...
29
30 (* Vector space ℝ^3 with extensions *)
31 module Space3D =
32   LinAlgebra(Make3D(Reals))
33
34 (* Are sets of reals the same
35    as sets of scalars of ℝ^3 ? *)
36 Space3D.SSet.t =? Set(Reals).t
```

Figure 4: An example of code pattern where transparent signatures are necessary. On the left-hand side, `VectorSpace` defines an interface for vector spaces which contains a sub-module `Scalar` for the field of scalar numbers. The functor `LinAlgebra` (line 14) uses a vector space to define linear algebra operations, one of them using sets of scalar numbers. At some other point in the development (line 21), 3D vector spaces are built directly from any field `K` via the functor `Make3D`. Its signature contains a transparent signature on its parameter `K`. Finally, on line 32, the module `Space3D` implements linear algebra for the vector space $\mathbb{R}^3$. We want the inner sets `Space3D.SSet.t`, and `Set(Reals).t` to be compatible. This requires the aliasing information to be kept between the parameter and the body of the functor `Make3D`. Currently, OCaml fails to share and identify the types `Space3D.SSet.t` and `Set(Reals).t`.

type-checking.

Transparent signatures `(= P < S)` can be seen as a special case of the more general module sharing mechanism of *F-ing* (Rossberg et al. [2014]) `like P`: we have could define `(=P<S)`$\triangleq$`(like (P : S))` (with a transparent ascription of `P` by `S`). This sharing mechanism relies on a general notion of *semantic* paths which stand for any module expression that does not introduce new abstract types whereas OCaml uses more restrictive *syntactic* paths. Therefore, module sharing signatures with semantic paths could also play the role of transparent signatures. However, we argue that semantic paths are *too powerful for their own good.* By injecting arbitrary module expressions in signatures, it makes it hard for the user to know if two modules are equivalent.

As transparent signatures enable more sharing of identities in signatures of inferred modules, aliasing becomes a good static approximation of that equivalence.

1.3 A Key Challenge: the Signature Avoidance Problem

In this section, we present the *signature avoidance* problem, a key issue of ML module systems that comes from the interaction between abstraction and projection. In Section 1.3.1, we introduce the signature avoidance problem through examples. In Section 1.3.2, we discuss possible strategies to handle it. We give some considerations for the design of an algorithm that can be used to *solve* the avoidance problem, which we call *anchoring.* In Section 1.3.3, we propose a restriction of anchoring in the presence of applicative functors for a better usability. We conclude with some practical aspects in Section 1.3.4.

1.3.1 Introduction to the avoidance problem

At its core, signature avoidance originates from interaction of three mechanisms. First, type abstraction creates new types that are only compatible with themselves (and their aliases). Then, sharing abstract types between modules, which is essential for module interactions, produces inter-module dependencies in signatures. Finally, projection (and other mechanisms) allows the user to hide a type or module components, which can *break* such dependencies: types can be removed from the scope while they are still being referenced. Sometimes, no possible signature exists for a module in the source syntax; other times there are several incomparable ones (no principal source signature). A detailed overview of the avoidance problem can be found in Crary [2020].

In the literature, the nomenclature is not fully stabilized: sometimes the *avoidance problem* refers to cases where no source signature exists, sometimes to the process of finding a signature that avoids a certain type. In this thesis, we use *signature avoidance* or *the signature avoidance problem* to refer to the challenge of finding a signature that avoids a certain type, if it exists. The algorithm that actually finds such signature is called the *anchoring algorithm.*

Anonymous projection We start with an example that is not in valid OCaml syntax:

```
module M = (struct
  type t = A of int | B
  module X = struct
    let x = A(42)
  end
end).X
```

The module `M` is built by projecting *only* the submodule `X`, which exposes unsolvable dependencies with a type `t` that has become unreachable. There is no way of writing a signature for `M` that *avoids* `t`:

```
sig x : ? end
```

OCaml disallows *anonymous projections* precisely to prevent this type of patterns.

Anonymous functor call However, OCaml allows anonymous functor call, which leads to similar situations:

```
module M =
  (functor (Y:sig type t end) → struct
    let x : Y.t list = []
  end)(struct type t = A | B end)
```

```
Error: This functor has type functor (Y : T) → sig val x : Y.t list end
       The parameter cannot be eliminated in the result type.
       Please bind the argument to a module identifier.
```

Here, the algebraic datatype of the anonymous argument cannot be referred and is yet necessary for the resulting signature: there is not signature *avoiding* this type. The error message suggests a solution: naming the module argument to make its type components available for the rest of the program.

Anonymous `open` Finally, OCaml also offers anonymous **`open`**, a feature that is discussed in Section 1.5.1. We give here an example of signature avoidance using **`open`** for completeness, but it can be skipped by a reader unfamiliar with the feature:

```
module M = struct
  open (struct type t end)
  let x : t option = None
end
```

```
Error: The type t/4046 introduced by this open appears in the signature
Line 5, characters 6-7:
The value x has no valid type if t/4046 is hidden
```

The typechecker tries to remove the name `t` of the resulting signature, but it cannot: there is no signature *avoiding* this type.

Overall, signature avoidance can appear as long as there are ways to hide abstract types from the typing environment, creating dangling dependencies.

1.3.2 Strategies

There a three main strategies to handle the avoidance problem:

1. Preventing the loss of abstract types. For instance, naming all module expressions make all abstract types always accessible and the avoidance problem disappears. However, it greatly limits the usability of the module calculus and prevents a fine-grained management of types: it becomes impossible to hide intermediary module constructions without explicit ascriptions.

2. Trying to rewrite the signature to avoid the out-of-scope types. This might require deep rewrites inside the signature, to abstract type components, as done by Crary [2020]. OCaml uses an incomplete (undocumented) heuristic that can lead to *loss of type-sharing*: type equalities between in-scope types might be lost. We detail it below.

3. Extend the signature syntax to account for the existence of out-of-scope types, as done in SML'90 Milner et al. [1997] and in Dreyer et al. [2003]; Russo [2004]; Rossberg et al. [2014]; Harper and Stone [2000]. From a formalization point of view, the extension of syntax forces a specification *by elaboration*, where signatures are translated in a more expressive language, more or less distant from the source one.

The rewriting heuristic of OCaml When a signature mentions an out-of-scope type, say `t` for instance, OCaml uses the following heuristic:

1. If the type appears in a type field of the form `type u = t` that is in a strictly positive position, the field is transformed into an abstract type field `type u`
2. If the type appears in a module type field, the whole module type field is transformed into an abstract module type, which is actually a bug[9] .
3. If the type appears in any other position (value field, non-strictly positive type field, etc), a type-error is thrown.

Loss of type-sharing While simple, this heuristic has some surprising behavior. Let us consider the following code:

```
module F = functor (Y:sig type t end) → struct type u = Y.t type v = Y.t end
module M = F(struct type t = A | B end)
```

Here, the signature of `X` mentions an out-of-scope type `t` that should be avoided. OCaml make all type declarations that mention `t` abstract, giving the following signature [10]:

```
module F : functor (Y:sig type t end) → sig type u = Y.t type v = Y.t end
module M : sig type u type v end (* OCaml *)
```

Here, there is a loss of type-sharing: the type `u` and `v` are no longer equal! It can lead to errors latter in the program:

```
let f : M.u → M.v = fun x → x
```

```
Error: This expression has type M.u but an expression was expected of type M.v
```

Yet, a better signature was possible if instead of abstracting all type declarations mentioning `t`, the typechecker would recognize `u` as an available in scope alias that can be used to replace `t`:

```
module M : sig type u type v = u end
```

Inside the body of the functor, `Y.t` and `u` refer to the same type, and using one or the other in the definition of `v` should not change the result. Yet, writing `type v = u` instead of `type v = Y.t` would change the signature inferred by OCaml: the avoidance resolution heuristic would not rewrite a declaration that does not use out-of-scope types. Overall, the resolution mechanism is quite brittle: the typechecker can lose type equalities, and this depends on the choice of (equivalent) type aliases in the definitions.

P **Type-preserving anchoring**: all type equalities should be preserved: i.e., types that are considered equal before a projection and that remain in scope should still be considered equal after a projection. Type definitions should not be abstracted away silently, but only by an explicit opaque ascription. Resolution of avoidance should not depend on the choices of aliases in the definitions.

[9] Researching this behavior led us to uncover this bug and open the issue OCaml #10491

[10] In OCaml 4.14, the `-short-path` option can give a misleading output, writing the following signature for the functor: `module F : functor (Y:sig type t end) → sig type u = Y.t type v = u end`. It is misleading, because internally the type equality is `type v = Y.t` and not `type v = u`. Therefore the avoidance algorithm will in fact rewrite the type field, making it abstract `type v` in the resulting signature after the application.

1.3.3 Avoidance with applicative functors

Introducing applicative functors in the mix adds a new layer of complexity. While it is possible to define a type-preserving resolution of avoidance that supports applicative functors, we believe that it should be restricted to rely on module equivalence. To support that point, we present the difficulty of designing such resolution through examples, but, more importantly, we show why we believe that it would be impractical.

Change of domain The abstract type of applicative functors can be thought of as type functions over a certain *domain*: the signature of the parameter of the functor.

```
(struct
  module F (Y:S) = struct type t end
  module R = struct
    module F1 (Y:S1) = struct type t = F(Y).t end
    module F2 (Y:S2) = struct type t = F(Y).t end
    module X : S' = ...
    type t = F(X).t
  end
end).R
```

Using the abstraction heuristic of OCaml, we would get:

```
sig
  module F1 (Y:S1) : sig type t end
  module F2 (Y:S2) : sig type t end
  module X : S'
  type t
end
```

All type equalities have been lost, while some could be kept! To preserve type-sharing, one might be tempted to abstract the type-definition inside `F1` and rewrite all paths using `F` with `F1` instead. However, it might not be possible, as `F` and `F1` do not have the same parameter signature (resp. `S` and `S1`). To check if it correct, one would have to browse the rest of the signature to find if `F` is only "used" on a subset of its domain, which makes the resolution of avoidance non-local. If `S2` and `S'` happen to be subtypes of `S1`, then the following signature is correct:

```
(* if S2 < S1 and S' < S1 *)
sig
  module F1 (Y:S1) : sig type t end
  module F2 (Y:S2) : sig type t = F1(Y).t end
  module X : S'
  type t = F(X).t
end
```

If, instead, `S1` and `S2` are incompatible, i.e., no signature is subtype of both, then it would not loose type-sharing to abstract the type fields inside the body of both functors:

```
(* if there is no S such that S < S1 and S < S2 *)
sig
  module F1 (Y:S1) : sig type t end
  module F2 (Y:S2) : sig type t = F1(Y).t end
  module X : S'
  type t = F(X).t
end
```

Change of arity In addition to the domain, the arity of abstract types could be changed. We modify the previous example by adding a dummy parameter to `F1`:

```
(struct
  module F (Y:S) = struct type t end
  module R = struct
    module F1 (Y:S1) (Arg: sig end) = struct type t = F(Y).t end
    module X : S' = ...
    module A = struct end
    type t = F(X).t
  end
end).R
```

Again, the abstraction heuristic of OCaml would abstract all type fields mentioning `F`. However, in order to use `F1` in place of `F`, we would have to chose a second argument to the application. This choice would be completely arbitrary and very surprising to the user:

```
sig
  module F1 (Y:S1) (Arg: sig end) : sig type t end
  module X : S'
  module A : sig end
  type t = F1(X)(A).t
end
```

No invention of functor application Both the examples above constitute cases where the typechecker would try to solve signature avoidance by creating new functor applications "out of thin air", just to refer to types that have lost their original path. While sound, this would be very surprising to the user, if not misleading. Let us consider this final example, where the projection renders the `List` functor inaccessible.

```
module M = (struct
  module type T = ...
  module Lists (Y: T) : sig type t (* other fields ... *) end = (* ... *)
  module R = struct
    module Sets (E: T) = struct
      (* ... *)
      module ToList = struct
        type elist = Lists(E).t
        (* extraction from sets to lists ... *)
      end
    end
    module X = (* ... *)
    type t = Lists(X)
  end).R
```

An advanced anchoring algorithm could find the following signature:

```
module M : sig
  module Sets(E:T) : sig
    (* ... *)
    module ToList : sig
      type elist
      (* .. *)
    end
  end
  module X : (* ... *)
  type t = Sets(X).ToList.t
end
```

Here, it *suggests* the computation of the module `Sets(X)`, while this application was no present in the original code. We take the stance that such rewrites, while doable, would be confusing for the user and very hard to predict.

Restricting anchoring Overall, types inside functor applications cannot be considered "independtly" of the functor they were defined in. So if a functor `F` is lost by projection, when should anchoring *not throw* a type error ? Here, we propose to answer: if a functor `F'` *equivalent* to `F` is still in scope! We reuse the notion of module equivalence explored in Section 1.2.4: `F'` must have a *transparent signature* of the form `(= F < ...)`. This gives a simple and predictable criterion, which we propose for the design of anchoring:

> ⊕ P **Resolution of avoidance based on module equivalence**: a path `F(X).t` can avoid `F` if a module equivalent to `F` is still in scope, and can avoid `X` if a module equivalent to `X` is still in scope.

We believe that the simplicity of this criterion is a key design choice for usability: it reduces the problem of higher-order avoidance to a first-order problem at the module level.

1.3.4 Signature avoidance in practice

Naming modules OCaml users usually get around signature avoidance errors by explicitly naming modules before using them, which adds *always-accessible* type definitions. The module syntax of OCaml actually encourages this approach by limiting the places where inlined, anonymous modules can be used. In particular, projection on an anonymous module is forbidden. It is however cumbersome, as it prevents some concise code patterns and forces to expose type definitions.

Interface files Moreover, in practice, OCaml developers often have interface files (with the extension `.mli`) that behave as a file-wide ascription. This means that signature avoidance is actually a problem that occurs during typechecking of intermediary module expressions, before the final ascription forces a given signature. Therefore, we advocate for a mechanism that does not fail when sub-expressions cause signature avoidance:

> ⊕ C **Delayed avoidance**: the module system should support signatures with out-of-scope types for intermediary module expressions

1.4 Module-level abstraction

In this section we present *module-level abstraction.* Just as signature can contain abstract type declarations of the form **`type`** `t` (without a definition), they can contain abstract *module-type* declarations, of the form **`module type`** `T` (without a definition). Just as core-level abstraction, module-level abstraction can be used for ascription, to hide definitions, or with functors, for polymorphism.

Abstract module types are both one of the most esoteric, less understood, and under-appreciated features of the OCaml module system. They have their own set of challenges.

1.4.1 Abstract signatures

We introduce abstract signatures via two on-going examples. Fundamentally, they will not be surprising to readers used to rely on abstraction to protect invariants and factor out code. Their specificity lies in the fact that there are at the module level, and therefore require projects with a certain size and a strong emphasis on modularity to be justified.

Module sealing

Let's consider the following scenario. Two modules providing an implementation of UDP (`UDP1` and `UDP2`) are developed with different design trade-offs. They both implement a signature with basic send and receive operations. Then, functors are added as layers on top: taking a udp library as input, they return another udp library as an output.

- `Reliable` adds sequence numbers to the packets and re-sends missing packets;
- `CongestionControl` tracks the rate of missing packets to adapt the throughput to network congestion situations;
- `Encryption` encrypts the content of all messages.

The resulting signature might look something like:

```
module type UDPLib = sig
  module type UDP = sig
    val send : string → unit
    val get : unit → string
    (* ... *)
  end

  module UDP1 : UDP
  module UDP2 : UDP

  module Reliable : UDP → UDP
  module CongestionControl : UDP → UDP
  module Encryption : UDP → UDP
end
```

However, a project might need different combinations of the basic libraries and functors, while requiring that **all** combinations use encryption. To enforce this, the solution is to use the module-level *sealing* of abstract signatures. In practice, the signature of the whole library containing implementations and functors `UDPLib` (typically, its `.mli` file) is rewritten to abstract the `UDP` interface, except for the output of the `Encryption` functor.

```
module type UDPLib = sig
  module type UDP (* abstract! *)

  module UDP1 : UDP
  module UDP2 : UDP

  module Reliable : UDP → UDP
  module CongestionControl : UDP → UDP
  module Encryption : UDP →
    sig
      val send : string → unit
      val get : unit → string
      (* ... *)
    end
end
```

Just as type abstraction, signature abstraction can be used to enforce certain code patterns: users of `UDPLib` will only be able to use the module after calling the `Encryption` functor, and yet they have the freedom to choose between different implementations and features:

```
module UDPKeyHandshake = Encryption(Reliable(UDP1))
module UDPVideoStream  = Encryption(CongestionControl(UDP2))
```

Module polymorphism

Another use is to introduce polymorphism at the module level. Just as polymorphic functions can be used to factor code, module-level polymorphic functors can be used to factor module expressions. If a code happens to often feature functor applications of the form `Hashtbl.Make(F(X))` or `Set.Make(F(X))`, one can define the `MakeApply` functor as follows:

```ocaml
module type Type = sig module type T end
module MakeApply
  (A:Type) (X: A.T)
  (B:Type) (F: A.T → B.T)
  (C:Type) (H: sig module Make : B.T → C.T end) = H.Make(F(X))
```

Downstream the code is rewritten using `MakeApply`. Right now, the verbosity of such example would probably be a deal-breaker. We address this aspect at the end . Ignoring the verbosity, this can be useful for maintenance: by channeling all applications through `MakeApply`, only one place needs to be updated if the arity or order of arguments is changed. Similarly, if several functors expect a *constant argument* containing – for instance – global variables, a `ApplyGv` functor can be defined to always provide the right second argument, which can even latter be hidden away to the user of `ApplyGv`:

```ocaml
(* Constant argument *)
module Gv : GlobalVars
module ApplyGv (Y : sig module type A module type B end)
    (F : Y.A → GlobalVars → Y.B)(X : Y.A) = F(X)(Gv)
```

Downstream, code featuring `F(X)(GlobalVars)` is rewritten into `ApplyGv(...)(F)(X)`. Then, the programmer can hide the `GlobalVars` module while letting users use `ApplyGv`, ensuring that global variables are not modified in uncontrolled ways by certain part of the program.

Finally, polymorphism can also be used by a developer to prevent unwanted dependencies on implementation details. If the body of a functor uses an argument with a submodule `X`, but actually does not depend on the content of `S`, abstracting it is a "good practice":

```ocaml
module F (Arg : sig ... module X : S ... end) =
struct (* the code can depend on the content of S *) end

module F' (Y: sig module type S end)
  (Arg : sig ... module X : Y.S ... end ) =
struct (* the code cannot depend on the content of S *) end
```

1.4.2 Challenges of abstract signatures

In this subsection we explore the issues posed by abstract signatures as currently implemented in OCaml.

Variant interpretation

The challenge for understanding (and implementing) abstract signatures lies more in the meaning of the module-level *polymorphism* that they offer than the module level sealing, the latter being pretty straightforward. More specifically, the crux lies in the meaning of the *instantiation* of an abstract module-type variable `A` by some other signature `S`, that happens when a polymorphic functor is applied. The substitution-based intuition ("*replacing all occurences of* `A` *by* `S`") has some surprising behaviors when the signature `S` contains abstract types, as they have a *variant* interpretation: an abstract type in positive position indicates sealing, while an abstract type in negative position indicates polymorphism.

Therefore, when instantiating an abstract signature with a signature that has abstract fields, the user must be aware of this, and mentally infer the meaning of the resulting signature.

To illustrate how it can be confusing, let's revisit the first motivating example and let's assume that the developer actually want to expose part of the interface of the raw `UDP` libraries. One might be tempted to instantiate `UDP` with something along the following lines[11]:

```
module type UDPLib_expose = sig
  include UDPLib with module type UDP =
    sig
      module type UNSAFE
      module Unsafe : UNSAFE     (* this part remains abstract *)
      module Safe : sig ... end  (* this part is exposed *)
    end
end
```

```
module type UDPLib_expose =  sig
  module type UDP =
    sig
      module type UNSAFE
      module Unsafe : UNSAFE
      module Safe : sig ... end
    end
  module UDP1 : UDP
  module UDP2 : UDP
  module Reliable : UDP → UDP
  module CongestionControl : UDP → UDP
  module Encryption : UDP → sig val send : string → unit (* ... *) end
end
```

However, the variant interpretation of this signature in the negative positions produces a counter-intuitive result. For instance, if we expand the signature of the argument for the functor 'Reliable' (for instance) we see:

```
module Reliable :
sig
  module type UNSAFE
  module Unsafe : UNSAFE
  module Safe : sig ... end
end → UDP
```

This means that the functor actually has to be *polymorphic* in the underlying implementation of `UNSAFE`, rather than using the internal details, which has the opposite meaning as before. If the user wants to hide a `shared` unsafe core, accessible to the functor when they were defined by then abstracted away, the following pattern may be used instead:

```
module type UDPLib_expose' = sig
  module type UNSAFE
  include UDPLib with module type UDP = sig
    module Unsafe : UNSAFE
    module Safe : sig ... end
  end
end
```

Doing so, the instantiated signature does not contain abstract fields and therefore its variant reinterpretation will not introduce unwanted polymorphism.

Impredicativity

Abstract module types are impredicative: a signature containing an abstract signature can be instantiated by itself. One can trick the subtyping algorithm into an infinite loop of

[11]We use the construct **`with module type`** for the instantiation, which is introduced in Section 1.6.2.

instantiating an abstract signature by itself, as shown by Rosseberg[12], adapting an example from Harper and Lillibridge [1994]:

```
module type T = sig
  module type A
  module F : A → sig end
end

module type I = sig
  module type A
  module F : (T with module type A = A) → sig end
end

module type J = (T with module type A = I)

module Loop (X:J) = (X:I)
```

During the typechecking of the functor `Loop`, a subtyping check is made between `J` and `I`, i.e., between `T[A <- I]` and `I`. This leads to instantiating `A` by `I` in the right-hand side. When comparing the two functor declarations, the subtyping between the two arguments is again (by contravariance) the subtyping between `T[A <- I]` and `I`: the typechecker is thrown in a loop.

Impredicativity also allows type-checking of (non-terminating) programs with an absurd type, as shown by the encoding of the Girard's paradox Leo White [13].

Module-level-sharing

Abstract signatures require module-level sharing, like the transparent signatures presented in Section 1.2.4. To see why, let us consider the following two functors:

```
module F1 (Y: sig module type A module X : A end) = Y.X
module F2 (Y: sig module type A module X : A end) = (Y.X : Y.A)
```

```
(* Currently, both are given the same type: *)
module F1 (Y: sig module type A module X : A end) : Y.A
module F2 (Y: sig module type A module X : A end) : Y.A
```

Here, we would expect the body of `F1` to keep type-sharing with its argument, while the `F2` uses an opaque ascription and should indeed loose type-sharing. With transparent signatures, we would get :

```
(* Currently, both are given the same type: *)
module F1 (Y: sig module type A module X : A end) : (= Y.X < Y.A)
module F2 (Y: sig module type A module X : A end) : Y.A
```

Other issues

The implementation of abstract signatures has a number of issues besides the theoretical ones mentioned above. We uncovered some during our work on this topic, notably: OCaml #12204, OCaml #10491.

1.4.3 Simple abstract signatures

In this section we present a solution for fixing the issues of the current approach, while keeping the core of the functionality. The main criticism we make of the OCaml approach is that it is actually *too expressive for its own good.* Having impredicative instantiation with variant

[12]See https://sympa.inria.fr/sympa/arc/caml-list/1999-07/msg00027.html

[13]See https://github.com/lpw25/girards-paradox/tree/master

reinterpretation is hard to track for the user and interacts in very subtle ways with other features of the module system, slowing down its development—and breaking its theoretical properties. To address this, we take the opposite stance and propose to make the system actually *predicative*: we restrict the set of signatures that can be used to instantiate an abstract signature. This also indirectly addresses the complexity of the variant reinterpretation.

> ✚ M **Predicative abstract signatures**: we propose to restrict the instantiation of abstract signatures by *simple signatures*: signatures that may contain abstract type fields but no abstract module-type fields.

Expressivity One might wonder how restrictive is this proposal. Specifically, if we consider a simple polymorphic functor as:

```
module Apply (Y : sig module type A end) (F : Y.A → Y.A)(X : Y.A) = F(X)
```

The following partial application would be rejected:

```
(* Rejected as A would be instantiated by
  ‘sig module type B module X : B → B end‘ *)
module Apply’ = Apply(struct
  module type A = sig module type B module X : B → B end
end)
```

However, this could be circumvented by eta-expanding, thus expliciting module type parameters, and instantiating only a simple signature:

```
(* Accepted as A is instantiated by a signature with no abstract fields *)
module Apply’’ = functor (Y:sig module type B end) →
  Apply(struct
    module type A = sig module type B = Y.B module X : B → B end
  end)
```

Module-type arguments for functors A key aspect of abstract module types that reduces their usability is the fact that signatures have to be given as part of a module. Instead, we propose *module-type* arguments for functors. In practice, they could be indicated by using brackets instead of parenthesis, and interleaved with normal module arguments, as in this example:

```
(* At definition *)
module MakeApply
    [A] (X:A)
    [B] (F: A → B)
    [C] (H : sig module Make : B → C end)
  = H.Make(F(X))

module ApplyGv
    [A] [B] (F:A → GlobalVars → B) (X:A)
  = F(X)(Gv)

(* At the call site *)
module M1 = MakeApply
    [T] (X)
    [Hashtbl.HashedType] (F)
    [Hashtbl.S] (Hashtbl)

module M2 = ApplyGv [A] [B] (F) (X)
```

Technically, this is not just syntactic sugar for anonymous parameters due to the fact that OCaml relies on names for applicativity of functors.

Module-type arguments for functors

Following up on the previous point, usability of abstract signatures could even be improved with some form of inference at call sites. Further work is needed to understand to what extend this could be done.

1.5 Composition

Composition refers to the ways different modules can be combined together. In this section we propose a *taxonomy* to classify the forms of composition and briefly discuss the associated features. Unlike previous sections, we do not summarize the features we add to our wish-list at the end of each subsection, but discuss them all at the end.

Forms of composition Let us assume that we want to compose two modules, $\texttt{M}_1$ and $\texttt{M}_2$. We propose the following criteria:

1. **Open vs Closed**: If we know the modules that are composed, here $\texttt{M}_1$ and $\texttt{M}_2$, the composition is *closed*. If instead, we want to compose $\texttt{M}_2$ with a module that is not chosen yet, we call the composition *open*. In such case, the user has to provide the signature of what is expected of the other module ($\texttt{M}_1$). Open composition is associated with a *resolution* where the module $\texttt{M}_1$ is actually chosen[14]. This resolution can be made at another point than the definition of $\texttt{M}_2$, and it allows $\texttt{M}_1$ and $\texttt{M}_2$ to be compiled separately. It also allows to create several instances by composing $\texttt{M}_2$ with different modules. Overall, open composition has some flexibility but requires a signature annotation, and therefore, forces a *fixed view* of the one module that is resolved.

2. **Sequential vs Recursive**: If $\texttt{M}_2$ depends (either for value definitions or type definitions) on $\texttt{M}_1$ but not the other way around, we can compose them in a *sequence*: first $\texttt{M}_1$ then $\texttt{M}_2$. If the dependency is mutual, i.e., $\texttt{M}_1$ depends on $\texttt{M}_2$ and $\texttt{M}_2$ depends on $\texttt{M}_1$, we need a *recursive* composition. Expressivity-wise, sequential composition is a sub-case of recursive composition, which is more general. However, recursive composition also has theoretical and practical challenges that we explore in Section 1.5.2, which explains why the simplicity of sequential composition is still appealing for users and language designers.

3. **Hierarchical vs Flat**: When $\texttt{M}_1$ and $\texttt{M}_2$ are structures (not functors), another distinction arises. After the composition of $\texttt{M}_1$ and $\texttt{M}_2$, if the content of both are in separate namespaces, the composition is called *hierarchical*: the namespace hierarchy keeps a trace of the composition. If instead, the content of the two modules are merged in the same namespace, the composition is called *flat*. Flat composition can create clashes of namespaces, i.e., *shadowing*. Overall, flat composition is more general than hierarchical, as the desired name-space hierarchy can always be explicitly added before composition, but at the cost of dealing with shadowing.

We sum up the associated features in Figure 5. As a nomenclature convention when describing a form of composition, we omit “closed”, “sequential” and “hierarchical”, which are (traditionally) the default choice. Therefore, “flat composition” refers to *closed*-*sequential*-flat

[14] At this point, a subtyping check is made between what was expected of $\texttt{M}_1$ and what is actually provided. $\texttt{M}_1$ is then seen through either an opaque or transparent ascription, depending on the features. For functor calls, the effective ascription is transparent.

		Sequential		Recursive	
		Hierarchical	Flat	Hierarchical	Flat
Closed		submodule	include/open	recursive modules	
Open	creation	functor		mixin module[b]	
	resolution	application	include functor[a]	composition/instantiation [b]	

Figure 5: Summary of the language constructions for composition. The holes indicate a form of composition for which we don't know a corresponding construction in any ML dialects.
[a]Experimental feature present in an extension of OCaml
[b]Implemented in MixML Rossberg and Dreyer [2013], an experimental language based on mixin composition

composition, while "open recursive composition" refers to open-recursive-*hierarchical* composition.

1.5.1 Hierarchical and flat composition

Hierarchical

The most basic form of composition is *closed-hierarchical-composition*, which is provided by module bindings. The user can combine modules, but they remain in different name-spaces, with different prefixes, as illustrated by the following code:

```
module X1 = struct let x = 42 end
module X2 = struct let x = 43 end
module Z = struct module X = X1 module X' = X2 end
```

```
module X1 = struct let x = 42 end
module X2 = struct let x = 43 end
module Z : sig
  module X : sig val x : int end
  module X' : sig val x : int end
end
```

The module `Z` is the result of the composition of `X1` and `X2`. The content of `X1` are in a hierarchically different namespace from the content of `X2`, with a different prefix: `X1.x` vs `X2.x`.

Open-hierarchical-composition is provided by functors. The *creation* is definition of the functor, that depends on a *yet-unresolved* module parameter. At the point of functor application, the module is resolved, as illustrated by the following code:

```
(* Creation *)
module F (Y : sig val x : int end) = struct let z = Y.x + 42 end
module X  = struct let x = 1 end
module X' = struct let x = 2 let y = 3 end
(* Resolution *)
module Z = F(X)
module Z' = F(X')
```

```
module F : functor (Y : sig val x : int end) → sig val z : int end
module X : sig val x : int end
module X' : sig val x : int val y : int end

module Z : sig val z : int end
module Z' : sig val z : int end
```

The creation of `F`, `X` and `X'` can be in different files and in any order. The functor can be used several times, creating different instances `Z` and `Z'`. Inside the body of the functor, the content of the parameter is hierarchically separated (prefixed by `Y`). It is not re-exported by default.

Flat composition

There are two main language constructs for *closed-flat-composition*, that we detail below.

Include The first construct is **include**. It takes all the fields of a structure and put them, *at the same level*, inside the current structure. It can be used for modules expressions:

```
module X1 = struct let x1 = 42 end
module X2 = struct let x2 = 43 end
module Z = struct include X1 include X2 let z = x1 + x2 end
```

```
module X1 = sig val x1 : int end
module X2 = sig val x2 : int end
module Z : sig val x1 : int val x2 : int val x : int end
```

It can be used with anonymous modules, as long as they have a structural signature. It is also available for combining signatures. As flat composition merges two name-spaces, there can be shadowing between the two. In OCaml, shadowing *via* an include statement is disallowed and produces a typechecking error. A typical use-case for **include** is to *extend* a module with additional fields.

Open The second construct for flat composition is a variant of the **include** statement where the imported bindings are made available in the current structure but not re-exported: **open**.

```
module X = struct let x = 42 end
module Y = struct open X let y = x + 1 end
```

```
module X : sig val x : int end
module Y : sig val y : int end (* no field x *)
```

While the restricted **open** `P` statement (restricted to paths) is straightforward, the generalized statement **open** `M` Li and Yallop [2017] (where `M` is any module expression) is more subtle: it may create signature avoidance situations by opening abstract type declarations that are not exported and yet appear in the signature.. For instance, the following program fails to typecheck:

```
module X = struct
  open (struct type t = A | B end)
  let x : t = A
end
```

```
Error: The type t/1676 introduced by this open appears in the signature
       Line 3, characters 6-7:
         The value x has no valid type if t/1676 is hidden
```

As for **include**, opening a module can lead to shadowing of fields. While it is again disallowed by default in OCaml, the language also offers a variant **open**`!` that allows shadowing.

Open-flat composition

There is a missing construct for open-flat composition. The closest we can do in OCaml would be to combine a functor with an include statement, as in:

```
module type S = sig val x : int end
module F = functor (Y:S) → struct include Y let f = x + x end
module X = struct let x = 42 let y = 41 end
module Z = F(X)
```

```
module Z : sig val x : int (* from X *) val f : int (* from F *) end
```

Technically, it is an open flat composition, as the content of `X` and `F` are indeed merged in the same namespace at the resolution point. However, this is not satisfactory, as `X` is restricted to the *fixed view* of `S`: `F` can be used only to extend it beyond the field of `S`, other extra-fields are lost (here, the field `y` of `X` is not present in `Z`). Due considerations of compilation-schemes (discussed in Section 1.2.4), this implicit (transparent) ascription at functor call could not be removed. Alternatively, we can use a temporary structure and flat-compose the argument together with the result of functor application, as in:

```
1 module Z = struct
2   open (struct module Temp = struct let x = 42 let y = 41 end end)
3   include Temp
4   include F(Temp)
5 end
```

```
1 module Z : sig val x : int val y : int (* from Temp *) val f : int (* from F *) end
```

Again, this pattern is not fully satisfactory, as it (1) requires to use a dummy name `Temp` and (2) does not compose well: every include of a functor call would require a new temporary structure, with the verbose pattern of line 3.

Include functor A proposal[15] as been made for OCaml (and is already implemented in JaneStreet's work of OCaml) for a new feature called **`include functor`**, aimed a solving this problem. It provides a way to do the *resolution* of open-flat composition, combining a structure and the result of a functor call:

```
module Z = struct
  let x = 42
  let y = 41
  include functor F (* = include F(current structure up to this point) *)
end
```

Using the combined keywords **`include functor`** it allows to flat-compose the result of applying `F` to the current structure being built.

1.5.2 Recursive composition

Sequential composition does not allow the user to define modules that have mutual dependencies. This is a strong limitation of modularity: while the core-language provides mutually recursive types and values, their definitions cannot cross module boundaries. As stated by Russo [2001]: "This limitation compromises modular programming, forcing the programmer to merge conceptually (i.e. architecturally) distinct modules." To overcome this limitation, a notion of *recursive modules* (for closed composition) and *mixin modules* (for open composition) were introduced, allowing for the definition of separate modules that have mutual dependencies. This topic has been heavily researched over the year, by Rossberg and Dreyer [2013]; Im et al. [2011]; Montagu and Rémy [2009]; Dreyer [2007a,b]; Nakata and Garrigue [2006]; Russo [2001]; Crary et al. [1999]; Jaakkola [2020] (to name a few). As is was not at the heart of this thesis, we only briefly present existing features, challenges, or research ideas for future work. We start with an example.

Recursive modules in OCaml OCaml offers a notion of *recursive module binding*, introduced by the keywords **`module rec`**. Each recursive module must be given an explicit signature. The pattern is therefore :

```
module rec X1 : S1 = M1
       and X2 : S2 = M2
       ...
       and Xn : Sn = Mn
```

[15]See RFC at `https://github.com/ccasin/RFCs/blob/include-functor/rfcs/include_functor.md`

The result of typechecking is a *recursive module declaration*, of the form:

```
module rec X1 : S1
       and X2 : S2
       ...
       and Xn : Sn
```

A classical use-case for recursive modules is when a type definition relies on another type definition that is provided by a functor call on the current structure. An example, adapted from Nakata and Garrigue [2006], is the following:

```
1  module rec Tree :
2    sig type t = L | N of TreeSet.t val compare : t → t → int end =
3    struct
4      (* Leafs or sets of sub-trees *)
5      type t = L | N of TreeSet.t
6      let compare (t1: t) (t2 : t) = match (t1, t2) with
7        | (Leaf, Leaf) → 0
8        | (Node(_), Leaf) → 1
9        | (Leaf, Node(_)) → -1
10       | (Node(s1), Node(s2)) → TreeSet.compare s1 s2
11   end
12    and TreeSet : (Set.S with type elt = Tree.t) = Set.Make(Tree)
```

```
module rec Tree : sig type t = L | N of TreeSet.t val compare : t → t → int end
    and TreeSet : sig type t type elt = Tree.t val compare : t → t → int ... end
```

Here, the definition of `Tree.t` at line (3) relies on sets of itself at each node `TreeSet.t`, where sets are provided by the `Set` functor. Both the type definitions and value bindings are mutually recursive between `Tree` and `TreeSet`.

Initialization of recursive structures

```
module rec X : sig val x : unit → int end =
  struct let x () = Y.x + 1 end
and Y : sig val y : int end =
  struct let y = X.x () + 1 end
```

```
Fatal error: exception Undefined_recursive_module
```

Typechecking recursive signatures in the presence of applicative functors

```
module F = functor (Y: sig type t = int end) → struct type u end

module rec X : sig type t = int end =
   struct type t = int end
and Y : sig type u = F(X).t end =
   struct type u = F(X).t end
```

```
Error: Modules do not match: sig type t = X.t end is not included in
       sig type t = int end
     Type declarations do not match:
       type t = X.t
     is not included in
       type t = int
     The type t is not equal to the type int
```

Typechecking recursive structures

1.5.3 Open recursion and mixins modules

1.6 Other features

In this section, we briefly present the remaining features that have not been covered yet. We start with the interaction between the core and module language in Section 1.6.1. In Section 1.6.2, we discuss the features for signature manipulation. In Section 1.6.4, we propose a new feature of *protected types* to provide more flexible abstraction barriers.

1.6.1 Core and module language interactions

The separation between the core and module languages makes modules *second-class* citizens: they cannot be handled directly by core-language expressions. We present some features designed to facilitate interactions between the two language layers.

Local modules The stratification prevents interleaving module and value definitions. OCaml features a *local module* construct that allows to define a module within a value definition, using the **`let module`** keyword:

```
let module M = struct let y = 42 + x end in
M.y + x
```

First-class modules Russo [2000]; Dreyer et al. [2003]; Rossberg et al. [2014]; Rossberg [2006]

Modular explicits/implicits

1ML

1.6.2 Handling signatures

As the module language becomes more and more expressive, the signature language must also be enriched to allow for a fine grained control of interfaces. The two main

- Substitutions `S with type t = int`
- Destructive substitution `S with type t := int`
- Other stuff of Norman Ramsey

1.6.3 Interacting with signature inference

- Using inference **`module type of`** and generation of `.mli` files
- Controlling inference from the code: local defs and local -abs def

1.6.4 Flexible abstraction barriers

Protected types

1.7 Other techniques

OOP Typeclasses Files Scala

1.8 Summary of features

Chapter 2

The ML source system

In this short technical chapter we present our OCaml-like language that we study in the rest of this thesis. In the rest of this thesis, we refer to it as the “source” language. We discuss the syntax, some syntactic mechanisms, and the semantics of the language in this chapter, while we present two type systems for this language in the next two chapter (Chapter 3 and **??**).

Overview In Section 2.1, we present the syntax of the grammar and we discuss technical choices, syntactic sugar and other practical aspects. In Section 2.2 we briefly give an type-erasing semantics.

Paths and Prefixes

$$\begin{array}{rll} P ::= & Q.X & \text{(Module)} \\ \mid & Y & \text{(Module Parameter)} \\ \mid & P(P) & \text{(Applicative application)} \\ Q ::= & A \mid P & \text{(Prefix)} \end{array}$$

Identifiers

$$\begin{array}{rll} I ::= & x & \text{(Variable)} \\ \mid & t & \text{(Type)} \\ \mid & X & \text{(Module)} \\ \mid & T & \text{(Module Type)} \\ \mid & Y & \text{(Functor Parameter)} \end{array}$$

Module Expressions

$$\begin{array}{rll} \mathtt{M} ::= & \mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end} & \text{(Structure)} \\ \mid & P & \text{(Path)} \\ \mid & (Y : \mathtt{S}) \to \mathtt{M} & \text{(Applicative functor)} \\ \mid & () \to \mathtt{M} & \text{(Generative functor)} \\ \mid & P() & \text{(Generative application)} \\ \mid & \mathtt{M}.X & \text{(Anonymous projection)} \\ \mid & (P : \mathtt{S}) & \text{(Ascription)} \end{array}$$

Signatures

$$\begin{array}{rll} \mathtt{S} ::= & \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end} & \text{(Structural signature)} \\ \mid & (= P < \mathtt{S}) & \text{(Transparent signature)} \\ \mid & (Y : \mathtt{S}) \to \mathtt{S} & \text{(Applicative functor)} \\ \mid & () \to \mathtt{S} & \text{(Generative functor)} \\ \mid & Q.T & \text{(Module type)} \end{array}$$

Bindings

$$\begin{array}{rll} \mathtt{B} ::= & \mathtt{let}\ x = \mathtt{e} & \text{(Value)} \\ \mid & \mathtt{type}\ t = \mathtt{u} & \text{(Type)} \\ \mid & \mathtt{module}\ X = \mathtt{M} & \text{(Module)} \\ \mid & \mathtt{module\ type}\ T = \mathtt{S} & \text{(Module type)} \end{array}$$

Declarations

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

Core language expressions

$$\begin{array}{rll} \mathtt{e} ::= & \ldots & \text{(Other constructs)} \\ \mid & Q.x & \text{(Qualified value)} \end{array}$$

Core language types

$$\begin{array}{rll} \mathtt{u} ::= & \ldots & \text{(Other constructs)} \\ \mid & Q.t & \text{(Qualified type)} \end{array}$$

Figure 6: Syntax of our OCaml-like language

2.1 Syntax

The source grammar is given on Figure 6, and discussed below. The language is the combination of three main parts: *paths and identifiers*, *module and signatures* and the *core language*. We start with some minor technical choices:

- **Fonts**: module-related meta-variables use *typewriter uppercase letters*, $\mathtt{M}$, $\mathtt{S}$, etc., while lowercase letters are used for expressions and types of the core language. Lists are written with an overhead bar: $\overline{\mathtt{D}}$ is a list of $\mathtt{D}$. Identifiers I and paths P use a standard (mathematical) font.
- **Structures vs components**: we separate declarations $\mathtt{D}$ from signatures $\mathtt{S}$ and, respectively, bindings $\mathtt{B}$ from module expressions $\mathtt{M}$. In the literature, declarations and signatures (and bindings and modules) are sometimes merged in the same syntactic category.
- **Functor parameters** Y are α-convertible, i.e., can be freely renamed. The other identifiers (X, T, x and t) are not, as they play the role of both internal and external names.

Self-references In order to simplify the treatment of scoping and shadowing, we introduce *self-references*, ranged over by letter A, in both structures and signatures. They are used to refer to the current object; their binding occurrence appears as a subscript to the structure or signature they belong to ($\texttt{struct}_A \dots \texttt{end}, \texttt{sig}_A \dots \texttt{end}$). They are α-convertible. Importantly, they **do not serve as a way to define recursive structures** or do forward references. They only serve as structures and signatures are *dependent* record and fields can refer **previously** defined ones. They are not present in OCaml and should be thought of as being added by a first pass before typing. We explain how they help treat shadowing in Section 2.1.1.

Prefixes In order to have a uniform treatment of access to local and non-local variables, we use *prefixes*, written with the letter Q, to range over either a path P or a self reference A.

Abstract types We use a trick to represent abstract type fields: they are defined as types pointing to themselves, of the form $\texttt{type}\, t = A.t$ where A is the self-reference of the current structure (and often grayed out for readability). This is only a way to removing the need to distinguish between abstract and concrete type fields in the syntax.

Core language We leave abstract a core language of expressions $\texttt{e}$ and types $\texttt{u}$. This language can be thought of as ML, but could very well contain with exotic features. We only extend the language of terms with *qualified values* $Q.x$, that are values bound by the module level, and the language of types with *qualified types* $Q.t$, that are types bound by the module level. These are the only ways for the core level to access the module level.

Functors As in OCaml, we syntactically distinguish applicative and generative functors: generative functors take a special unit argument () as input. The unit argument () is not the same as an empty structure. We find the effect-suggesting syntax of taking a unit argument fitting, as calling a generative functor can indeed produce effects. Our grammar features *unary* applicative functors and *nullary* generative functors. A unary generative functor can be obtained as an applicative functor returning a generative one via the currying notation:

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

While n-ary applicative functors are straightforward, one might wonder if n-ary generative functors require a unit argument between every parameter. Actually, the () acts as a *generative barrier* and can be placed to control the sharing between partial applications:

$$(Y_1 : \texttt{S}_1)(Y_2 : \texttt{S}_2)() \to M$$

is *fully generative* (every instance is new), while

$$(Y_1 : \texttt{S}_1)()(Y_2 : \texttt{S}_2) \to M$$

is generative with regard to the first argument and applicative with regard to the second one.

Projectibility Choosing (1) whether projection is allowed on any module expression or only on a restricted subset, and (2) how the core language can refer to values and types of modules is an important design choice in ML systems, coined *projectibility* by Dreyer et al. [2003]. Contrary to *F-ing* (Rossberg et al. [2014]), but following Leroy [1995] and Russo [2004] (and others), we chose to use a *syntactic* notion of path.

- We allow projection on any module expression, but we restrict functor applications and ascriptions to paths. OCaml does the opposite, mainly to prevent code patterns prone to triggering signature avoidance. Our choice is more general, as we can define a *let* construct for modules using the following syntactic sugar:

$$\mathsf{let}\ X = \texttt{M}\ \mathsf{in}\ \texttt{M}' \triangleq (\texttt{struct}_A\ \texttt{module}\ X = \texttt{M}\quad \texttt{module}\ \mathsf{Res} = \texttt{M}'\ \texttt{end}).\mathsf{Res}$$

Using this construct, we easily get functor application and ascription on arbitrary module expressions:

$$\begin{aligned}\mathtt{M}(\mathtt{M}') &\triangleq \mathsf{let}\ F = \mathtt{M}\ \mathsf{in}\ \mathsf{let}\ X = \mathtt{M}'\ \mathsf{in}\ F(X)\\ \mathtt{M}() &\triangleq \mathsf{let}\ G = \mathtt{M}\ \mathsf{in}\ G()\\ (\mathtt{M} : \mathtt{S}) &\triangleq \mathsf{let}\ X = \mathtt{M}\ \mathsf{in}\ (X : \mathtt{S})\end{aligned}$$

The reverse encoding of projection as an anonymous functor call requires an explicit signature annotation on the argument and thus cannot be seen as syntactic sugar.

- A qualified access inside a generative functor application, of the form $G().t$, is syntactically ill-formed, as paths do not contain the unit argument $()$. By contrast, a qualified access inside an applicative functor application $F(X).t$ is permitted.
- A qualified access inside a module type, which would be of the form $Q.T.t$, is syntactically ill-formed, as paths do not contain module type identifiers T.
- We only provide opaque ascription in module expressions, as concrete ascription is given by the following syntactic sugar: $(P < \mathtt{S}) \triangleq (P : (= P < \mathtt{S}))$

As both path and module expressions feature a projection dot, the grammar is slightly ambiguous. However, this is not a problem as we see paths as a subset of module expressions. In particular, we only consider the projection dot of module expressions in the typing rules.

Transparent ascription Transparent ascription as a module expression, as available in SML, can be encoded using either a normal ascription with a transparent signature, or as a anonymous functor call:

$$\begin{aligned} &(\mathtt{M} < \mathtt{S}) \triangleq \mathsf{let}\ X = \mathtt{M}\ \mathsf{in}\ (X : (= X < \mathtt{S}))\\ \text{or}\quad &(\mathtt{M} < \mathtt{S}) \triangleq \mathsf{let}\ F = ((Y : \mathtt{S}) \to Y)\ \mathsf{in}\ F(M)\end{aligned}$$

Program Formally, we can define the top-level of a file as the body of generative functor

2.1.1 Name spaces

Scopes The structure of paths define *scopes*: names are accessible only in some parts of the code. The mechanism is the same in module expressions and in signatures; we use the latter for illustration. The two constructs that introduce a new scope are generative functors and module types :

$$() \to \mathtt{S} \qquad\qquad \mathtt{module\ type}\ T = \mathtt{S}$$

The names defined in $\mathtt{S}$ are not accessible outside of $\mathtt{S}$, as there is no path with a unit argument and no path with a projection out of a module type name. By contrast, module bindings and applicative functors introduce sub-scopes that are still part of the same enclosing scope. If we consider:

$$\mathtt{module}\,X : \mathtt{sig}_A\ \mathtt{type}\ t = \ldots\ \mathtt{end} \qquad\qquad (Y : \mathtt{S}_a) \to \mathtt{sig}_A\ \mathtt{type}\ t = \ldots\ \mathtt{end}$$

The name t is accessible outside of the signature, via a qualified access with a path.

2.1.2 Shadowing

There are two main variants of shadowing: *direct* and *local*. Direct shadowing occurs when to bindings are made with the same name in the same structure or same structural signature. OCaml allows direct shadowing of value fields in structures, which is used by some coding patterns, but disallows direct shadowing of type fields[1]. In both type system of Chapter 3 and Chapter 4, we disallow direct shadowing. Local shadowing occurs when a bindings made inside a structure uses the same name as a binding coming from an enclosing structure. In our language, we use self-references to disambiguate. Let us consider the following example of OCaml code (on the left) and our language (on the right):

```
1 module M = struct
2   type t = int
3   module X = struct
4     type t = bool
5     val x : t = true
6   end
7   type u = t * X.t
8 end
```

```
1 module M = struct_A
2   type t = int
3   module X = struct_B
4     type t = bool
5     let x : B.t = true
6   end
7   type u = A.t × A.X.t
8 end
```

In the OCaml code, at (line 5) the type `t` refers to the local definition at (line 4), which as shadowed the declaration at (line 2). This shadowing ends when we exit the submodule, at (line 6), and both declarations become accessible at the same time. Using self-references, we can distinguish $A.t$ and $B.t$: they are considered as different identifiers. In practice, OCaml generates fresh identifiers attached to every name for disambiguation. This serves more-or-less the same purpose as self-references, but in an effectfull way that would be hard to model formally. By contrast, self-references act as a normal name binder attached to the structure (or signature), they can be freely renamed (α-convertibility). We believe that overall, they simplify the formal presentation at the cost of being more verbose.

2.2 Semantics

ML modules have (surprisingly) very simple semantics. Basically, at runtime, all types are removed – the semantic is *type-erasing* – and modules are elaborated away into core-language constructs (using functions and records). The type erasing semantics ensures that safety guarantees of the type system are obtained at no cost at runtime.

Core language We assume that the core language supports simple untyped functions, records and unit argument. Formally, we suppose that it extends the following calculus:

$$e ::= x \mid () \mid \lambda x.e \mid e\, e \mid \left\{\overline{\ell_i = x_i}\right\} \mid e.\ell \mid e \mathbin{@} e \mid \dots \qquad \text{(Terms)}$$

$$v ::= x \mid () \mid \lambda x.e \mid \left\{\overline{\ell_i = v_i}\right\} \mid \dots \qquad \text{(Values)}$$

[1] **include** and **open** have a special treatment: the declarations or bindings imported by them can be shadowed, even type fields, but they cannot be used to shadow neither a type or value field.

We denote record concatenation with the symbol @. The small-steps semantic judgment, written $e \rightsquigarrow e'$, is given by the following rules:

$$(\lambda x.e)\, v \rightsquigarrow e[x \mapsto v] \qquad \frac{e_1 \rightsquigarrow e_1'}{e_1\, e_2 \rightsquigarrow e_1'\, e_2} \qquad \frac{e_2 \rightsquigarrow e_2'}{v_1\, e_2 \rightsquigarrow v_1\, e_2'}$$

$$\{\dots, \ell = v, \dots\}\,.\ell \rightsquigarrow v \qquad \frac{e_i \rightsquigarrow e_i'}{\{\dots, \ell_i = e_i, \dots\} \rightsquigarrow \{\dots, \ell_i = e_i', \dots\}}$$

$$\frac{\overline{\ell_1} \# \overline{\ell_2}}{\left\{\overline{\ell_1 = v_1}\right\} @ \left\{\overline{\ell_2 = v_2}\right\} \rightsquigarrow \left\{\overline{\ell_1 = v_1}, \overline{\ell_2 = v_2}\right\}} \qquad \frac{e_1 \rightsquigarrow e_1'}{e_1 @ e_2 \rightsquigarrow e_1' @ e_2} \qquad \frac{e_2 \rightsquigarrow e_2'}{v_1 @ e_2 \rightsquigarrow v_1 @ e_2'}$$

We denote disjointness of the lists of labels with #.

Type erasure We can then define an erasing operation $\lfloor\cdot\rfloor$ that translate modules and paths into this untyped calculus. We start with two technical details:

1. We assume a collection of labels indexed by identifiers, i.e., ℓ_I is the unique label associated with the identifier I
2. We assume that the variables of the core language can be extended to contain the variables of the form $A.I$ for any self-reference and identifier and Y for functor parameters.

Using those, we define the erasing translation. Elaboration of paths is defined by the following cases:

$$\lfloor A.X \rfloor = A.X \qquad \lfloor P.X \rfloor = \lfloor P \rfloor.\ell_X \qquad \lfloor Y \rfloor = Y \qquad \lfloor P(P') \rfloor = \lfloor P \rfloor\,(\lfloor P' \rfloor)$$

Then the erasing of module expressions is given by the following rules:

$$\lfloor (P : \mathtt{S}) \rfloor = \lfloor P \rfloor \qquad \lfloor \mathtt{M}.X \rfloor = \lfloor \mathtt{M} \rfloor.\ell_X \qquad \lfloor \mathtt{struct}_A\, \overline{\mathtt{D}}\, \mathtt{end} \rfloor = \overline{\lfloor \mathtt{D} \rfloor_A} \qquad \lfloor () \to \mathtt{M} \rfloor = \lambda x.\lfloor \mathtt{M} \rfloor$$

$$\lfloor (Y : \mathtt{S}) \to \mathtt{M} \rfloor = \lambda x_Y.\lfloor \mathtt{M} \rfloor \qquad \lfloor P() \rfloor = \lfloor P \rfloor\,()$$

Finally, the erasing of modules relies on the erasing of bindings, which shows that structures are "dependent" records:

$$\lfloor \mathtt{let}\, x = \mathtt{e}, \overline{\mathtt{D}} \rfloor_A = (\lambda x'.\, \{\ell_x = x'\} @ \lfloor \overline{\mathtt{D}} \rfloor_A[A.x \mapsto x'])\, \lfloor e \rfloor$$

$$\lfloor \mathtt{module}\, X = \mathtt{M}, \overline{\mathtt{D}} \rfloor_A = (\lambda x.\, \{\ell_X = x\} @ \lfloor \overline{\mathtt{D}} \rfloor_A[A.X \mapsto x])\, \lfloor \mathtt{M} \rfloor$$

$$\lfloor \mathtt{type}\, t = \mathtt{u}, \overline{\mathtt{D}} \rfloor_A = \lfloor \overline{\mathtt{D}} \rfloor_A \qquad \lfloor \mathtt{module\ type}\ T = \mathtt{S}, \overline{\mathtt{D}} \rfloor_A = \lfloor \overline{\mathtt{D}} \rfloor_A$$

And that's it. The semantics of modules is very simple. ML modules are an interesting case of programming language research where the type system is quite complex, with involved soundness proofs, while the semantic model is mostly trivial. In the literature, the underlying semantic is sometimes not even mentioned.

Chapter 3

M^{ω}

In this chapter, we present M^{ω}, a type system for ML modules that produces inferred signatures in an ML-like syntax extended with F^{ω} type binders $(\exists, \forall, \lambda)$. The goal of this system is to provide a specification and an intuition for the core mechanisms informally presented in the previous chapter. The system is proven sound by a full elaboration in F^{ω}.

Contributions M^{ω} is strongly inspired by previous works, notably Rossberg et al. [2014] and Russo [2004]. The core contributions of this chapter are the introduction of *transparent existential types* for the soundness proof of the skolemization operation, the *anchoring algorithm* for reconstruction of source signatures from M^{ω} signatures and the treatment of *abstract signatures* via predicative kind polymorphism.

Overview We present the type system in Section 3.1: the elaboration of ML signatures into M^{ω} signatures is presented in Section 3.1.2, along with the key mechanisms of *extrusion* and *skolemization*. Subtyping is presented in Section 3.1.3. The typing of module expressions is detailed in Section 3.1.4.

In Section 3.2, we explain how the right granularity of applicativity can be piggy-backed on the type system by introducing extra *identity* abstract type fields.

In Section 3.3, we focus on the reconstruction of source signatures from M^{ω} signatures, a process call *anchoring* that acts as the inverse of the signature elaboration. We first explicit the expressiveness gaps of the source signature syntax, then we present an algorithm that implements anchoring. Finally, we state and prove the properties of this algorithm.

In Section 3.4, we prove the soundness of M^{ω} via a full elaboration into F^{ω}. This proof relies on the new mechanism of *transparent existential types* (Section 3.4.4), a weaker form of existential types that supports skolemization. We show that extending F^{ω} with existential types can be done *internally*, i.e., we define transparent existential types as a library of F^{ω} (Section 3.4.5). Finally, the actual elaboration is given an proven sound with respect to F^{ω} typing rules (Sections 3.4.6 and 3.4.7).

M^ω Kinds

$\kappa ::= \star \mid \kappa \rightarrow \kappa$

M^ω Types

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

Environment

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

Opacity

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

M^ω signature

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

M^ω declaration

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

Figure 7: Syntax of M^ω signatures.

3.1 The M^ω type system

3.1.1 Overview and technical details

In Figure 7, we introduce the syntax for M^ω-signatures $\mathcal{C}$ and M^ω-declarations $\mathcal{D}$. By convention, we use curvy capitals ($\mathcal{C}$, $\mathcal{D}$, ...) for M^ω-objects.

Types The grammar of M^ω types is the same as the grammar of source types, except that qualified types $Q.t$ are replaced by abstract types α or applied abstract types $\tau(\overline{\tau})$, where α range over a collection of abstract type variables that are distinct from the type variables of the source type language. We use the base kind $\star$ for the kind of first-order types, and $\kappa \rightarrow \kappa$ for higher order kinds. In the examples, we sometimes use φ instead of α to denote an higher-order type variable. The language is explicitly kinded, as is F^ω. However we leave kinds mostly implicit in this section for the sake of readability—we display them for the soundness proof in Section 3.4.

Quantifiers positions M^ω-signatures $\mathcal{C}$ use F^ω binders: the universal binder $\forall$ for functor parameters, the existential binder $\exists$ for the body of generative functors and the (type-level) lambda binder λ for module types. We annotate existential quantifiers with an opacity flag $\Diamond$ to indicate generativity (using the opaque flag $\blacktriangledown$) or applicativity (using the transparent flag $\triangledown$). This notion is unrelated to transparent signatures. Transparent existentials $\exists^{\triangledown}\overline{\alpha}.\mathcal{C}$do not appear directly in the grammar of Figure 7 but in the typing judgment for module expressions, which uses existentially quantified signatures of either form $\exists^{\triangledown}\overline{\alpha}.\mathcal{C}$ or $\exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}$. Module types $\lambda\overline{\alpha}.\mathcal{C}$ are parametric in each type variable α. A parametric signature $\lambda\overline{\alpha}.\mathcal{C}$ may later be transformed into a universal signature $\forall\overline{\alpha}.\mathcal{C}$, an existential signature $\exists^{\Diamond}\overline{\alpha}.\mathcal{C}$, or just a concrete signature $\mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]$. Crucially, module declarations $\mathsf{module}\ X : \mathcal{C}$ foes not allow quantifiers in front of $\mathcal{C}$ – they must have been extruded away at the top of the signature – nor inside the codomain of applicative functors $\forall\overline{\alpha}.\mathcal{C}_a \rightarrow \mathcal{C}$ – they must have been skolemized away.

Type equivalence We consider M^ω types up to $\alpha\beta$-equivalence. α-equivalence is standard: bounded type variables and functor parameters can be freely renamed, but other identifiers cannot. β-equivalence is defined as the symmetric, reflexive, and transitive and congruent closure of β-reduction, which is defined by the single following rule for type applications :

$$(\lambda\alpha.\sigma)\tau \leadsto^\beta \sigma[\alpha \mapsto \tau]$$

Environments and wellformedness Typing environments contain three types of bindings: an abstract type variable $(\alpha : \kappa)$, a functor argument $Y : \mathcal{C}$, or a declaration $A.\mathcal{D}$. All bindings in Γ are unique (there is no shadowing). Technically, this is achieved by defining mutually recursive *well-formedness* predicates over environments $\vdash \Gamma$, signatures $\Gamma \vdash \mathcal{C}$, and declarations $\Gamma \vdash \mathcal{D}$, along with a *well-kindness* predicate for types $\Gamma \vdash \tau : \kappa$. The rules are standard and given in Figure 8. We write $\cdot \notin \Gamma$ as a shortcut for $\cdot \notin \mathsf{dom}(\Gamma)$. The only subtlety comes from the wellformedness of structural signatures:

$$\frac{\mathsf{disjoint}(\overline{\mathcal{D}}) \qquad \Gamma \vdash \overline{\mathcal{D}}}{\Gamma \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}}$$

Here, we use an helper predicate $\mathsf{disjoint}(\overline{\mathcal{D}})$ to ensure that all field identifiers present in the list $\overline{\mathcal{D}}$ are pair-wise distinct.

Wellformedness conditions As a simplifying convention for the rest of this chapter, we consider wellformedness of the environment as a precondition to all rules. Alternatively, we could have added wellformedness preconditions sparingly, only to the rules that are *leaves* of the derivation, i.e., that do not have another typing or subtyping as a premise. Then, we would have shown that typing and subtyping always implies the wellformedness of the environment: a rule is either a leaf and has an explicit wellformedness premise, or is not a leaf and the wellformedness of the environment is implied by another premise by induction hypothesis. Here, we do not see any benefit in this latter approach and we use the simplifying convention of having wellformedness everywhere.

Judgments

- $\Gamma \vdash \mathtt{M} : \exists^\Diamond\overline{\alpha}.\mathcal{C}$ and $\Gamma \vdash \mathtt{D} : \exists^\Diamond\overline{\alpha}.\mathcal{D}$: typechecking of modules and bindings, defined in Figure 13 and discussed in Section 3.1.4.
- $\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}$ and $\Gamma \vdash \mathtt{D} : \lambda\overline{\alpha}.\mathcal{C}$: typechecking of signatures and declarations, defined in Figure 9 and discussed in Section 3.1.2.
- $\Gamma \vdash \mathtt{u} : \tau$ and $\Gamma \vdash \mathtt{e} : \tau$ — elaboration of core-language types and type-checking of core-language expressions
- $\Gamma \vdash \mathcal{C} < \mathcal{C}'$ and $\Gamma \vdash \mathcal{D} < \mathcal{D}'$: subtyping of signatures and declarations, defined in Figure 11 and discussed in Section 3.1.3.

3.1.2 Signatures type-checking

The key concepts of M$^\omega$ can be illustrated with the typechecking of signatures $\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}$ (and declarations), which translates a source signature $\mathtt{S}$ into its M$^\omega$ counterpart $\lambda\overline{\alpha}.\mathcal{C}$, making the set of abstract type $\overline{\alpha}$ explicit. The typechecking of signature also acts as an *elaboration*: source signatures are checked an translated into M$^\omega$-signatures. We refer to both *typechecking* and *elaboration*, using the former to emphasize on the wellformedness check and the latter to emphasize on the translation into the M$^\omega$-signature language. The full set of rules is given in Figure 9 and discussed below.

$$\vdash \varnothing \qquad \frac{\vdash \Gamma \qquad \alpha \notin \Gamma}{\vdash \Gamma, (\alpha : \kappa)} \qquad \frac{\vdash \Gamma \qquad Y \notin \Gamma \qquad \Gamma \vdash \mathcal{C}}{\vdash \Gamma, Y : \mathcal{C}} \qquad \frac{\vdash \Gamma \qquad A.x \notin \Gamma \qquad \Gamma \vdash \tau : \star}{\vdash \Gamma, A.\mathsf{val}\ x : \tau}$$

$$\frac{\vdash \Gamma \qquad A.t \notin \Gamma \qquad \Gamma \vdash \tau : \kappa}{\vdash \Gamma, A.\mathsf{type}\ t = \tau} \qquad \frac{\vdash \Gamma \qquad A.X \notin \Gamma \qquad \Gamma \vdash \mathcal{C}}{\vdash \Gamma, A.\mathsf{module}\ X : \mathcal{C}}$$

$$\frac{\vdash \Gamma \qquad A.T \notin \Gamma \qquad \overline{\alpha} \notin \Gamma \qquad \Gamma, \overline{(\alpha : \kappa)} \vdash \mathcal{C}}{\vdash \Gamma, A.\mathsf{module\ type}\ T = \lambda \overline{\alpha}.\mathcal{C}}$$

(a) Wellformdness of environments

$$\frac{\mathsf{disjoint\_identifiers}(\overline{\mathcal{D}}) \qquad \Gamma \vdash \overline{\mathcal{D}}}{\Gamma \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}} \qquad \frac{\overline{\alpha} \notin \Gamma \qquad \Gamma, \overline{(\alpha : \kappa)} \vdash \mathcal{C}_a \qquad \Gamma, \overline{(\alpha : \kappa)} \vdash \mathcal{C}}{\Gamma \vdash \forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C}}$$

$$\frac{\overline{\alpha} \notin \Gamma \qquad \Gamma, \overline{(\alpha : \kappa)} \vdash \mathcal{C}}{\Gamma \vdash () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}}$$

(b) Wellformdness of signatures

$$\frac{\Gamma \vdash \tau : \star}{\Gamma \vdash \mathsf{val}\ x : \tau} \qquad \frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \mathsf{type}\ t = \tau} \qquad \frac{\Gamma \vdash \mathcal{C}}{\Gamma \vdash \mathsf{module}\ X : \mathcal{C}} \qquad \frac{\overline{\alpha} \notin \Gamma \qquad \Gamma, \overline{(\alpha : \kappa)} \vdash \mathcal{C}}{\Gamma \vdash \mathsf{module\ type}\ T = \mathcal{C}}$$

(c) Wellkindness of declarations

$$\frac{(\alpha : \kappa) \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : \kappa \to \kappa' \qquad \Gamma \vdash \sigma : \kappa}{\Gamma \vdash (\tau\, \sigma) : \kappa'} \qquad \frac{\alpha \notin \Gamma \qquad \Gamma, (\alpha : \kappa) \vdash \tau : \kappa'}{\Gamma \vdash \lambda(\alpha : \kappa).\tau : \kappa \to \kappa'}$$

(d) Well-kindness of types

Figure 8: Wellformedness rules for environment, signatures and types.

Declarations

An abstract type declaration introduces an abstract type variable α that is λ-bound:

M-Typ-Decl-TypeAbs

$$\Gamma \vdash_A (\mathtt{type}\, t = A.t) : \lambda \alpha.(\mathsf{type}\ t = \alpha)$$

Since the name t is also be accessible in the following declarations, the λ-binder for α must be *lifted* to enclose the whole region where t, hence α, is accessible. This lifting is performed in two places:

M-Typ-Decl-Seq

$$\frac{\Gamma \vdash_A \mathtt{D}_1 : \lambda \overline{\alpha}_1.\mathcal{D}_1 \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D}_1 \vdash_A \overline{\mathtt{D}} : \lambda \overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \vdash_A \mathtt{D}_1, \overline{\mathtt{D}} : \lambda \overline{\alpha}_1\, \overline{\alpha}.\ \mathcal{D}_1, \overline{\mathcal{D}}}$$

M-Typ-Decl-Mod

$$\frac{\Gamma \vdash \mathtt{S} : \lambda \overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module}\, X : \mathtt{S}) : \lambda \overline{\alpha}.(\mathsf{module}\ X : \mathcal{C})}$$

First, when merging a list of declarations in Rule M-Typ-Decl-Seq: the two sets of abstract types $\overline{\alpha}$ and $\overline{\alpha}_1$ are merged together in front of the list of declarations. Second, lifting also occurs when typing a module declaration (Rule M-Typ-Decl-Mod) where the set of abstract types $\overline{\alpha}$ introduced in the elaboration of $\mathtt{S}$ are lifted to the outside of the declaration. In a nutshell, abstract types introduced by submodules are not bound at the declaration of the submodule, but lifted to the enclosing signature.

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

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

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

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

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

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

(a) Signature typing rules.

M-Typ-Decl-Val
$$\dfrac{\Gamma \vdash \mathtt{u} : \tau}{\Gamma \vdash_A (\mathtt{val}\, x : \mathtt{u}) : (\mathsf{val}\ x : \tau)}$$

M-Typ-Decl-Type
$$\dfrac{\Gamma \vdash \mathtt{u} : \tau}{\Gamma \vdash_A (\mathtt{type}\, t = \mathtt{u}) : (\mathsf{type}\ t = \tau)}$$

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

M-Typ-Decl-Mod
$$\dfrac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module}\, X : \mathtt{S}) : \lambda\overline{\alpha}.(\mathsf{module}\ X : \mathcal{C})}$$

M-Typ-Decl-ModType
$$\dfrac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module\ type}\ T = \mathtt{S}) : (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C})}$$

M-Typ-Decl-Empty
$$\Gamma \vdash_A \varnothing : \varnothing$$

M-Typ-Decl-Seq
$$\dfrac{\Gamma \vdash_A \mathtt{D}_1 : \lambda\overline{\alpha}_1.\mathcal{D}_1 \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D}_1 \vdash_A \overline{\mathtt{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \vdash_A \mathtt{D}_1, \overline{\mathtt{D}} : \lambda\overline{\alpha}_1\,\overline{\alpha}.\ \mathcal{D}_1, \overline{\mathcal{D}}}$$

(b) Declaration typing rules.

M-Typ-Type-Path
$$\dfrac{\Gamma \vdash P : \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{type}\ t = \tau \in \overline{\mathcal{D}}}{\Gamma \vdash P.t : \tau}$$

M-Typ-Type-Local
$$\dfrac{A.(t :\ \mathsf{type}\ \tau) \in \Gamma}{\Gamma \vdash A.t : \tau}$$

(c) Extension to the core-language typing rules

Figure 9: Signature and declaration typing rules.

By contrast, when typing module-type declarations, the binder is not lifted, as the module-type declaration defines its own scope. The declarations inside $\mathcal{C}$ are not accessible in the rest of the enclosing signature.

M-Typ-Decl-ModType
$$\dfrac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module\ type}\ T = \mathtt{S}) : (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C})}$$

Typing declaration of value or type fields relies on the elaboration judgment of core-language types into F^ω, $\Gamma \vdash \mathtt{u} : \tau$ which is left abstract.

M-Typ-Decl-Val
$$\dfrac{\Gamma \vdash \mathtt{u} : \tau}{\Gamma \vdash_A (\mathtt{val}\, x : \mathtt{u}) : (\mathsf{val}\ x : \tau)}$$

M-Typ-Decl-Type
$$\dfrac{\Gamma \vdash \mathtt{u} : \tau}{\Gamma \vdash_A (\mathtt{type}\, t = \mathtt{u}) : (\mathsf{type}\ t = \tau)}$$

Still, we extend it with rules to translate qualified types into M^ω types:

M-Typ-Type-Path
$$\dfrac{\Gamma \vdash P : \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{type}\ t = \tau \in \overline{\mathcal{D}}}{\Gamma \vdash P.t : \tau}$$

M-Typ-Type-Local
$$\dfrac{A.(t :\ \mathsf{type}\ \tau) \in \Gamma}{\Gamma \vdash A.t : \tau}$$

```
sig_A
  type t = A.t
  module type T =
    sig_C type t = C.t end
  module X : A.T
  module G : () → A.T
  module F : (Y : A.T) → A.T

  type v = A.t × A.X.t
  type w = A.F(A.X).t
end
```

```
λα_1, α_2, φ.sig
  type t = α_1
  module type T =
    λα.sig type t = α end
  module X : sig type t = α_2 end
  module G : () → ∃^▼α. sig type t = α end
  module F : ∀β.sig type t = β end →
      sig type t = φ(β) end
  type v = α_1 × α_2
  type w = φ(α_u)
end
```

Figure 10: On the left-hand side, we consider a source signature. The corresponding signature resulting from typechecking in M^{ω} is given on the right-hand side. The signature mostly have type declarations because it displays the interesting mechanisms of the type system, but in real use-cases, signatures have mostly value declarations. Both the abstract type variables α_1, α_2 and φ are extruded at the top-level of the signature, despite the fact that they come from different levels of the signature. The intermediate definition of the module type `T` is inlined. The applicative functor introduces a higher-order type φ, while the generative introduces only a base kind type variable that is not extruded.

Signatures

Structural signatures The typing of declarations is injected in the typing of signatures by the rule for structural signatures:

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

This is the last place where extrusion happens (see M-Typ-Decl-Type, M-Typ-Decl-Mod, M-Typ-Decl-Seq for the other rules): the abstract types $\overline{\alpha}$ introduced by the declarations are lifted in front of the whole signature.

Module-types Module-type definitions are inlined by the following two rules:

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

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

Therefore M^{ω} signatures do not have a counterpart for module-type paths $Q.T$ that occur in source signatures.

Transparent signatures The rule M-Typ-Sig-Trans displays an interesting mechanism:

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

The source signature `S` is first elaborated into an M^{ω}-signature $\lambda\overline{\alpha}.\mathcal{C}'$. The M^{ω}-signature $\mathcal{C}$ of the path P is then obtained by module typing. From there, we use subtyping to compare the two signatures. But there is a catch: $\lambda\overline{\alpha}.\mathcal{C}$ is parametric, whereas $\mathcal{C}'$ is not. Therefore, we need to first find an *instantiation* of the abstract types $\overline{\alpha}$ by some concrete types $\overline{\tau}$. Finding such instantiation is non-trivial, especially in the presence of higher-order types. It is discussed in more details along with subtyping in Section 3.1.3. Once the instantiation has been found, the instantiated signature $\mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]$ can be compared against $\mathcal{C}'$. The result of the elaboration is the instantiation $\mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]$. Notably, no new abstract type is introduced. It fits the intuition that transparent signatures share all their type fields with another pre-existing module, and therefore all abstract type variables have always already been introduced.

Generative functors Here, the abstract types $\overline{\alpha}$ introduced by the codomain of the functor are not extruded in front of the whole signature. Instead, they are left as an opaque existential signature $\exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}$ for the functor codomain. Indeed, every instantiation of the functor will generate *new* (incompatible) abstract types $\overline{\alpha}$, as required for generativity.

M-Typ-Sig-GenFct

$$\dfrac{\Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash () \to \mathsf{S} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}$$

Applicative functors By contrast, applicative functors should not be assigned a signature of the form $\forall\overline{\alpha}.\mathcal{C} \to \exists^{\blacktriangledown}\overline{\beta}.\mathcal{C}'$ where all applications would produce new abstract types, nor $\lambda\beta.\forall\overline{\alpha}.\mathcal{C} \to \mathcal{C}'$ where all applications would share the same types regardless of their argument. Instead, the following rule follows the solution of Biswas [1995] and reused in Russo [2004] and *F-ing* (Rossberg et al. [2014]):

M-Typ-Sig-AppFct

$$\dfrac{\Gamma \vdash \mathsf{S}_a : \lambda\overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a \vdash \mathsf{S} : \lambda\overline{\beta}.\mathcal{C}}{\Gamma \vdash (Y : \mathsf{S}_a) \to \mathsf{S} : \lambda\overline{\beta'}.\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]}$$

That is, we use higher-order abstract types β' and apply each to the universally quantified variables $\overline{\alpha}$ to capture the fact that the each abstract type is *some type function* of the arguments. This gives a signature of the form $\lambda\overline{\beta'}.\forall\overline{\alpha}.\mathcal{C} \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]$. We call this extrusion out of an arrow type and out of an universal binder *skolemization*. Here, it is merely a manipulation of types and signatures that does not pose a challenge for soundness. In a sense, we can manipulate types in any way as long as they remain wellformed. One key technical aspect of M^ω is to justify a similar skolemization but for the type of module expressions, as described in Section 3.4.

Here and until Section 3.2, we have a type-only applicativity, like Moscow ML: the abstract types $\overline{\beta'}$ of the codomain depend on the abstract types $\overline{\alpha}$ of the parameter, not on the whole module. We then change it to the module-level equality of OCaml in Section 3.2.

3.1.3 Subtyping

The subtyping judgment $\Gamma \vdash \mathcal{C} < \mathcal{C}'$ and its helper judgment $\Gamma \vdash \mathcal{D} < \mathcal{D}'$ check that a signature $\mathcal{C}$ is *more restrictive* than a signature $\mathcal{C}'$. In practice, this boils down to $\mathcal{C}$ having more fields and introducing fewer abstract types than $\mathcal{C}'$. Intuitively, it can be understood as "*there are less modules with the signature $\mathcal{C}$ than with the signature $\mathcal{C}'$*". Overall, subtyping in M^ω is a combination of three orthogonal mechanisms:

- structural subtyping between record types, including deletion and reordering of fields.
- subtyping between function types: covariant on their codomains, contravariant on their domains
- subtyping between universal types and between existential types via instantiation. Originally introduced as the (sub) rule for "type containment" in Mitchell [1988], or sometimes called *subtyping à la F^η*, it allows to compare types with different quantifiers by partially instantiating the quantifiers.

The full set of subtyping rules is given in Figure 11 and discussed below.

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

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

$$\textsc{M-Sub-Decl-Val}\quad \Gamma \vdash (\mathsf{val}\ x : \tau) < (\mathsf{val}\ x : \tau) \qquad \textsc{M-Sub-Decl-Type}\quad \Gamma \vdash (\mathsf{type}\ t = \tau) < (\mathsf{type}\ t = \tau)$$

$$\textsc{M-Sub-Decl-Mod}\quad \dfrac{\Gamma \vdash \mathcal{C} < \mathcal{C}'}{\Gamma \vdash (\mathsf{module}\ X : \mathcal{C}) < (\mathsf{module}\ X : \mathcal{C}')} \qquad \textsc{M-Sub-Decl-ModType}\quad \dfrac{\Gamma, \overline{\alpha} \vdash \mathcal{C} < \mathcal{C}' \qquad \Gamma, \overline{\alpha} \vdash \mathcal{C}' < \mathcal{C}}{\Gamma \vdash (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}) < (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}')}$$

(a) Subtyping rules for signatures

$$\textsc{M-Sub-Decl-Val}\quad \Gamma \vdash (\mathsf{val}\ x : \tau) < (\mathsf{val}\ x : \tau) \qquad \textsc{M-Sub-Decl-Type}\quad \Gamma \vdash (\mathsf{type}\ t = \tau) < (\mathsf{type}\ t = \tau)$$

$$\textsc{M-Sub-Decl-Mod}\quad \dfrac{\Gamma \vdash \mathcal{C} < \mathcal{C}'}{\Gamma \vdash (\mathsf{module}\ X : \mathcal{C}) < (\mathsf{module}\ X : \mathcal{C}')} \qquad \textsc{M-Sub-Decl-ModType}\quad \dfrac{\Gamma, \overline{\alpha} \vdash \mathcal{C} < \mathcal{C}' \qquad \Gamma, \overline{\alpha} \vdash \mathcal{C}' < \mathcal{C}}{\Gamma \vdash (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}) < (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}')}$$

(b) Subtyping rules for declarations

Figure 11: M^{ω}– Subtyping rules (signatures and declarations)

Structural signatures The key rule is the comparison of two structural signatures:

$$\textsc{M-Sub-Sig-Struct}\quad \dfrac{\overline{\mathcal{D}}_0 \subseteq \overline{\mathcal{D}} \qquad \Gamma \vdash \overline{\mathcal{D}}_0 < \overline{\mathcal{D}}'}{\Gamma \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} < \mathsf{sig}\ \overline{\mathcal{D}}'\ \mathsf{end}}$$

A subset $\overline{\mathcal{D}}_0$ of the fields $\overline{\mathcal{D}}$ of the left-hand side signature is matched and subtyped against the full set of fields of the right-hand side signature $\overline{\mathcal{D}}'$. This subset $\overline{\mathcal{D}}_0$ can be reordered and does not have to contain all the fields of the right-hand side signature. Algorithmically, it is easy to find $\overline{\mathcal{D}}_0$ from the combination of $\overline{\mathcal{D}}$ and $\overline{\mathcal{D}}'$: for each declaration of $\overline{\mathcal{D}}'$, we must find a corresponding declaration in $\overline{\mathcal{D}}$ with the same identifier, and there exists at most one (identifiers are pairwise distinct in a structural signature).

Declarations Subtyping between declarations boils down to a matching of definitions, as we assume no subtyping on the core language. The identifiers always have to be the same on both sides. For values and type declarations, we have:

$$\textsc{M-Sub-Decl-Val}\quad \Gamma \vdash (\mathsf{val}\ x : \tau) < (\mathsf{val}\ x : \tau) \qquad \textsc{M-Sub-Decl-Type}\quad \Gamma \vdash (\mathsf{type}\ t = \tau) < (\mathsf{type}\ t = \tau)$$

Subtyping propagates down to submodules, which gives the following rule:

$$\textsc{M-Sub-Decl-Mod}\quad \dfrac{\Gamma \vdash \mathcal{C} < \mathcal{C}'}{\Gamma \vdash (\mathsf{module}\ X : \mathcal{C}) < (\mathsf{module}\ X : \mathcal{C}')}$$

Subtyping between module-types only allows for a different ordering of fields between the two definitions, which we enforce by requiring subtyping in both directions:

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

We could also have made subtyping between module types more restricted, requiring the same signature (up to M$^\omega$ signature equivalence defined in Section 3.1.1) on both sides. OCaml uses a slightly more restricted criterion, requiring code-free subtyping between the two declarations. It is discussed in more detail in Section 4.3.6. This more restricted criterion makes sens when the names of the fields are kept in the resulting signature: changing the definition of `type` $t =$ `u` when there are occurrences of t remaining in the signature should only be allowed if the change of definition is code-free. In M$^\omega$, definitions are inlined, so changing the definition does not affect the rest of the signature.

Generative functors The rule for generative functors M-Sub-Sig-GenFct shows the mechanism of subtyping by *instantiation* of quantifiers:

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

It can be read as a two-step process, where we first check the subtyping between $\exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}$ and $\exists^{\blacktriangledown}\overline{\alpha}'.\mathcal{C}'$, which in turn amounts to finding an instantiation of $\overline{\alpha}'$ by some type $\overline{\tau}'$ (that might use the $\overline{\alpha}$). While this is one of the standard ways of specifying subtyping for existential types, as done in Mitchell [1988]; Russo [2004]; Rossberg et al. [2014], it quite permissive, as it does not require both sets of variables to be the same. However, it is quite algorithmically challenging in the presence of higher-order abstract types, and could potentially lead to undecidability of subtyping.

This problem has already been identified and solved by Rossberg et al. [2014]. We reuse their argument, which we sum-up below. Decidability follows from the fact that subtyping $\Gamma \vdash \mathcal{C} < \mathcal{C}'$ is actually only checked when the right-hand-side signature $\mathcal{C}'$ has an additional property that makes the instantiation easy to compute. More precisely, they define:

- *rooted type variables* Let us consider a type variable α of kind $\star$ first. We say that α is rooted in a signature $\mathcal{C}'$ if the signature contains a type field of the form $\mathsf{type}\ t = \alpha$ in a strictly positive position. The key observation is that rooted type variables can be easily instantiated during subtyping. Indeed, for the subtyping to succeed, there have to be some corresponding type field of the form $\mathsf{type}\ t = \tau$ with the same identifier t occurring in a strictly positive part of $\mathcal{C}$. From the non-variance of declaration subtyping, we know that the only possible instantiation of α has to be τ. Therefore, the instantiation can be found for rooted type variables, using the two signatures $\mathcal{C}$ and $\mathcal{C}'$.
  This argument extends to higher-order types. For functors with one argument, a type variable φ is rooted in $\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}$ if $\mathcal{C}$ contains in a strictly positive position a type field of the form $\mathsf{type}\ t = (\varphi\,\overline{\alpha})$. Again, the left-hand-side signature $\mathcal{C}$ must have a corresponding type field of the form $\mathsf{type}\ t = \tau$. Therefore, the only possible instantiation of φ is $\lambda\overline{\alpha}.\tau$ For several arguments, the set of type variables $\overline{\alpha}$ has to be the concatenation of all universally quantified variables from the different levels of functors.
- *explicit signatures* are signatures where all quantified abstract types are rooted. Signatures obtained by elaboration from source signatures are always explicit.
- *valid signatures* are signatures where all functors have an explicit signature on the left-hand side of the arrow.

From there, they show that typechecking only produces valid signatures, and that subtyping is only checked with an explicit signature on the right-hand side and a valid signature on the left-hand side.

An interesting consequence is that decidability of subtyping (and therefore, decidability of typechecking) crucially relies on the fact that the elaboration of source signatures always produces explicit signatures. If we where to allow the user to input non-explicit signatures—either with an unrestricted **`module type of`** construct, or directly in M^ω signatures syntax—we would lose decidability. This hints at the fact that ML-modules are not just merely a *mode of use of* F^ω (as written in *F-ing* (Rossberg et al. [2014])), but also an *sweet spot*, where changes seen as minor from M^ω could easily break decidability.

Applicative functors Subtyping between functors combines both instantiation and subtyping between arrow types:

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

We have a standard contra-variance for the argument of applicative functors. The same instantiation $[\overline{\alpha} \mapsto \overline{\tau}]$ is used for both the domain and codomain.

3.1.4 Module Expressions type-checking

Typechecking of expressions $\Gamma \vdash \mathtt{M} : \exists^\Diamond \overline{\alpha}.\mathcal{C}$ infers an M^ω-signature $\exists^\Diamond \overline{\alpha}.\mathcal{C}$ given a source module $\mathtt{M}$. Technically, the judgment should be read $\Gamma \vdash^\Diamond \mathtt{M} : \exists^\Diamond \overline{\alpha}.\mathcal{C}$ where the opacity flag on the judgment is a typing mode that is the same as the opacity mode of the existential quantifier. As we use opacity flags as modes, we also refer to the opaque flag $\blacktriangledown$ as the *generative* mode, and to the transparent flag $\triangledown$ as the *applicative* mode.

To lighten the notation, we omit the mode on the judgment except when it is generative and there is no existential type to enforce it. Thus, when we write $\Gamma \vdash \mathtt{M} : \exists^\blacktriangledown \varnothing.\mathcal{C}$ or $\Gamma \vdash \mathtt{M} : \exists^\triangledown \varnothing.\mathcal{C}$ when $\overline{\alpha}$ is empty, we actually mean $\Gamma \vdash^\blacktriangledown \mathtt{M} : \mathcal{C}$ and $\Gamma \vdash^\triangledown \mathtt{M} : \mathcal{C}$. The same convention applies to typing rules for bindings M-Typ-Decl-*. Typing rules for module expressions and bindings are given on Figure 13.

Skolemization The rule for applicative functors is a crucial one:

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

In order to share the abstract types $\overline{\beta}$ produced by the inference of the body of the functor $\mathtt{M}$, we *skolemize* them out of the universal quantification (and out of the arrow type) by making them higher-order, and applying them to the universally quantified variables $\overline{\alpha}$. However, this is unsound when the abstract types are opaque existential types. We thus enforce transparent existential types for the body of the functor. The technical reasons behind this restriction are detailed in Section 3.4. In a nutshell, skolemization of transparent existential types is always sound, and applicative functor only need this restricted form of existentials.

One might wonder why the corresponding rule M-Typ-Sig-AppFct for typing signatures of applicative functors does not feature a similar restriction. The key difference is that typing of signatures is just a mere manipulation of types where everything is possible, as long as the resulting signatures are wellformed. By contrast, for the typing of module expressions, the soundness relies on the ability to actually produce a term of the corresponding type in F^ω.

Propagation of modes Signatures with transparent existentials are inferred by default and are *required* for the body of applicative functors, as seen above. Module expressions that are inherently generative, such as calling a generative functor or computing impure core expressions (or unpacking a first-class module), can only be typed with opaque existential signatures (in generative mode):

$$\textsc{M-Typ-Mod-GenFct}\quad \frac{\Gamma \vdash \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}{\Gamma \vdash () \to \mathtt{M} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}} \qquad \textsc{M-Typ-Mod-AppGen}\quad \frac{\Gamma \vdash P : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}{\Gamma \vdash P() : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}$$

Modules can be *downgraded* from applicative to generative via subsumption, but not the other way around:

$$\textsc{M-Typ-Mod-Seal}\quad \frac{\Gamma \vdash \mathtt{M} : \exists^{\triangledown}\overline{\alpha}.\mathcal{C}}{\Gamma \vdash \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}$$

All other rules are agnostic of the typing mode. With the convention that M-Typ-Mod-Seal is only used when the generative mode is required for the premise of another rule, i.e., applicative signatures are inferred by default, the system is syntax directed.

Structures are regular, i.e., all bindings have the same mode. This is enforced by the Rule M-Typ-Bind-Seq for sequences, where the mode is the same on both premises:

$$\textsc{M-Typ-Mod-Struct}\quad \frac{\Gamma \vdash_A \overline{\mathtt{B}} : \exists^{\diamond}\overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end} : \exists^{\diamond}\overline{\alpha}.\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}} \qquad \textsc{M-Typ-Bind-Seq}\quad \frac{\Gamma \vdash_A \mathtt{B} : \exists^{\diamond}\overline{\alpha}_1.\mathcal{D} \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D} \vdash_A \overline{\mathtt{B}} : \exists^{\diamond}\overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \vdash_A \mathtt{B}, \overline{\mathtt{B}} : \exists^{\diamond}\overline{\alpha}_1, \overline{\alpha}.\mathcal{D}, \overline{\mathcal{D}}}$$

Bindings The rules for bindings assume a given core-language expression typing judgment $\Gamma \vdash^{\diamond} \mathtt{e} : \tau$ equipped with a mode that tracks the presence of effects. As explained in Section 1.2.2, it is not present in current OCaml, where it is the user's responsibility to use the generative functors in such cases:

$$\textsc{M-Typ-Bind-Let}\quad \frac{\Gamma \vdash^{\diamond} \mathtt{e} : \tau}{\Gamma \vdash_A^{\diamond} (\mathtt{let}\, x = \mathtt{e}) : (\mathsf{val}\ x : \tau)} \qquad \textsc{M-Typ-Bind-Type}\quad \frac{\Gamma \vdash \mathtt{u} : \tau}{\Gamma \vdash_A^{\diamond} (\mathtt{type}\, t = \mathtt{u}) : (\mathsf{type}\ t = \tau)}$$

$$\textsc{M-Typ-Bind-Mod}\quad \frac{\Gamma \vdash \mathtt{M} : \exists^{\diamond}\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module}\, X = \mathtt{M}) : (\exists^{\diamond}\overline{\alpha}.\mathsf{module}\ X : \mathcal{C})} \qquad \textsc{M-Typ-Bind-ModType}\quad \frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module\ type}\ T = \mathtt{S}) : (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C})}$$

The core-language expression typing is extended by the following rules for qualified variables:

$$\textsc{M-Typ-Type-Path}\quad \frac{\Gamma \vdash P : \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{val}\ x : \tau \in \overline{\mathcal{D}}}{\Gamma \vdash^{\diamond} P.x : \tau} \qquad \textsc{M-Typ-Type-Local}\quad \frac{A.(x :\ \mathsf{val}\ \tau) \in \Gamma}{\Gamma \vdash^{\diamond} A.x : \tau}$$

Paths The typing of paths, which may contain functor applications (of applicative functors), is defined by the following rules:

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

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

There is an implicit subtyping check at functor application.

Introduction of abstract types Ascription to a signature S that elaborates to a parametric signature $\lambda\overline{\alpha}.\mathcal{C}$ returns a signature $\exists^\nabla\overline{\alpha}.\mathcal{C}$ with transparent abstract types:

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

This rule has some resemblance with Rule M-Typ-Sig-Con for typechecking a concrete signature $(= P < \mathtt{S})$: in both cases, we check that the M^ω signature of P is a subtype of the M^ω-signature $\lambda\overline{\alpha}.\mathcal{C}'$ of S. By contrast, however, we here drop the matching substitution in the result signature $\exists^\nabla\overline{\alpha}.\mathcal{C}'$ and instead introduce the abstract types $\overline{\alpha}$. If the signature is not parametric, i.e., does not bind abstract types, the result does not introduce new abstract types either. In particular, transparent ascription $(P <: \mathtt{S})$, which is syntactic sugar for $(P : (= P < \mathtt{S}))$, i.e., the opaque ascription of P to the transparent signature $(= P < \mathtt{S})$, behaves as expected, filtering out components of P as prescribed by S but without creating new abstract types. Note that applications of an applicative functor (Rule M-Typ-Mod-AppApp) do not introduce new abstract types *per se*, but *applications* of *already existing* higher-order abstract types—which is the key to the sharing between different applications of the same (or an equivalent) functor to the same (or equivalent) arguments.

The only other construct that introduces abstract types is abstract-type binding:

$$\textsc{M-Typ-Bind-AbsType}\quad \Gamma \vdash_A (\mathsf{type}\, t = A.t) : \exists^\diamond\alpha.\,(\mathsf{type}\ t = \alpha)$$

Projection and signature avoidance As explained in Section 1.3, typechecking the projection of a submodule M.X is often a source of signature avoidance: the dependencies of the source signature of X might become dangling after the other components of the signature of M have been lost. However, M^ω signatures do not have internal dependencies; they are non-dependent records, as all paths present in concrete type definitions have been inlined and binders for abstract types have been lifted. This gives a simple projection rule:

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

In principle, the Rule M-Typ-Mod-Proj could return the signature $\exists^\diamond\overline{\alpha}.\mathcal{C}$, leaving all variables in scope after projection. However, it also performs some form of garbage collection by just keeping the subset $\overline{\alpha}'$ of abstract types $\overline{\alpha}$ that appear free in the submodule signature $\mathcal{C}$ so as to avoid keeping useless, unreachable abstract types.

3.2 Identity, Aliasing, and Type Abstraction

So far, our system handles applicativity with a type-level granularity of applicativity, as promoted by Moscow ML. In this section, we present the introduction of identity tags in a simple *source-to-source* transformation as a way to piggy-back the OCaml module-level granularity. We also present a derived type-system that treats identities in a primitive way. Finally, we state and prove a property of identity types that hints at abstract safety (without obtaining it).

3.2.1 A source-to-source transformation

As explained in Section 1.2.2, applicativity of functors rely on a criterion for *module equivalence* which gives rise to three notions of applicativity:

- *type-level applicativity*: the two modules have the same type fields

Source signature

```
struct_A
  module type T = sig_C type t = C.t end
  module M = (struct_B
    module X = (struct type t = int end : A.T)
    module Y = struct
      type a = A.X.t × bool
      type b = A.X.t × int
    end
  end).Y

 let f ((x, _ ) : A.M.a) : A.M.b = (x, 42)
end
```

M^ω signature

```
∃^∇α.sig
  module type T = λα.sig type t = α end
  module M : sig
    type a = α × bool
    type b = α × int
  end

  val f : α × bool → α × int
end
```

Figure 12: Example of typechecking a module in M^ω. The resulting signature in M^ω, on the right-hand-side can express the type-sharing between a and b, despite the fact that there is not type field that defines α (i.e., there is no field of the form type $t = \alpha$)

- *value-level applicativity*: the two modules have the same type and value fields
- *module-level applicativity*: the two modules are statically known as being equal

To obtain the *abstraction safety* provided by the last two options, Rossberg et al. [2014] introduced *semantic paths*: marking value and module fields with *phantom* abstract types and using the type sharing mechanism to track value or module sharing. Then, type-level applicativity can be transformed into either value level or module-level (*à la OCaml*) by marking either all values or only modules.

However, as phantom abstract types act *exactly* as regular abstract types, we can split the introduction of those types from the typing. We propose a simple, compositional *source-to-source* transformation that explicitly introduces special abstract type fields *id*, called *identity tag* in Figure 14. We call *tagged* expressions those resulting from the transformation, so as to distinguish them from *raw* (untagged) expressions. Structures and functors are wrapped inside a two-field structure with its identity tag and the actual value. New (abstract) identity tags are introduced when typing structures and functors, or via an ascription. Conversely, identity tags are shared when aliasing a module.

Controlling the applicativity granularity by a source-to-source transformation allows for a simpler set of typing rules. Besides, it leaves open the choice to apply the transformation so as to obtain OCaml coarse-grain granularity (and abstraction safety), or just stay with the default static equivalence.

3.2.2 Derived typing system

The source-to-source transformation could also be inlined, introducing the phantom identity types during type-checking, as in *F-ing* (Rossberg et al. [2014]). This would affect the rules that "create" new modules,i.e., the rules for structures and functors. Writing the tagged structures with a pair for sake of conciseness, this would give for module expressions:

M-Typ-Mod-Struct
$$\frac{\Gamma \vdash_A \overline{\mathtt{B}} : \exists^{\Diamond}\overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathsf{end} : \exists^{\Diamond}\alpha_{id}, \overline{\alpha}.(\alpha_{id}, \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end})}$$

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

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

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

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

M-Typ-Mod-Seal
$$\frac{\Gamma \vdash \mathtt{M} : \exists^{\triangledown}\overline{\alpha}.\mathcal{C}}{\Gamma \vdash \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}$$

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

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

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

M-Typ-Mod-GenFct
$$\frac{\Gamma \vdash \mathtt{M} : \exists^{\diamond}\overline{\alpha}.\mathcal{C}}{\Gamma \vdash () \to \mathtt{M} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}$$

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

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

M-Typ-Mod-Proj
$$\frac{\Gamma \vdash \mathtt{M} : \exists^{\diamond}\overline{\alpha}.\,\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{module}\ X : \mathcal{C} \in \overline{\mathcal{D}} \qquad \overline{\alpha}' = \mathsf{fv}(\mathcal{C}) \cap \overline{\alpha}}{\Gamma \vdash \mathtt{M}.X : \exists^{\diamond}\overline{\alpha}'.\mathcal{C}}$$

(a) Module expression (and path) typing rules

M-Typ-Bind-Let
$$\frac{\Gamma \vdash^{\diamond} \mathtt{e} : \tau}{\Gamma \vdash_A^{\diamond} (\mathtt{let}\, x = \mathtt{e}) : (\mathsf{val}\ x : \tau)}$$

M-Typ-Bind-Type
$$\frac{\Gamma \vdash \mathtt{u} : \tau}{\Gamma \vdash_A^{\diamond} (\mathtt{type}\, t = \mathtt{u}) : (\mathsf{type}\ t = \tau)}$$

M-Typ-Bind-AbsType
$$\Gamma \vdash_A (\mathtt{type}\, t = A.t) : \exists^{\diamond}\alpha.(\mathsf{type}\ t = \alpha)$$

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

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

M-Typ-Bind-Empty
$$\Gamma \vdash_A \varnothing : \varnothing$$

M-Typ-Bind-Seq
$$\frac{\Gamma \vdash_A \mathtt{B} : \exists^{\diamond}\overline{\alpha}_1.\mathcal{D} \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D} \vdash_A \overline{\mathtt{B}} : \exists^{\diamond}\overline{\alpha}.\overline{\mathcal{D}}}{\Gamma \vdash_A \mathtt{B}, \overline{\mathtt{B}} : \exists^{\diamond}\overline{\alpha}_1, \overline{\alpha}.\mathcal{D}, \overline{\mathcal{D}}}$$

(b) Binding typing rules

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

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

(c) Extension of core-language expression typing

Figure 13: M^{ω}– Module and binding typing rules.

Tagging

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

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

Paths

$$[\![A.X]\!] \triangleq A.X \qquad [\![P.X]\!] \triangleq [\![P]\!].\mathit{Val}.X$$

$$[\![Y]\!] \triangleq Y \qquad [\![P(P')]\!] \triangleq [\![P]\!].\mathit{Val}([\![P']\!])$$

Module expressions

$$[\![\mathtt{M}.X]\!] \triangleq [\![\mathtt{M}]\!].\mathit{Val}.X \qquad [\![P()]\!] \triangleq [\![P]\!].\mathit{Val}()$$

$$[\![(P : \mathtt{S})]\!] \triangleq ([\![P]\!] : [\![\mathtt{S}]\!])$$

$$[\![() \to \mathtt{M}]\!] \triangleq \mathsf{Tag}\,\{() \to [\![\mathtt{M}]\!]\}$$

$$[\![(Y : \mathtt{S}) \to \mathtt{M}]\!] \triangleq \mathsf{Tag}\,\{(Y : [\![\mathtt{S}]\!]) \to [\![\mathtt{M}]\!]\}$$

$$[\![\mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end}]\!] \triangleq \mathsf{Tag}\left\{\mathtt{struct}_A\ [\![\overline{\mathtt{B}}]\!]\ \mathtt{end}\right\}$$

Signatures

$$[\![A.T]\!] \triangleq A.T \qquad [\![P.T]\!] \triangleq [\![P]\!].\mathit{Val}.T$$

$$[\![(= P < \mathtt{S})]\!] \triangleq (= [\![P]\!] < [\![\mathtt{S}]\!])$$

$$[\![() \to \mathtt{S}]\!] \triangleq \mathsf{Tag}\,\{() \to [\![\mathtt{S}]\!]\}$$

$$[\![(Y : \mathtt{S}_a) \to \mathtt{S}]\!] \triangleq \mathsf{Tag}\,\{(Y : [\![\mathtt{S}_a]\!]) \to [\![\mathtt{S}]\!]\}$$

$$[\![\mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end}]\!] \triangleq \mathsf{Tag}\left\{\mathtt{sig}_A\ [\![\overline{\mathtt{D}}]\!]\ \mathtt{end}\right\}$$

Figure 14: Source-to-source transformation introducing identity tags for structures and functors using two reserved identifiers `id` and `Val`. Bindings and declarations are transformed by immediate map over submodules and submodule-types.

Signature elaboration would be modified similarly, introducing parameterized abstract types for structures and functors. The rest of the type system would not be affected, only dealing with $\mathit{id} - \mathit{Val}$ pairs rather than plain signatures in some of the rules. As explained before, the identity tags act exactly as normal type fields. Therefore, subtyping on identity types would be restricted to equality.

3.2.3 Property of identity tags

Identity tags are introduced for new modules, then shared when a module is aliased, either directly or via a transparent ascription. Therefore, if two modules share the same identity tag, they *originate* from a common ancestor with a better signature, as stated by the following theorem:

Theorem 1: Identity tags

Two module expressions that share the same identity tag *originate* from a common ancestor with a better signature. More precisely, if we have

$$\Gamma \vdash [\![\mathtt{M}_1]\!] : \mathsf{sig\ module}\ \mathit{Val} : \mathcal{C}_1\ \ \mathsf{type}\ \mathit{id} = \tau\ \mathsf{end}$$

$$\Gamma \vdash [\![\mathtt{M}_2]\!] : \mathsf{sig\ module}\ \mathit{Val} : \mathcal{C}_2\ \ \mathsf{type}\ \mathit{id} = \tau\ \mathsf{end}$$

then, they originate from the same module, which had a signature that subsumes both:

$$\exists \mathcal{C}_0, \quad \Gamma \vdash \mathcal{C}_0 < \mathcal{C}_1 \quad \wedge \quad \Gamma \vdash \mathcal{C}_0 < \mathcal{C}_2$$

The rest of this section is dedicated to the proof of this theorem. First, we consider a slightly modified system called $\mathsf{M}^\omega_{<:}$ based on F^ω extended with bounded quantification. We identify M^ω types and M^ω signatures and use τ and $\mathcal{C}$ interchangeably in this section. The system $\mathsf{M}^\omega_{<:}$ is built from M^ω as follows:

1. We extend the quantifiers $\exists^\Diamond$, $\forall$, and λ to support bounded quantification for abstract types that serve as identities (the other types being bound by the top bound $\top$).

Source code

```
module X  : sig type t end
module X' : (= X < sig end)

module F (Y:sig end) : (= Y < sig end)
```

Transformed code

```
module X  : sig type id        module Val : sig type t end end
module X' : sig type id = X.id module Val : sig end        end

module F : functor (Y:sig type id module Val : sig end end) →
  sig type id = Y.id module Val : sig end end)
```

Figure 15: An example of the source-to-source transformation. For readability, the result of the transparent ascriptions have been inlined to display the equality of identity types.

2. We modify the typing and subtyping rules accordingly. Omitting the rules that either do not feature bounds or simply thread them from the premise to the conclusion, only three typing rules are affected, as they now feature an additional subtyping condition as their premise (in yellow background to emphasize the differences):

MS-Typ-Sig-Con

$$\frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathtt{S} : \lambda(\overline{\alpha} < \overline{\tau}').\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} < \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa} \qquad \Gamma \vdash \overline{\tau} < \overline{\tau}'}{\Gamma \vdash (= P < \mathtt{S}) : \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]}$$

MS-Typ-Mod-Ascr

$$\frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathtt{S} : \lambda(\overline{\alpha} < \overline{\tau}').\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} < \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa} \qquad \Gamma \vdash \overline{\tau} < \overline{\tau}'}{\Gamma \vdash (P : \mathtt{S}) : \exists^{\nabla}(\overline{\alpha} < \overline{\tau}').\mathcal{C}'}$$

MS-Typ-Mod-AppApp

$$\frac{\begin{array}{c}\Gamma \vdash P : \forall(\overline{\alpha} < \overline{\tau}').\mathcal{C}_a \to \mathcal{C} \\ \Gamma \vdash P' : \mathcal{C}' \qquad \Gamma \vdash \mathcal{C}' < \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa} \qquad \Gamma \vdash \overline{\tau} < \overline{\tau}'\end{array}}{\Gamma \vdash P(P') : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]}$$

3. We modify the typing rules for introducing abstract types (MS-Typ-Decl-TypeAbs and MS-Typ-Bind-AbsType) by distinguishing identity tags (rules MS-Typ-Decl-TypeAbsId and MS-Typ-Bind-AbsTypeId) from other abstract types. While abstract types are simply bound by $\top$, identity tags are bound by the signature of the associated value.

MS-Typ-Decl-TypeAbs

$$\frac{t \neq id}{\Gamma \vdash_A (\mathsf{type}\, t = A.t) : \lambda(\alpha < \top).(\mathsf{type}\, t = \alpha)}$$

MS-Typ-Decl-TypeAbsId

$$\frac{\Gamma \vdash A.Val : \mathcal{C}}{\Gamma \vdash_A (\mathsf{type}\, id = A.id) : \lambda(\alpha < \mathcal{C}).(\mathsf{type}\, id = \alpha)}$$

MS-Typ-Bind-AbsType

$$\frac{t \neq id}{\Gamma \vdash_A (\mathsf{type}\, t = A.t) : \exists^{\Diamond}(\alpha < \top).(\mathsf{type}\, t = \alpha)}$$

MS-Typ-Bind-AbsTypeId

$$\frac{\Gamma \vdash A.Val : \mathcal{C}}{\Gamma \vdash_A (\mathsf{type}\, id = A.id) : \exists^{\Diamond}(\alpha < \mathcal{C}).(\mathsf{type}\, id = \alpha)}$$

4. Subtyping is extended with a new rule MS-Sub-Bound for subtyping applied higher-order bound variables (MS-Sub-Bound-Star is just a particular case of MS-Sub-Bound):

$$\textsc{MS-Sub-Bound}\quad \frac{\varphi < \lambda(\overline{\alpha} < \overline{\tau}').\mathcal{C} \in \Gamma \qquad \Gamma \vdash \overline{\tau} < \overline{\tau}'}{\Gamma \vdash (\varphi\,\overline{\tau}) < \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]} \qquad \textsc{MS-Sub-Bound-Star}\quad \frac{\alpha < \mathcal{C} \in \Gamma}{\Gamma \vdash \alpha < \mathcal{C}}$$

Besides, the two following rules for subtyping between functors are also extended with additional premises to ensure the correct instantiation of abstract types:

$$\textsc{MS-Sub-Sig-GenFct}\quad \frac{\Gamma, \overline{\alpha} < \overline{\tau} \vdash \mathcal{C} < \mathcal{C}'[\overline{\alpha}' \mapsto \overline{\tau_1}] \qquad \Gamma, \overline{\alpha} < \overline{\tau} \vdash \overline{\tau_1} < \overline{\tau}'}{\Gamma \vdash () \to \exists^{\blacktriangledown}(\overline{\alpha} < \overline{\tau}).\mathcal{C} < () \to \exists^{\blacktriangledown}(\overline{\alpha}' < \overline{\tau}').\mathcal{C}'}$$

$$\textsc{MS-Sub-Sig-AppFct}\quad \frac{\Gamma, \overline{\alpha}' < \overline{\tau}' \vdash \mathcal{C}_a' < \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau_1}] \qquad \Gamma, \overline{\alpha}' < \overline{\tau}' \vdash \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau_1}] < \mathcal{C}' \qquad \Gamma, \overline{\alpha}' < \overline{\tau}' \vdash \overline{\tau_1} < \overline{\tau}}{\Gamma \vdash \forall(\overline{\alpha} < \overline{\tau}).\mathcal{C}_a \to \mathcal{C} < \forall(\overline{\alpha}' < \overline{\tau}').\mathcal{C}_a' \to \mathcal{C}'}$$

Crucially, subtyping in $\mathsf{M}^{\omega}_{<:}$ remains transitive.

The proof then proceeds in two steps:

1. We first show that the $\mathsf{M}^{\omega}_{<:}$ maintains an stronger notion of wellformedness: *identity wellformedness*. If $\Gamma \vdash \mathsf{sig\ type}\ id = \tau\ \mathsf{module}\ Val : \mathcal{C}\ \mathsf{end}$ then $\Gamma \vdash \tau < \mathcal{C}$ (**1**). To do so, we reinforce wellformedness for identity tags by changing the rule to

$$\frac{\Gamma \vdash \mathcal{C} : \mathsf{wf} \qquad \Gamma \vdash \tau < \mathcal{C}}{\Gamma \vdash \mathsf{sig\ type}\ id = \tau\ \mathsf{module}\ Val : \mathcal{C}\ \mathsf{end} : \mathsf{wf}}$$

   This defines a stronger wellformedness judgment $\Gamma \vdash \mathcal{C} : \mathsf{wf}_{id}$. We show that $\mathsf{M}^{\omega}_{<:}$ typing judgments for modules and signatures always produce signatures that preserves identity-tag wellformedness. That is, $\vdash \Gamma : \mathsf{wf}_{id}$ and either $\Gamma \vdash \mathtt{S} : \lambda(\overline{\alpha} < \overline{\mathcal{C}}).\mathcal{C}'$ or $\Gamma \vdash \mathtt{M} : \exists^{\diamondsuit}(\overline{\alpha} < \overline{\mathcal{C}}).\mathcal{C}'$ implies $\Gamma, \overline{\alpha} < \overline{\mathcal{C}} \vdash \mathcal{C}' : \mathsf{wf}_{id}$.

   Therefore, we may restrict $\mathsf{M}^{\omega}_{<:}$ derivations to use the identity-tag wellformedness invariant as long as we start with an identity-tag wellformed environment. By inversion of wellformedness, this ensures the invariant (1).

2. Using the previous invariant, we then show that typability in M^{ω} implies typability in $\mathsf{M}^{\omega}_{<:}$.

The proof of the first step proceeds by a simple induction over the typing derivation: identities are either introduced fresh, in which case the bound is equal to the signature of the corresponding *Val*-field, or obtained via subtyping, in which case we use transitivity of subtyping.

The rest of this section is dedicated to the proof of the second step. The only differences between the original and the enhanced typing systems are the addition of subtyping bounds and subtyping relations between the bounds. The core of the proof is to show that these are actually not restrictive, which follows from two key facts: (1) subtyping is only done with a right-hand side signature that *comes from a source signature*, i.e., that is the result of typing a source signature; and (2) such signatures always contain the bounds of their identity tags in at least one positive occurrence.

Properties of elaborated signatures An M^ω signature that the elaboration of a source signature (referred to as TSS in the following) has actually a stronger property than being just identity-wellformed. In a TSS, the bound of an identity tag α is *exactly* the signature $\mathcal{C}$ of the first module occurrence with an identity tag α, and therefore, the bound always appears *explicitly* in the signature. By contrast, in signatures that just identity-wellformed (like the result of inference), there might be no module with the same signature as the bound, but only supertypes.

First-order example Before diving into the proof by induction, we consider the typing of a basic source signature $\mathtt{S}$:

$$\Gamma \vdash \mathtt{S} : \lambda(\alpha < \mathcal{C}_\alpha).\mathcal{C}$$

There must be a subterm of $\mathcal{C}$ at a positive occurrence that is equal to:

$$\mathsf{sig\ module}\ \mathit{Val} : \mathcal{C}_\alpha\ \mathsf{type}\ \mathit{id} = \alpha\ \mathsf{end}$$

When subtyping this signature with another one in the enhanced system,

$$\Gamma \vdash \lambda(\alpha < \mathcal{C}_\alpha).\mathcal{C} < \lambda(\beta < \mathcal{C}_\beta).\mathcal{C}'$$

there is a new subtyping check between the bounds:

$$\Gamma \vdash \mathcal{C}_\beta < \mathcal{C}_\alpha \qquad (\mathbf{1})$$

Whenever the subtyping between $\mathcal{C}'$ and $\mathcal{C}$ succeeded in M^ω, $\mathcal{C}'$ features a subterm of the form

$$\mathsf{sig\ module}\ \mathit{Val} : \mathcal{C}_0\ \mathsf{type}\ \mathit{id} = \beta\ \mathsf{end}$$

By subtyping, we have $\Gamma \vdash \mathcal{C}_0 < \mathcal{C}_\alpha$. Thanks to the invariant of the first part of the proof, we know that $\Gamma \vdash \mathcal{C}_\beta < \mathcal{C}_0$. Transitivity of subtyping ensures (1). Hence subtyping also succeeds in $\mathsf{M}^\omega_{<:}$.

Proof by induction We prove by induction over the typing derivation that typing and subtyping in M^ω implies typing and subtyping in $\mathsf{M}^\omega_{<:}$. Cases for unchanged rules, or rules that just thread the bounds from the premise to the conclusion are immediate. The only interesting cases are the three typing rules and two subtyping rules shown above have an additional premise In each case we prove that this additional premise is actually implied by the other premises.

Typing M-Typ-Sig-Con. The M^ω derivation ends with the following rule.

M-Typ-Sig-Con

$$\frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} < \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}] \qquad \Gamma \vdash \overline{\tau} : \overline{\kappa}}{\Gamma \vdash (= P < \mathtt{S}) : \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]}$$

For simplification of presentation, we assume that $\overline{\alpha}$ is a single variable α. We show that we can rebuild a $\mathsf{M}^\omega_{<:}$ derivation:

MS-Typ-Sig-Con

$$\frac{\begin{array}{c}\Gamma \vdash P : \mathcal{C} \\ \Gamma \vdash \mathtt{S} : \lambda(\alpha < \tau').\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} < \mathcal{C}'[\alpha \mapsto \tau]\ (\mathbf{2}) \qquad \Gamma \vdash \tau : \kappa \qquad \Gamma \vdash \tau < \tau'\ (\mathbf{3})\end{array}}{\Gamma \vdash (= P < \mathtt{S}) : \mathcal{C}'[\alpha \mapsto \tau]}$$

All the premises but (3), hence including (2), follow by induction hypothesis. The remaining goal is to show that the additional condition (3) actually follows from (2). For this purpose, we define $[\mathcal{C}]^+$, the *positive* declarations of a signature $\mathcal{C}$, as the flattened list of declarations in strict positive positions inside $\mathcal{C}$ (without entering inside generative functors or module types), taken in the usual binding order, as follows:

$$\begin{aligned}
[\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}]^+ &= \overline{[\mathcal{D}]^+} && \text{(Structural signature)}\\
[\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}]^+ &= \forall\overline{\alpha}.[\mathcal{C}]^+ && \text{(Applicative functor)}\\
[() \to \_\,]^+ &= \varnothing && \text{(Generative functor)}\\
[\mathsf{module}\ X : \mathcal{C}]^+ &= (\mathsf{module}\ X : \mathcal{C}), [\mathcal{C}]^+ && \text{(Submodule declaration)}\\
[\mathcal{D}]^+ &= \mathcal{D} && \text{(Other declarations)}
\end{aligned}$$

Declarations inside applicative functors are universally quantified. A key observation is that a signature in TSS form always contains the bound of its identity type among its positive declarations:

$$\begin{aligned}
\Gamma \vdash \mathsf{S} : \lambda(\alpha < \tau).\mathcal{C} &\implies (\mathsf{module}\ \mathit{Val} : \tau) \in [\mathcal{C}]^+ \land \mathsf{type}\ \mathit{id} = \alpha \in [\mathcal{C}]^+\\
\Gamma \vdash \mathsf{S} : \lambda(\alpha < \lambda\overline{\beta}.\tau).\mathcal{C} &\implies \forall\overline{\beta}.(\mathsf{module}\ \mathit{Val} : \tau) \in [\mathcal{C}]^+ \land \forall\overline{\beta}.\mathsf{type}\ \mathit{id} = (\alpha\overline{\beta}) \in [\mathcal{C}]^+
\end{aligned}$$

Crucially, all the instantiations we consider feature a TSS $\mathcal{C}'$ on their right-hand side. By construction, subtyping between two signatures $\mathcal{C}$ and $\mathcal{C}'$ implies subtyping between declarations at any positive occurrence in $\mathcal{C}$ and its corresponding declaration in $\mathcal{C}'$. Ignoring applicative functors at first, this would give:

$$\Gamma \vdash \mathcal{C} < \mathcal{C}' \implies \forall \mathcal{D}' \in [\mathcal{C}']^+.\ \exists \mathcal{D} \in [\mathcal{C}]^+.\ \Gamma \vdash \mathcal{D} < \mathcal{D}'$$

But there is a catch: the declarations $\mathcal{D}$ and $\mathcal{D}'$ might be deep inside the signature, and therefore the subtyping between them might be true only in an extended environment Γ'. The correct statement is therefore:

$$\Gamma \vdash \mathcal{C} < \mathcal{C}' \implies \forall \mathcal{D}' \in [\mathcal{C}']^+.\ \exists \mathcal{D} \in [\mathcal{C}]^+, \Gamma'.\ \Gamma' \vdash \mathcal{D} < \mathcal{D}' \land \Gamma' < \Gamma$$

More specifically, there exists a declaration in the positive part of $\mathcal{C}$ that is a subtype of the explicit bound of α, which appears in $[\mathcal{C}']^+$. That is,

$$\exists \mathcal{C}_0.\ \Gamma' \vdash \mathsf{module}\ \mathit{Val} : \mathcal{C}_0 < \mathsf{module}\ \mathit{Val} : \tau'$$

This implies $\Gamma' \vdash \mathcal{C}_0 < \tau'$. Using the invariant of the first step of the proof, we get $\Gamma' \vdash \tau < \mathcal{C}_0$, which implies (3) by transitivity of subtyping and weakening of the environment. This argument applies to the three typing rules. It extends to higher-order declarations with universal quantification only adding universally quantified variables in the typing context Γ.

Typing rules MS-TYP-MOD-ASCR *and* MS-TYP-MOD-APPAPP. The former is exactly the same as the previous case. For the latter, the only change if that the bounded quantification $\lambda(\alpha < \tau').\mathcal{C}'$ is being replaced by $\forall(\alpha < \tau').\mathcal{C}'_a \to \mathcal{C}'$.

Subtyping Rule MS-SUB-SIG-APPFCT. The same argument as the one used by typing rules applies, just with an extended context. Restricting again to a single type variable for the sake of readability, we need to rebuild a derivation in $\mathsf{M}^{\omega}_{<:}$ that ends with:

MS-SUB-SIG-APPFCT

$$\frac{\Gamma, \alpha' < \tau' \vdash \mathcal{C}'_a < \mathcal{C}_a[\alpha \mapsto \tau_1]\ (4) \qquad \dots \qquad \Gamma, \alpha' < \tau' \vdash \tau_1 < \tau\ \ \mathbf{(5)}}{\Gamma \vdash \forall(\alpha < \tau).\mathcal{C}_a \to \mathcal{C} < \forall(\alpha' < \tau').\mathcal{C}'_a \to \mathcal{C}'}$$

where all the premises but (5) follow by induction hypothesis.

As $\mathcal{C}_a$ is a TSS, we have

$$\mathsf{sig}\ \mathsf{type}\ id = \alpha\ \mathsf{module}\ Val : \tau\ \mathsf{end}\ \in\ [\mathcal{C}_a]^{+}$$

Therefore, since α is not free in τ,

$$\mathsf{sig}\ \mathsf{type}\ id = \tau_1\ \mathsf{module}\ Val : \tau\ \mathsf{end}\ \in\ [\mathcal{C}_a[\alpha \mapsto \tau_1]]^{+}$$

Correspondingly, we must have in $\mathcal{C}'_a$, for some signature σ:

$$\mathsf{sig}\ \mathsf{type}\ id = \tau_0\ \mathsf{module}\ Val : \sigma\ \mathsf{end}\ \in\ [\mathcal{C}'_a]^{+}\ (\mathbf{6})$$

Subtyping component by component, we have, since subtyping in non-variant on type fields in general and on identity type fields in particular:

$$\Gamma, \alpha' < \tau' \vdash \sigma < \tau \quad \wedge \quad \tau_1 = \tau_0$$

The identity-tag wellformedness invariant of (6) implies:

$$\Gamma, \alpha' < \tau' \vdash \tau_0 < \sigma$$

By transitivity of subtyping we get $\Gamma, \alpha' < \tau' \vdash \tau_1 < \tau$, i.e.,. (5), as expected.

Rule MS-Sub-Sig-GenFct. We rebuild a derivation of the form

$$\textsc{MS-Sub-Sig-GenFct}\quad \frac{\Gamma, \overline{\alpha} < \overline{\tau} \vdash \mathcal{C} < \mathcal{C}'\left[\overline{\alpha}' \mapsto \overline{\tau_1}\right]\ (\mathbf{7}) \qquad \Gamma, \overline{\alpha} < \overline{\tau} \vdash \overline{\tau_1} < \overline{\tau}'\ (\mathbf{8})}{\Gamma \vdash () \to \exists^{\blacktriangledown}(\overline{\alpha} < \overline{\tau}).\mathcal{C} < () \to \exists^{\blacktriangledown}(\overline{\alpha}' < \overline{\tau}').\mathcal{C}'}$$

The proof is similar to the previous case where (7) follows by induction and (8) follows from (7) and the fact that $\mathcal{C}'$ is in TSS-form.

This completes the proof.

3.3 Rebuilding Source Signatures

In this section, we present a reverse translation from M^{ω} signatures back into the source syntax, called *anchoring*. It a partial inverse of signature elaboration. This translation is necessarily incomplete as some inferred signatures cannot be expressed in the less expressive source syntax. Anchoring relies on identity tags, and therefore assumes that we have tagged source programs as described in Section 3.2 prior to type checking. That is, anchoring translates *tagged* signatures back into source (hence untagged) signatures.

In Section 3.3.1, we detail the three expressiveness gaps of the source syntax, which gives us three *anchoring conditions*, associated with three new typechecking errors if those conditions are violated. We argue that these guidelines lead to more understandable signature avoidance error messages. In Section 3.3.2, we present an anchoring algorithm. In Section 3.3.3, we state and prove the properties of this algorithm, which shines a new light on the differences in the way type-sharing is expressed between M^{ω} and the source syntax.

3.3.1 The Expressiveness Gaps of the Source Syntax

In this section we introduce three *anchoring conditions*. For each, we show examples of an M^{ω}-signature meting the condition and we detail the error cases when the condition is not met. Those error cases constitute a new form of typechecking error that is more fine grained than a simple "avoidance" error. Overall, the anchoring conditions ensure that an M^{ω}-signature is the result of the elaboration of some source signature, with an additional constraint for functor applications. We state three anchoring conditions for pedagogical purposes, but the third one subsumes the first two.

For the sake of readability, we do not display all module identities in the examples, only the ones that are relevant.

Abstract Type Fields

Structural information and introduction of types A first key insight is the difference in the source syntax between the declaration of a *manifest* type (`type` t = `u`) and that of an *abstract* type (`type` $t = A.t$). An abstract type declaration `type` $t = A.t$ in a covariant position effectively *creates* a new abstract type (introducing an existential quantifier in M^ω) and *adds* a type field t to the signature. By contrast a manifest type definition `type` t = `u` only states structural information—adding a field t to refer to the existing type `u`.

Therefore, in the source syntax, the structural information (name and position of fields) determines the scope of abstract types. Conversely, the M^ω syntax *separates* the introduction of new abstract types from the introduction of fields by using explicit quantifiers. In particular, they may mention an abstract type without (or before) having a type declaration that refer to it. Overall, anchoring is possible if the structural and scoping information happen to coincide, i.e., if there are type declarations for each type variable that "introduce" the variable in the right scope. As detailed in Section 2.1.1, new scopes are introduced by generative functors and module type definitions. This gives the following first anchoring guideline:

Anchoring Condition 1

The first occurrence of an abstract type variable α must be a type declaration that is: (1) of the form type $t = \alpha$, (2) in a strictly positive position, and (3) in the same scope as the binder.

If such type declaration exists, its called the *anchoring point* of α. If one of the conditions is not met, the M^ω signature cannot be anchored. This leads to the following error cases:

1. If the first occurrence is not a type field (as on the left-hand side), or if does not contain exactly α (as on the right-hand side):
   $\exists^\triangledown \alpha.$ sig val $x : \alpha$ ... end $\exists^\triangledown \alpha.$ sig type $t = \alpha \times$ `bool` ... end
2. If the first occurrence is not in a (strictly) positive position, used in a functor parameter for instance:
   $\exists^\diamond \alpha.$ sig module F : (sig type $t = \alpha$ end) $\rightarrow$ (sig ... end) end
3. The first occurrence is not in the same scope as the binder, either inside a generative functor or a module type (we detail the treatment of applicative functor later in this section):
   $\exists^\diamond \alpha.$ module G : () $\rightarrow$ sig type $t = \alpha$ end
   $\exists^\diamond \alpha.$ module type T = sig type $t = \alpha$ end

Related notions This notion is linked to *type locators* of ?Rossberg and Dreyer [2013]: the locator is the original anchoring point of α. A key mechanism of anchoring is to re-discover a new locator, as the original one may have been lost. The notion of anchoring point is also closely related to the notion of *type variable roots* in Rossberg et al. [2014]: the anchoring point of α is a root for α, but the converse is not necessarily true. Type roots are used to show the decidability of type-checking, but do not have to be the first occurrence.

Example 3.3.1. *In the following M^ω signature (left-hand side), the first occurrence of α is a type declaration that can serve as an anchoring point. Therefore, we can anchor the signature to the right-hand-side source signature:*

```
∃◇α. module M : sig
  type t = α
  val x : α
  type u = α × int
end
```

```
module M : sig_A
  type t = A.t
  val x : A.t
  type u = A.t × int
end
```

The type declaration $\texttt{type}\, t = A.t$ is the *anchoring point* of α.

Module Identities

The source syntax can only express identity sharing between modules via *transparent* signatures $(= P < \texttt{S})$. However, a transparent signature $(= P < \texttt{S})$ is wellformed only if the signature of the path P is a subtype of $\texttt{S}$. This forces all modules sharing the (same) identity of P to have a signature that is a subtype of (the signature of) the module at P. Therefore, identity sharing is tied with *subtyping conditions* between the signature of each occurrence and the signature of the first occurrence. By contrast, M^ω signatures can express identity sharing regardless of the associated signatures.

Anchoring Condition 2

The first occurrence of an identity tag variable α_{id} must be at a module binding of the form module $X : (\alpha_{id}, \mathcal{C})$, in a strictly positive position, and in the same scope as the binder. Besides, all further occurrences of α_{id}, of the form $(\alpha_{id}, \mathcal{C}')$, must satisfy a subtyping criterion between $\mathcal{C}$ and $\mathcal{C}'$.

As a consequence of this condition, anchoring can fail even if all variables have an anchoring point as their first occurrence: we might discover latter in the signature that subtyping conditions are not met. In addition to the three error cases that are similar to type variables, we have a new error case: if one of the following occurrences of the identity tag variable is associated with a signature that is not a subtype of the one at the anchoring point:

```
∃◇α_id. module M : sig
  module X₁ : (α_id, sig end)
  module X₂ : (α_id, sig type t = int end)
end
```

Here, the signature of X_1 is not a subtype of the one of X_2: sig end $\not<$ sig type $t = \texttt{int}$ end

Example 3.3.2. *In the following M^ω signature (left-hand-side), the first module with the identity tag serves as the anchoring point for the identity variable α_{id}.*

```
∃◇α_id, α. module M : sig
  module X₁ : (α_id, sig type t = α end)
  module X₂ : (α_id, sig end)
end
```

```
module M : sig_A
  module X₁ : sig_A type t = A.t end
  module X₂ : (= X₁ < sig end)
end
```

All other occurrences of α_{id} (here, only at the module declaration of X_2) validate the subtyping condition, as we have: sig end $<$ sig type $t = \alpha$ end

Higher-order Abstract Types

Let us consider an abstract type t inside an applicative functor:

```
module F : (Y : S) → sig_A ... type t = A.t ... end
```

This type field t is reachable by a path with a functor application, of the form $F(X).t$. Therefore, the type does not act exactly as an higher-order type, but is restricted to a certain *domain* that is limited by the signature of the parameter S. Yet M^ω uses a normal higher-order abstract variable φ to model this type declaration, and φ can be applied to any type argument (without restriction). For anchoring, we need to replace occurrences of φ by an applicative paths of the form $F(X).t$, which will introduce subtyping conditions (between the signature of X and S).

Anchoring Condition 3 (1/2)

The first occurrence of an higher-order abstract type φ must be a type declaration that is: (1) of the form type $t = \varphi(\overline{\alpha})$, (2) where $\overline{\alpha}$ is exactly the set of universally quantified variables in the environment in the current scope, (3) in a strictly positive position and (4) in the same scope as the binder. Besides, all further occurrences of φ, of the form $\varphi(\overline{\tau})$, must be the result of the elaboration of a (well-formed) path, of the form $F(X).t$.

Again, this introduces new error cases if the conditions are not met. If an higher-order abstract type does not have an anchoring point, we say that we have a "lost functor" anchoring error. Besides the three conditions (first occurrence, strict positivity, same scope) that are similar to base type variables, we have:

1. If the first occurrence of φ is a type declaration of the form type $t = \varphi(\overline{\tau})$, where $\overline{\tau}$ is not exactly the set of universally quantified variables, it means that φ *originates* from a functor with a different arity, and we have an error:

```
∃◇ φ, α_id. sig
  module X : sig type t = bool end
  module F : ∀α.(α, sig  end) → sig type t = φ(α, α_id) end
  type t = φ(α_id, α_id)
end
```

   Here, the variable φ was introduced by a functor that took two parameters, and it only remains a partial application $\varphi(\alpha, \alpha_{id})$ in the signature. Even if the rest of the signature only uses φ in a way that could be emulated using F, i.e., the last field could technically be anchored as type $t = F(X).t$, we refuse to do so and throw an error, as it is a form of over-abstraction.

2. If we have a suitable anchoring point, there might be occurrences further down the signature that cannot be expressed as paths to the anchoring point, due to the subtyping conditions.

```
∃◇ φ, α_id. sig
  module F : ∀α.(α, sig val x : int end) → sig type t = φ(α) end
  module X : (α_id, sig val x : bool end)
  type t = φ(α_id)
end
```

   Here, the type declaration cannot be anchored, as the path $F(X).t$ would be ill-formed. Indeed, the signature of X is not a subtype of the functor's parameter, as they differ on the type of the value field x. Such situation can appear by inference, when φ was introduced by a functor that was *more general* than F, i.e., which parameter's signature did not contain a field x.

Relation with the other anchoring conditions The first anchoring condition is a subcase of this one, restricted to base types (of the base kind $\star$). The subtyping condition of the second anchoring condition are similar to the one of this condition. This comes from the fact transparent ascription can be encoded as functor application: for each signature $\mathtt{S}$, we could introduce a dummy functor $F_{\mathtt{S}} : (Y : \mathtt{S}) \rightarrow Y$. Each transparent ascription $(= P < \mathtt{S})$ could be obtained as the result of the application $F_{\mathtt{S}}(P)$.

Example 3.3.3. *In the following M^{ω}-signature (left-hand side), the anchoring conditions are met. All occurrences of φ can be anchored as paths to the anchoring point. Those paths can contain functor parameters, as displayed by F_2.*

```
∃▽ φ, α_id. module M : sig
  module F_1 :
    ∀α_id.(α_id, sig  end) →
      sig type t = φ(α_id) end
  module X : (α_id, sig val x : int end)
  type t = φ(α_id)
  module F_2 :
    ∀α_id.(α_id, sig val x : bool end) →
      sig type t = φ(α_id) end
end
```

```
module M : sig_A
  module F_1 :
    (Y : sig  end) →
      sig_B type t = B.t end
  module X : sig val x : int end
  type t = A.F_1(A.X).t
  module F_2 :
    (Y : sig val x : bool end) →
      sig type t = A.F_1(Y).t end
end
```

Disabling functor applications "out of thin air" As discussed in Section 1.3.3, we want to prevent the anchoring from *inventing* paths with functor applications that never appeared in the source, just for referring to abstract types that have lost their original path. We say that This would be quite surprising, if not misleading, as it suggests a computation that will never happen. Therefore, we extend the third anchoring condition:

Anchoring Condition 3 (2/2)

The anchoring point of an higher-order type should be *original*, either in the functor that introduced it or in an alias of it.

To distinguish original anchoring points, we will need to slightly instrument the typing rules, as this information cannot be reconstructed just from types.

3.3.2 The Anchoring Process

In this section we present an algorithm that produces a source signature given an M^ω one, when the anchoring conditions are met. For pedagogical purposes, it is split in two steps: we first translate M^ω signatures into tagged source signatures, before removing tags to obtain source signatures. Both steps are presented as relations, although they are deterministic. We first explain some instrumentation added to the typing judgment: scope barriers, arity delimiters, type variables flavor, and application marks.

Instrumenting the typing judgment (1/4) First, we extend the grammar of environments with *scope barriers*: we write $\Gamma \cdot \Gamma'$ for an environment that behaves as Γ, Γ' but with a barrier between Γ and Γ' and let Δ range over environments without barriers. Hence, by writing $\Gamma \cdot \Delta$, we mean that Δ is the part of the environment right after the rightmost mark. This is used to indicate scopes (adding a barrier) and prevent anchoring of types that have been introduced in a larger scope. Marks are introduced in the context by typing rules that open scopes:

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

M-Typ-Decl-ModType
$$\frac{\Gamma \cdot\ \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module\ type}\ T = \mathtt{S}) : (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C})}$$

M-Typ-Mod-GenFct
$$\frac{\Gamma \cdot\ \vdash \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}{\Gamma \vdash () \to \mathtt{M} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}}$$

M-Typ-Bind-ModType
$$\frac{\Gamma \cdot\ \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module\ type}\ T = \mathtt{S}) : (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C})}$$

Instrumenting the typing judgment (2/4) We also instrument Rule M-Typ-Mod-AppFct to mark skolemization steps, writing $\beta'\langle\overline{\alpha}\rangle$ instead of $\beta'(\overline{\alpha})$ but to mean the same, so that anchoring may pattern-match on list of lists of arguments rather than on a flat list:

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

This makes the treatment of arity of functors easier.

Instrumenting the typing judgment (3/4) When adding new variables in the environment, we add a flag to indicate if they come from an universal quantification. We write $?\overline{\alpha}$ to indicate that $\overline{\alpha}$ where universally quantified and we write $!\overline{\alpha}$ otherwise (for existential or lambda quantification). This is done only so we can define operator $\mathsf{args}(\Delta)$, which returns a list (of lists) of universally quantified variables. For the sake of readability, the flags are written only when relevant for the context. Alternatively, we could also have identified universally quantified variables by the fact that they immediately precede a functor parameters in Δ. We added flags instead because they also simplify the presentation of the proofs in Section 3.3.3.

Instrumenting the typing judgment (4/4) Finally, we modify the typing rule for functor application M-Typ-Mod-AppApp to *mark* higher-order abstract types, in order to identify *original* anchoring points: type declarations obtained by a functor application are marked as not original, while aliasing a functor copies its type declaration unmarked. Technically, we use a syntactic mark $\tau^{\dagger}$ on types, which can be seen as the introduction of a postfixed constant † that behaves as $\lambda\alpha.\alpha$. That is, $\tau^{\dagger}$ syntactically differ from τ but really means τ. Marks ignored by subtyping rules that may freely erase them. We write $\mathcal{C}^{\dagger}$ for the signature $\mathcal{C}$ where all type declarations $\mathsf{type}\ t = \tau$ of the structure and substructures have been rewritten into $\mathsf{type}\ t = \tau^{\dagger}$—but the marking does not go inside the body of functors nor inside module types. Therefore, we only change the resulting signature of the rule M-Typ-Mod-AppApp to a marked signature $\mathcal{C}'^{\dagger}[\overline{\alpha} \mapsto \overline{\tau}]$:

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

From M^{ω} Signatures to Tagged Source Signatures

The algorithm proceeds by visiting the M^{ω} signature in left-to-right depth-first order. Along the way, it removes all universal and existential quantifiers from the M^{ω} signature and replaces occurrences of the corresponding abstract type variables by either a self-reference (at its anchoring point) or a path referring to its anchoring point. An *anchoring map* θ from M^{ω} types τ to qualified types $P.t$ is built and updated during the visit. The algorithm is defined by mutually recursive judgments :

- **Environment anchoring** $\Gamma \hookrightarrow \theta$ checks that θ is a correct anchoring map for Γ.

- **Signature anchoring** $\Gamma ; \theta \vdash \mathcal{C} \overset{P}{\hookrightarrow} \mathtt{S} : \theta$, given a path P (which is actually shallow, i.e., taking one of the three forms Y, $A.X$, or $A.\mathit{Val}(Y)$), translates the M^{ω} signature $\mathcal{C}$ into a tagged source signature $\mathtt{S}$ and produces a (possibly empty) local anchoring map θ of the abstract types anchored in $\mathtt{S}$, prefixed by P. We also define declaration anchoring $\Gamma ; \theta \vdash \mathcal{D} \overset{A}{\hookrightarrow} \mathtt{D} : \theta$, with a self reference A in place of the path P.

- **Local anchoring** $\Gamma;\theta \vdash \mathcal{C} \hookrightarrow \mathtt{S} : (\overline{\alpha} \mapsto \_\,)$ just checks that signature $\mathcal{C}$ can be translated into $\mathtt{S}$ producing a local map of domain $\overline{\alpha}$ that is ignored afterwards.

- **Type anchoring** $\Gamma\,;\theta \vdash \tau \hookrightarrow \mathtt{u}$ translates the M^ω type τ to a source type $\mathtt{u}$, replacing each M^ω variable by a path to its anchor, as described in θ.

The whole set of rules is given in Figure 16. The key rules are those that extend, update, or use the anchoring map to reconstruct source type expressions. We start with a simple example:

Example 3.3.4. *To illustrate the anchoring process, we consider the M^ω signature on the left-hand side. At each line, we give the corresponding anchoring map, used to produce the source signature on the right-hand side.*

1	$\exists^\nabla \alpha, \beta.\mathsf{sig}$	$\varnothing$	$\mathtt{sig}_A$
2	$\mathsf{type}\ t = \alpha$	$(\alpha \mapsto A.t)$	$\mathtt{type}\ t = A.t$
3	$\mathsf{module}\ X : \mathsf{sig}$	$(\alpha \mapsto A.t)$	$\mathtt{module}\ X : \mathtt{sig}_B$
4	$\quad\mathsf{type}\ u = \beta$	$(\alpha \mapsto A.t) \uplus (\beta \mapsto B.u)$	$\quad\mathtt{type}\ u = B.t$
5	$\quad\mathsf{val}\ x : \beta \times \alpha$	$(\alpha \mapsto A.t) \uplus (\beta \mapsto B.u)$	$\quad\mathtt{val}\ x : B.t \times A.t$
6	end	$(\alpha \mapsto A.t) \uplus (\beta \mapsto A.X.u)$	$\mathtt{end}$
7	$\mathsf{val}\ y : \beta \times \alpha$	$(\alpha \mapsto A.t) \uplus (\beta \mapsto A.X.u)$	$\mathtt{val}\ y : A.X.u \times A.t$
8	end		$\mathtt{end}$

The anchoring map is extended at line (2) and (4) when the anchoring points for α and β are found. The map is updated at line (6) when exiting the submodule X: the path for the anchoring point of β is prefixed by X.

Anchoring points A new anchoring point is introduced when reaching a type declaration of the form $\mathsf{type}\ t = \alpha$ where α is not anchored yet, i.e., not in the domain of θ. A simplified rule for first-order types is:

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

We ensure that the type α was introduced after the left-most barrier, by requiring $\alpha \in \Delta$, and check that the type declaration has not been made inside an applicative functor by requiring that the environment Δ contains no universally quantified types. The check for positivity is made when exiting a functor definition, it is not visible here. We then return the singleton map $(\alpha \mapsto A.t)$. The general version of the rule A-Decl-Anchor considers a declaration for a possibly higher-order *unmarked* type expression φ:

A-Decl-Anchor
$$\frac{\varphi \notin \mathsf{dom}(\theta) \qquad \varphi \in \Delta \qquad \mathsf{args}(\Delta) = \overline{\alpha_1}\,; \ldots\, \overline{\alpha_n}}{\Gamma \cdot \Delta\,; \theta \vdash \mathsf{type}\ t = \varphi\langle\overline{\alpha_1}\rangle \ldots \langle\overline{\alpha_n}\rangle \xhookrightarrow{A} \mathtt{type}\ t = A.t : (\varphi \mapsto A.t)}$$

This rule only applies when φ is both unmarked and applied to exactly the sequence $\mathsf{args}(\Delta)$ of abstract types in Δ (which necessarily follow φ). Anchoring fails if one of the conditions does not hold. The process could be made more permissive or more restrictive by tweaking this rule.

Updates The anchoring map θ is *updated* in the two places where access paths to types must be changed as we exit scopes: (1) in Rule A-Sig-StrPath, locally anchored abstract types are made available through the path P and (2) in Rule A-Sig-FctApp, paths to anchored types of θ are point-wise abstracted over the functor parameter Y in the returned map $\lambda Y.\theta$. The list $\overline{\alpha}$ is marked as existentially quantified in the left-hand side premise (when anchoring $\mathcal{C}_a$)

$$\textsc{A-Env-Decl}\quad \frac{\Gamma\,;\theta \vdash \mathcal{D} \xhookrightarrow{A} \mathtt{D} : \theta'}{\Gamma, A.\mathcal{D} \hookrightarrow \theta \uplus \theta}
\qquad
\textsc{A-Env-Arg}\quad \frac{\Gamma \cdot \overline{\alpha}\,;\theta \vdash \mathcal{C}_a \xhookrightarrow{Y} \mathtt{S}_a : \theta_a \qquad \mathsf{dom}(\theta_a) = \overline{\alpha}}{\Gamma, \overline{\alpha}, (Y : \mathcal{C}) \hookrightarrow \theta \uplus \theta_a}
\qquad
\textsc{A-Env-Abs}\quad \frac{\Gamma \hookrightarrow \theta}{\Gamma, \overline{\alpha} \hookrightarrow \theta}$$

(a) Anchoring of environment

$$\textsc{A-Sig-StrPath}\quad \frac{\Gamma\,;\theta \vdash \overline{\mathcal{D}} \xhookrightarrow{A} \overline{\mathtt{D}} : \theta \qquad A \notin \Gamma}{\Gamma\,;\theta \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \xhookrightarrow{P} \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end} : \theta[A \mapsto P]}
\qquad
\textsc{A-Sig-StrNone}\quad \frac{\Gamma\,;\theta \vdash \overline{\mathcal{D}} \xhookrightarrow{A} \overline{\mathtt{D}} : \theta \qquad A \notin \Gamma \qquad \mathsf{dom}(\theta) = \overline{\alpha}}{\Gamma\,;\theta \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \hookrightarrow \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end} : (\overline{\alpha} \mapsto \_\,)}$$

$$\textsc{A-Sig-FctGen}\quad \frac{\Gamma \cdot \overline{\alpha}\,;\theta \vdash \mathcal{C} \hookrightarrow \mathtt{S} : (\overline{\alpha} \mapsto \_\,)}{\Gamma\,;\theta \vdash () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \xhookrightarrow{A.\mathit{Val}} () \to \mathtt{S} : \varnothing}
\qquad
\textsc{A-Sig-FctApp}\quad \frac{\begin{array}{c}\Gamma \cdot \overline{\alpha}\,;\theta \vdash \mathcal{C}_a \xhookrightarrow{Y} \mathtt{S}_a : \theta_a \qquad \mathsf{dom}(\theta_a) = \overline{\alpha}\\ \Gamma, \overline{\alpha}, Y : \mathcal{C}_a\,;\theta \uplus \theta_a \vdash \mathcal{C} \xhookrightarrow{A.\mathit{Val}(Y)} \mathtt{S} : \theta\end{array}}{\Gamma\,;\theta \vdash \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \xhookrightarrow{A.\mathit{Val}} (Y : \mathtt{S}_a) \to \mathtt{S} : \lambda Y.\theta}$$

(b) Anchoring of signatures

$$\textsc{A-Decl-Val}\quad \frac{\Gamma\,;\theta \vdash \tau \hookrightarrow \mathtt{u}}{\Gamma\,;\theta \vdash \mathsf{val}\ x : \tau \xhookrightarrow{A} (\mathtt{val}\,x : \mathtt{u}) : \varnothing}
\qquad
\textsc{A-Decl-Anchor}\quad \frac{\varphi \notin \mathsf{dom}(\theta) \qquad \varphi \in \Delta \qquad \mathsf{args}(\Delta) = \overline{\alpha_1}\,;\ \dots\ \overline{\alpha_n}}{\Gamma \cdot \Delta\,;\theta \vdash \mathsf{type}\ t = \varphi\langle\overline{\alpha_1}\rangle \dots \langle\overline{\alpha_n}\rangle \xhookrightarrow{A} \mathtt{type}\,t = A.t : (\varphi \mapsto A.t)}$$

$$\textsc{A-Decl-Type}\quad \frac{\Gamma\,;\theta \vdash \tau \hookrightarrow \mathtt{u}}{\Gamma\,;\theta \vdash \mathsf{type}\ t = \tau \xhookrightarrow{A} \mathtt{type}\,t = \mathtt{u} : \varnothing}
\qquad
\textsc{A-Decl-Mod}\quad \frac{\Gamma\,;\theta \vdash \mathcal{C} \xhookrightarrow{A.X} \mathtt{S} : \theta}{\Gamma\,;\theta \vdash \mathsf{module}\ X : \mathcal{C} \xhookrightarrow{A} \mathtt{module}\,X : \mathtt{S} : A.\theta}$$

$$\textsc{A-Decl-Empty}\quad \Gamma\,;\theta \vdash \varnothing \xhookrightarrow{A} \varnothing : \varnothing
\qquad
\textsc{A-Decl-ModType}\quad \frac{\Gamma, \overline{\alpha}\,;\theta \vdash \mathcal{C} \hookrightarrow \mathtt{S} : (\overline{\alpha} \mapsto \_\,)}{\Gamma\,;\theta \vdash (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}) \xhookrightarrow{A} \mathtt{module\ type}\ T = \mathtt{S} : \varnothing}$$

$$\textsc{A-Decl-Seq}\quad \frac{\Gamma\,;\theta \vdash \mathcal{D} \xhookrightarrow{A} \mathtt{D} : \theta_1 \qquad \Gamma, A.\mathcal{D}\,;\theta \uplus \theta_1 \vdash \overline{\mathcal{D}} \xhookrightarrow{A} \overline{\mathtt{D}} : \theta_2}{\Gamma\,;\theta \vdash \mathcal{D}, \overline{\mathcal{D}} \xhookrightarrow{A} \mathtt{D}, \overline{\mathtt{D}} : \theta_1 \uplus \theta_2}$$

(c) Anchoring of declarations

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

(d) Anchoring of types

Figure 16: Anchoring rules

and universally quantified in the right-hand side one (when anchoring $\mathcal{C}$).

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

$$\frac{\Gamma \cdot !\overline{\alpha}\,;\theta \vdash \mathcal{C}_a \hookrightarrow \mathtt{S}_a : \theta_a \qquad \mathsf{dom}(\theta) = \overline{\alpha} \qquad \Gamma, ?\overline{\alpha}, Y : \mathcal{C}_a\,;\theta \uplus \theta_a \vdash \mathcal{C} \xhookrightarrow{A.\mathit{Val}(Y)} \mathtt{S} : \theta}{\Gamma\,;\theta \vdash \forall\overline{\alpha}.\mathcal{C}_a \rightarrow \mathcal{C} \xhookrightarrow{A.\mathit{Val}} (Y : \mathtt{S}_a) \rightarrow \mathtt{S} : \lambda Y.\theta}\ \textsc{A-Sig-FctApp}$$

By contrast, the anchoring map of the body of a generative functor is thrown away (Rule A-Sig-FctGen), as generative functors cannot appear in paths, [1] and a barrier is added in the premise, as the body cannot capture types defined outside of the functor.

Path resolution Finally, the anchoring map is used for anchoring an M^{ω}-types τ into a qualified type – which requires finding a suitable path to access the anchoring point – by the following rule:

$$\frac{\begin{array}{c}\tau = \varphi\langle\tau_1 \ldots\rangle \ldots \langle\tau_n \ldots\rangle \qquad \theta(\varphi) = \lambda Y_k. \ldots \lambda Y_n.\ P.t \\ \forall i \in [\![k,n]\!]\,.\ \Gamma\,;\theta \vdash \tau_i \hookrightarrow P_i.id \qquad \mathtt{u} = \theta(\varphi)(P_k)\ldots(P_n) \qquad \Gamma \vdash \mathtt{u} : \tau\end{array}}{\Gamma\,;\theta \vdash \tau \hookrightarrow \mathtt{u}}\ \textsc{A-Type-Application}$$

Here, we consider types stripped of their marks, e.g., by reducing $\varphi^{\dagger}$ to φ. The rule A-Type-Application applies when τ is of the form $\varphi\langle\tau_1 \ldots\rangle \ldots \langle\tau_n \ldots\rangle$ and $\theta(\varphi)$ is a (mathematical) function of the form $\lambda Y_k. \ldots \lambda Y_n.P.t$. Types τ_k to τ_n may only be resolved to identity tags $P_k.id$ to $P_n.id$. Then, theThis rule is designed to allow anchoring of a type τ that is abstract over a certain number of parameters (here, $n-k+1$) even if τ is actually applied to more parameters (here, n). This comes from the fact that source signatures do not display the depth of enclosing applicative functors. The resulting type $\mathtt{u}$ is the path (resulting from the mathematical application) $\theta(\varphi)(P_k)\ldots(P_n)$. However, $\mathtt{u}$ must be re-typechecked to ensure that paths occurring in τ only contain valid functor applications[2] and that it returns the same type as the input type τ.

Example 3.3.5. *We illustrate the anchoring process of a type inside an applicative functor. We give the corresponding anchoring map at the end of the line. For the sake of readability, we do not write the identity of the functor (nor its body) and we write tagged signatures with*

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

[2]Indeed, it can happen that a module `X` is lost, while a transparent ascription `X'` is kept. The types resulting from a functor application `F(X).t` may not be anchorable as `F(X').t` if `X'` lacks certain fields.

pairs.

1	$\exists^{\nabla}\varphi, \alpha_{id_1}, \alpha_{id_2}.\mathsf{sig}$	$\varnothing$
2	$\quad$ module F :	$\varnothing$
3	$\quad\ \forall\alpha_{id}, \alpha.(\alpha_{id}, \mathsf{sig\ type}\ t = \alpha\ \mathsf{end}) \to$	$(\alpha_{id} \mapsto Y.id,\ \alpha \mapsto Y.Val.t)$
4	$\quad\quad$ sig type $t = \varphi(\alpha)$ end	$(\alpha_{id} \mapsto Y.id,\ \alpha \mapsto Y.Val.t,\ \varphi \mapsto C.t)$
5		$(\varphi \mapsto \lambda Y.A.F(Y).t)$
6	$\quad$ module $X_1 : (\alpha_{id_1}, \mathsf{sig\ type}\ t = \mathtt{int}\ \mathsf{end})$	$(\varphi \mapsto \lambda Y.A.F(Y).t,\ \alpha_{id_1} \mapsto A.X_1.id)$
7	$\quad$ module $X_2 : (\alpha_{id_2}, \mathsf{sig\ type}\ t = \mathtt{bool}\ \mathsf{end})$	$(\varphi \mapsto \lambda Y.A.F(Y).t,\ \alpha_{id_1} \mapsto A.X_1.id,\ \alpha_{id_2} \mapsto A.X_2.id)$
8	$\quad$ type $t = \varphi\ \langle\alpha_{id_1}, \mathtt{int}\rangle$	$(\varphi \mapsto \lambda Y.A.F(Y).t,\ \alpha_{id_1} \mapsto A.X_1.id,\ \alpha_{id_2} \mapsto A.X_2.id)$
9	$\quad$ type $u = \varphi\ \langle\alpha_{id_2}, \mathtt{bool}\rangle$	$(\varphi \mapsto \lambda Y.A.F(Y).t,\ \alpha_{id_1} \mapsto A.X_1.id,\ \alpha_{id_2} \mapsto A.X_2.id)$
10	end	

1	$\mathtt{sig}_A$
2	$\quad\mathtt{module}\ F : (Y : \mathtt{sig}_B\ \mathtt{type}\ t = B.t\ \mathtt{end}) \to \mathtt{sig}_C\ \mathtt{type}\ t = C.t\ \mathtt{end}$
3	$\quad\mathtt{module}\ X_1 : \mathtt{sig}_{A_1}\ \mathtt{type}\ id = A_1.id\ \ \mathtt{module}\ Val : (\mathtt{sig\ type}\ t = \mathtt{int\ end})\ \mathtt{end}$
4	$\quad\mathtt{module}\ X_2 : \mathtt{sig}_{A_2}\ \mathtt{type}\ id = A_2.id\ \ \mathtt{module}\ Val : (\mathtt{sig\ type}\ t = \mathtt{bool\ end})\ \mathtt{end}$
5	$\quad\mathtt{type}\ t = A.F(A.X_1).t$
6	$\quad\mathtt{type}\ u = A.F(A.X_2).t$
7	$\mathtt{end}$

We have the following steps:

3. *When anchoring the body of the functor, we have an anchoring of the universally quantified variables α_{id}, α.*
4. *Then, an anchoring point for φ is found, initially bound to the local name $C.t$.*
5. *When exiting the functor, the anchoring of α_{id}, α is thrown, as the variables are no longer accessible, and the anchoring of φ is updated to a parameterized path $\lambda Y.A.F(Y).t$*
6. *The identity tag α_{id_1} is anchored to $X_1.id$*
7. *The identity tag α_{id_2} is anchored to $X_2.id$*

Finally, the anchoring map is used at lines (8) and (9) to rebuild the qualified types.

Untagging

The first step of anchoring returns a tagged source signature S. It remains to remove the tags, i.e., to return a signature S' with the `id` and `Val` fields stripped of S, recursively, but expressing the same sharing using concrete signatures. This is defined as a judgment $\Gamma \vdash \mathsf{S} \hookrightarrow \mathsf{S}'$. The two interesting rules are for untagging structural signatures:

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

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

3.3.3 Properties of Anchoring

In this section we state and prove the meta-theoretic properties of anchoring.

Anchoring of elaborated signatures Conceptually, anchoring and elaboration of signatures are inverse of each other, which we could write as "[Elaboration ∘ Anchoring = Id]". There is however a caveat, as typechecking is not injective: several source signatures can express the same type sharing information. Therefore, we may quotient source signatures by the equivalence induced by M^ω typing (using the normal type equivalence of M^ω). We define the equivalence of source signatures as:

$$\Gamma \vdash \mathtt{S} \approx \mathtt{S}' \ \triangleq\ \Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C} \ \wedge\ \Gamma \vdash \mathtt{S}' : \lambda\overline{\alpha}.\mathcal{C}' \ \wedge\ \lambda\overline{\alpha}.\mathcal{C} \equiv \lambda\overline{\alpha}'.\mathcal{C}'.$$

Anchoring produces a signature where all type equalities have been aggressively inlined.

Theorem 2: Elaboration ∘ Anchoring = Id

The elaboration of source signatures produces anchorable signatures. Given a source signature $\mathtt{S}$, typing environment Γ and anchoring map θ such that:

$$\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C} \ \wedge\ \Gamma \hookrightarrow \theta$$

Then, there exists a source signature $\mathtt{S}'$ and an anchoring map θ' such that:

$$\Gamma \cdot \overline{\alpha}\,;\theta \vdash \mathcal{C} \hookrightarrow \mathtt{S}' : (\overline{\alpha} \mapsto \_\ )$$

The signature $\mathtt{S}'$ might no be syntactically equal to $\mathtt{S}$, but they are in the same equivalence class with regard to M^ω typing:

$$\Gamma \vdash \mathtt{S} \approx \mathtt{S}'$$

Elaboration of anchored signatures The other direction would be to show that, starting with a M^ω signature $\mathcal{C}$, finding an anchored signature $\mathtt{S}$ and elaborating it back to M^ω would give a signature $\mathcal{C}'$ that is equivalent to $\mathcal{C}$, the signature we started with. This could be seen as "Anchoring ∘ Elaboration = Id". Yet, there is a subtlety. M^ω signatures can express the fact that they are *inside an applicative functor*, as their abstract types are applied to a certain set of universally quantified type variables. This does not appear in source signatures, which are the same whatever the number of enclosing applicative functors. Therefore, we only have an equivalence "up to *skolemization*".

Theorem 3: Anchoring ∘ Elaboration ≃ Id

Given a M^ω signature $\mathcal{C}$, a source signature $\mathtt{S}$, a typing environment and anchoring maps such that:

$$\Gamma \cdot \Delta\,;\theta \vdash \mathcal{C} \hookrightarrow \mathtt{S} : \theta \ \wedge\ \mathsf{dom}(\theta) = \overline{\alpha} \ \wedge$$

Typing back the anchoring gives the original signature, up to re-skolemization of current universally quantified types.

$$\Gamma \hookrightarrow \theta \implies \Gamma \vdash \mathtt{S} : \lambda\overline{\beta}.\mathcal{C}' \ \wedge\ \mathcal{C}'\big[\overline{\beta} \mapsto \overline{\alpha}(\mathsf{args}(\Delta))\big] = \mathcal{C}$$

Untagging Finally, the untagging phase is not surprising, and is the inverse of tagging. The result relies on a notion of *tag-wellformed source signatures*: signatures that have identity tags at each module binding and functor body. This is trivially maintained by typing and anchoring. We have the following result:

Theorem 4: Untagging

Given a tag-wellformed signature $\mathtt{S}$, untagging it and tagging it back yields an equivalent signature:

$$\Gamma \vdash \mathtt{S} \hookrightarrow \mathtt{S}' \implies \Gamma \vdash \mathtt{S} \approx [\![\mathtt{S}']\!]$$

The rest of this section is dedicated to the corresponding proofs.

Proof of Theorem 3

We show the result by mutual induction on the anchoring of signatures and declarations. The induction hypothesis is :

$$\begin{aligned}
&\Gamma \cdot \Delta \,;\theta \vdash \mathcal{C} \hookrightarrow \mathtt{S} : \theta \ \wedge\ \mathsf{dom}(\theta) = \overline{\alpha} \ \wedge\ \Gamma \hookrightarrow \theta \\
&\qquad \implies \Gamma \cdot \Delta \vdash \mathtt{S} : \lambda\overline{\beta}.\mathcal{C}' \ \wedge\ \mathcal{C}'\left[\overline{\beta \mapsto \overline{\alpha}(\mathsf{args}(\Delta))}\right] = \mathcal{C} \\
&\Gamma \cdot \Delta \,;\theta \vdash \overline{\mathcal{D}} \xhookrightarrow{A} \overline{\mathtt{D}} : \theta \ \wedge\ \mathsf{dom}(\theta) = \overline{\alpha} \ \wedge\ \Gamma \hookrightarrow \theta \\
&\qquad \implies \Gamma \cdot \Delta \vdash \overline{\mathtt{D}} : \lambda\overline{\beta}.\overline{\mathcal{D}}' \ \wedge\ \overline{\mathcal{D}}'\left[\overline{\beta \mapsto \overline{\alpha}(\mathsf{args}(\Delta))}\right] = \overline{\mathcal{D}}
\end{aligned}$$

The proof is by structural induction on the anchoring derivation.

- A-Sig-FctGen: The conclusion of the rule is the signature anchoring judgment

  $$\Gamma \cdot \Delta \,;\theta \vdash () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \xhookrightarrow{A.Val} () \to \mathtt{S} : \varnothing$$

  Since the domain of the local map is empty, we just have to show

  $$\Gamma \cdot \Delta \vdash () \to \mathtt{S} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}\ (\mathbf{1})$$

  The premise of the rule is $\Gamma \cdot \Delta \cdot !\overline{\alpha}\,;\theta \vdash \mathcal{C} \hookrightarrow \mathtt{S} : (\overline{\alpha \mapsto \_})$. By induction hypothesis, we have $\Gamma \cdot \Delta \vdash \mathtt{S} : \lambda\overline{\beta}.\mathcal{C}'$ (**2**) and $\mathcal{C}'\left[\overline{\beta \mapsto \alpha}\right] = \mathcal{C}$, since $\mathsf{args}(!\overline{\alpha})$ is empty, that is $\lambda\overline{\beta}.\mathcal{C}' = \lambda\overline{\alpha}.\mathcal{C}$.

  Then (1) follows by M-Sig-GenFct applied to (2).

- A-Sig-FctApp: The rule is

  $$\frac{\Gamma \cdot \Delta \cdot !\overline{\alpha}\,;\theta \vdash \mathcal{C}_a \xhookrightarrow{Y} \mathtt{S}_a : \theta_a\ (\mathbf{1}) \qquad \Gamma \cdot \Delta, ?\overline{\alpha}, Y : \mathcal{C}_a\,;\theta \uplus \theta_a \vdash \mathcal{C} \xhookrightarrow{A.Val(Y)} \mathtt{S} : \theta\ (\mathbf{2})}{\Gamma \cdot \Delta\,;\theta \vdash \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \xhookrightarrow{A.Val} (Y : \mathtt{S}_a) \to \mathtt{S} : \lambda Y.\theta}$$

  with $\mathsf{dom}(\theta_a) = \overline{\alpha}$. By IH applied to (1), we have $\Gamma \cdot \Delta \vdash \mathtt{S}_a : \lambda\overline{\alpha}.\mathcal{C}_a$ (**3**) since $\mathsf{args}(\overline{\alpha}, \overline{\alpha})$ is empty. Let $\overline{\gamma}$ be $\mathsf{dom}(\theta)$. By IH applied to (2), we have $\Gamma \cdot \Delta, ?\overline{\alpha}, Y : \mathcal{C}_a \vdash \mathtt{S} : \lambda\overline{\beta}.\mathcal{C}'$ (**4**) with $\mathcal{C}'\left[\overline{\beta \mapsto \overline{\gamma}(\mathsf{args}(\Delta), \overline{\alpha}))}\right] = \mathcal{C}$ (**5**), since $\mathsf{args}(\Delta, ?\overline{\alpha})$ is equal to $\mathsf{args}(\Delta), \overline{\alpha}$. By rule M-Typ-Sig-AppFct applied to (3) and (4), we have:

  $$\Gamma \cdot \Delta \vdash (Y : \mathtt{S}_a) \to \mathtt{S} : \lambda\overline{\beta'}.\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}'\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]$$

  It remains to check:

  $$\left(\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}'\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]\right)\left[\overline{\beta}' \mapsto \overline{\gamma}(\mathsf{args}(\Delta))\right] \equiv \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}$$

  which follows from (5) by composition of substitutions, since β' does not occur free in $\mathcal{C}_a$.

- A-Sig-StrPath and A-Sig-StrNone are immediate by induction.
- For A-Decl-Val, A-Decl-Type, we may easily show that $\Gamma ; \theta \vdash \tau \hookrightarrow \mathtt{u}$ implies $\Gamma \vdash \mathtt{u} : \tau$.

  In both cases, the conclusion is of the form $\Gamma ; \theta \vdash \mathtt{D} \overset{A}{\hookrightarrow} \mathcal{D} : \varnothing$, it suffices to show $\Gamma \cdot \Delta \vdash \mathtt{D} : \mathcal{D}$ (**1**). In the case of A-Decl-Val, the conclusion is $\Gamma ; \theta \vdash \mathsf{val}\, x : \tau \overset{A}{\hookrightarrow} (\mathtt{val}\, x : \mathtt{u}) : \varnothing$ and the premise is $\Gamma \vdash \mathtt{u} : \tau$, which implies $\Gamma \vdash \mathtt{u} : \tau$. Then, (1) follows by M-Typ-Decl-Val.

  The case of A-Decl-Type is similar.
- A-Decl-Mod, A-Decl-ModType are immediate by induction.
- A-Decl-Anchor: the conclusion of the rule is:

  $$\Gamma \cdot \Delta ; \theta \vdash \mathsf{type}\, t = \varphi\langle\overline{\alpha_1}\rangle \dots \langle\overline{\alpha_n}\rangle \overset{A}{\hookrightarrow} \mathsf{type}\, t = A.t : (\varphi \mapsto A.t)$$

  By Rule M-Typ-Decl-TypeAbs, we have $\Gamma \cdot \Delta \vdash (\mathsf{type}\, t = A.t) : \lambda\alpha.\mathsf{type}\, t = \alpha$. From the premises, we know that $\varphi \in \Delta$ and $\mathsf{args}(\Delta) = \overline{\alpha_1} ; \dots \overline{\alpha_n}$. Hence, $(\mathsf{type}\ t = \alpha)[\varphi \mapsto \varphi\, \overline{\alpha_1} ; \dots \overline{\alpha_n}]$ is equal to $\mathsf{type}\ t = \varphi\langle\overline{\alpha_1}\rangle \dots \langle\overline{\alpha_n}\rangle$, which is exactly the declaration we started with.
- A-Decl-Seq: The Rule is :

  $$\text{A-Decl-Seq}\quad \frac{\Gamma ; \theta \vdash \mathcal{D}_1 \overset{A}{\hookrightarrow} \mathtt{D}_1 : \theta_1 \qquad \Gamma, A.\mathcal{D}_1 ; \theta \uplus \theta_1 \vdash \overline{\mathcal{D}_2} \overset{A}{\hookrightarrow} \overline{\mathtt{D}_2} : \theta_2}{\Gamma ; \theta \vdash \mathcal{D}_1, \overline{\mathcal{D}_2} \overset{A}{\hookrightarrow} \mathtt{D}_1, \overline{\mathtt{D}_2} : \theta_1 \uplus \theta_2}$$

  Writing the domain of $\theta_1 \uplus \theta_2$ as $\mathsf{dom}(\theta_1 \uplus \theta_2) = (\overline{\alpha}_1, \overline{\alpha}_2)$, we need to show that there exists $\mathcal{D}'_1$ and $\overline{\mathcal{D}'_2}$ such that:

  $$\Gamma \cdot \Delta \vdash \mathtt{D}_1, \overline{\mathtt{D}_2} : \lambda\overline{\alpha}_1, \overline{\alpha}_2.\mathcal{D}'_1, \overline{\mathcal{D}'_2} \qquad (\mathbf{1})$$
  $$\mathcal{D}'_1, \overline{\mathcal{D}'_2}\big[\overline{\beta}_1, \overline{\beta}_2 \mapsto \overline{\alpha}_1(\mathsf{args}(\Delta)), \overline{\alpha}_2(\mathsf{args}(\Delta))\big] = \mathcal{D}_1, \overline{\mathcal{D}_2} \qquad (\mathbf{2})$$

  We have by induction hypothesis:

  $$\Gamma \cdot \Delta \vdash \mathtt{D}_1 : \lambda\overline{\beta}_1.\mathcal{D}'_1 \quad \wedge \quad \mathcal{D}'_1\big[\overline{\beta}_1 \mapsto \overline{\alpha}_1(\mathsf{args}(\Delta))\big] = \mathcal{D}_1$$
  $$\Gamma \cdot \Delta, A.\mathcal{D} \vdash \overline{\mathtt{D}_2} : \lambda\overline{\beta}_2.\overline{\mathcal{D}'_2} \quad \wedge \quad \overline{\mathcal{D}'_2}\big[\overline{\beta}_2 \mapsto \overline{\alpha}_2(\mathsf{args}(\Delta, A.\mathcal{D}))\big] = \overline{\mathcal{D}_2}$$

  We introduce $\overline{\mathcal{D}''_2} = \overline{\mathcal{D}'_2}\big[\overline{\alpha}_1(\mathsf{args}(\Delta)) \mapsto \overline{\beta}_1\big]$.

  – Proof of (1)
  We use the Rule M-Typ-Decl-Seq:

  $$\text{M-Typ-Decl-Seq}\quad \frac{\Gamma \vdash_A \mathtt{D}_1 : \lambda\overline{\alpha}_1.\mathcal{D}_1 (\mathbf{3}) \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D}_1 \vdash_A \overline{\mathtt{D}_2} : \lambda\overline{\alpha}.\overline{\mathcal{D}_2} (\mathbf{4})}{\Gamma \vdash_A \mathtt{D}_1, \overline{\mathtt{D}_2} : \lambda\overline{\alpha}_1\, \overline{\alpha}.\ \mathcal{D}_1, \overline{\mathcal{D}_2}}$$

  The first premise (3) is immediate by induction hypothesis. For the second one (4), we show by induction that :

  $$\Gamma \cdot \Delta, A.\mathcal{D}_1 \vdash \overline{\mathtt{D}_2} : \lambda\overline{\beta}_2.\overline{\mathcal{D}'_2} \implies \Gamma \cdot \Delta, \overline{\beta}_1, A.\mathcal{D}'_1 \vdash \overline{\mathtt{D}_2} : \lambda\overline{\beta}_2.\overline{\mathcal{D}''_2}$$

  – Proof of (2)
  By definition, we have

  $$\mathsf{args}(\Delta, A.\mathcal{D}) = \mathsf{args}(\Delta)$$

As the declaration $\mathcal{D}'_1$ is wellformed in an environment that does not contain the $\overline{\beta}_2$, substituting for them does not change anything. Therefore, we have:

$$\begin{aligned}
&\mathcal{D}'_1, \overline{\mathcal{D}''_2}\left[\overline{\beta}_1, \overline{\beta}_2 \mapsto \overline{\alpha}_1(\mathsf{args}(\Delta)), \overline{\alpha}_2(\mathsf{args}(\Delta))\right]\\
&= \mathcal{D}'_1\left[\overline{\beta}_1 \mapsto \overline{\alpha}_1(\mathsf{args}(\Delta))\right], (\overline{\mathcal{D}''_2}\left[\overline{\beta}_1 \mapsto \overline{\alpha}_1(\mathsf{args}(\Delta))\right])\left[\overline{\beta}_2 \mapsto \overline{\alpha}_2(\mathsf{args}(\Delta))\right]\\
&= \mathcal{D}_1, \overline{\mathcal{D}'_2}\left[\overline{\beta}_2 \mapsto \overline{\alpha}_2(\mathsf{args}(\Delta))\right]\\
&= \mathcal{D}_1, \overline{\mathcal{D}'_2}\left[\overline{\beta}_2 \mapsto \overline{\alpha}_2(\mathsf{args}(\Delta, A.\mathcal{D}))\right]\\
&= \mathcal{D}_1, \overline{\mathcal{D}_2}
\end{aligned}$$

This conclude this case.

This concludes the proof.

Proof of Theorem 2

We first prove that typed signatures are anchorable, then that the result is equivalent to the signature we started with. We start with the following lemma:

Lemma 5 (Skolemization lemma)**.** *Skolemizing a signature does not change its anchoring:*

$$\Gamma, ?\overline{\alpha} \cdot !\overline{\beta}, \Delta \,;\theta \vdash \mathcal{C} \overset{P}{\hookrightarrow} \mathsf{S} : \theta \implies \Gamma \cdot !\overline{\beta'}, ?\overline{\alpha}, \Delta \,;\theta \vdash \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right] \overset{P}{\hookrightarrow} \mathsf{S} : \theta$$

Proof. The key observation is that, from the source syntax, accessing *local* types does take into account the depth of enclosing applicative functors. The two key rules are:

- A-Decl-Anchor: when anchoring a type, the number of (list of) universally quantified parameters n does not appear in the anchoring (nor in the resulting declaration).
- A-Type-Application: similarly, the first type arguments τ_1 to τ_k are ignored. If the anchoring point $\theta(\varphi)$ is parameterized by only $n-k$ arguments, adding another argument in front does not change the anchoring.

□

Coming back to the proof of Theorem 2, we first show that typed signatures are anchorable:

$$\Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C} \wedge \Gamma \hookrightarrow \theta \implies \exists \mathsf{S}'.\, \Gamma \cdot !\overline{\alpha}\,;\theta \vdash \mathcal{C} \hookrightarrow \mathsf{S}' : \overline{\alpha} \mapsto \_$$

We proceed by induction on the typing derivation of signatures and declarations:

- M-Typ-Sig-ModType and M-Typ-Sig-ModType are base-cases of the induction. We easily show that the anchoring of the environment $\Gamma \hookrightarrow \theta$ implies that stored module types are anchorable.
- M-Typ-Sig-GenFct and M-Typ-Sig-Str are immediate.
- M-Typ-Sig-AppFct: the induction hypothesis gives us that the signature of the core of the functor is anchorable before the substitution of $\overline{\beta}$ for $\overline{\beta'}(\overline{\alpha})$. Here, we use the skolemization lemma.
- M-Typ-Sig-Con: we first note that all type variables $\overline{\alpha}$ appear in $\mathcal{C}'$. Then, as subtyping between types is only defined by equality, all the types expressions $\overline{\tau}$ appear in $\mathcal{C}$. As $\mathcal{C}$ is anchorable, so are all its type components, making $\mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]$ anchorable.

Finally, the equivalence between source signatures is an immediate consequence of Theorem 3.

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

Figure 17: Syntax of F^ω

3.4 The Foundations: F^ω Elaboration

The M^ω system is designed to offer a standard, standalone, and expressive approach to the typing of ML modules, while hiding the complexity and artifacts of its encoding in F^ω. Yet, the elaboration of module expressions and signatures of M^ω in F^ω, which we now present, served as a basis for the design of M^ω and still shines a new light on its internal mechanisms. It is also used as a proof of type soundness. This elaboration is largely based on the work of Rossberg et al. [2014], but differs in a key manner for the treatment of abstract types defined inside applicative functors. A main contribution is the introduction of *transparent* existential types, an intermediate between the standard existential types, called *opaque* existential types, and the absence of abstraction. They bring the treatment of applicative and generative functors closer, and significantly simplify the elaboration.

In Section 3.4.1, we introduce F^ω with primitive record, existential types and kind polymorphism. In Section 3.4.2, we present the elaboration of M^ω signatures as F^ω types. In Section 3.4.3, we focus on the key mechanism of *repacking* that allows to extend the scope of an existential type. Repacking is the justification of the lifting of existential types through record types. In Section 3.4.4, we introduce transparent existential types to justify the skolemization of existential types through record types. In Section 3.4.5, we show that transparent existential types can actually be encoded *internally* as an F^ω library. In Section 3.4.6, we present the elaboration of modules as F^ω terms.

3.4.1 F^ω with Kind Polymorphism

We use a variant of explicitly typed F^ω with primitive records (including record concatenation), existential types, and predicative kind polymorphism. While primitive records and existential types are standard, kind polymorphism is less common. Predicativity of kind polymorphism is not needed for type soundness. However, it ensures coherence of types used as a logic, that is, it prevents typing terms with the empty type $\forall(\alpha : \star).\,\alpha$, whose evaluation would not terminate. For that purpose, kinds are split into two categories: large and small. Polymorphic kinds, which are large, can only be instantiated by small kinds, which in turn do not contain polymorphic kinds. In our setting, kind polymorphism is not essential, as it is only used to internalize the encoding of transparent existential types as F^ω-terms in Section 3.4.5. Alternatively, we could have assumed a family of transparent existential type operators indexed by small kinds, so as to never use large kinds, moving part of the encoding at the meta-level.

The syntax of F^ω is given in Figure 17. Typing rules are standard and given in Figure 18. Type equivalence, defined by $\alpha\beta$-conversion and reordering of record fields, is also standard and omitted. We use letters τ and e to range over types and expressions to distinguish them from types $\mathtt{u}$ and expressions $\mathtt{e}$ of the core language, even though these should actually be

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

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

(a) Environment checking

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

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

$$\frac{\Gamma, \omega \vdash \tau : \kappa}{\Gamma \vdash \Lambda \omega . \tau : \forall \omega . \kappa} \qquad \frac{\Gamma \vdash \tau : \forall \omega . \kappa \qquad \Gamma \vdash \varsigma}{\Gamma \vdash \tau \, \varsigma : \kappa[\omega \mapsto \varsigma]}$$

(b) Type checking

$$\textsc{F-Var}\ \frac{\vdash \Gamma \qquad x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{F-Abs}\ \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda(x : \tau) . e : \tau \to \tau'} \qquad \textsc{F-App}\ \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 \, e_2 : \tau}$$

$$\textsc{F-Record}\ \frac{\overline{\Gamma \vdash e : \tau} \qquad \#(\overline{\ell})}{\Gamma \vdash \{\overline{\ell = e}\} : \{\overline{\ell : \tau}\}} \qquad \textsc{F-Proj}\ \frac{\Gamma \vdash e : \{\ell : \tau, \overline{\ell_1 : \tau_1}\}}{\Gamma \vdash e.\ell : \tau} \qquad \textsc{F-Append}\ \frac{\Gamma \vdash e_1 : \{\overline{\ell_1 : \tau_1}\} \qquad \Gamma \vdash e_2 : \{\overline{\ell_2 : \tau_2}\} \qquad \overline{\ell_1} \# \overline{\ell_2}}{\Gamma \vdash e_1 \mathbin{@} e_2 : \{\overline{\ell_1 : \tau_1}.\overline{\ell_2 : \tau_2}\}}$$

$$\textsc{F-Tapp}\ \frac{\Gamma \vdash e : \forall(\alpha : \kappa) . \sigma \qquad \Gamma \vdash \tau : \kappa}{\Gamma \vdash e \, \tau : \sigma[\tau \mapsto \alpha]} \qquad \textsc{F-Kapp}\ \frac{\Gamma \vdash e : \forall \omega . \tau \qquad \Gamma \vdash \varsigma}{\Gamma \vdash e \, \varsigma : \tau[\omega \mapsto \varsigma]} \qquad \textsc{F-Tabs}\ \frac{\Gamma, \alpha : \kappa \vdash e : \tau}{\Gamma \vdash \lambda(\alpha : \kappa) . e : \forall(\alpha : \kappa) . \tau}$$

$$\textsc{F-Kabs}\ \frac{\Gamma, \omega \vdash e : \tau}{\Gamma \vdash \Lambda \omega . e : \forall \omega . \tau} \qquad \textsc{F-Pack}\ \frac{\Gamma \vdash \exists^{\blacktriangledown}(\alpha : \kappa) . \sigma : \star \qquad \Gamma \vdash \tau : \kappa \qquad \Gamma \vdash e : \sigma[\tau \mapsto \alpha]}{\Gamma \vdash \mathsf{pack}\ \langle \tau, e \rangle\ \mathsf{as}\ \exists^{\blacktriangledown}(\alpha : \kappa) . \sigma : \exists^{\blacktriangledown}(\alpha : \kappa) . \sigma}$$

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

(c) Expression typing

Figure 18: Typing rules of F^ω

seen as a subset of τ and e. We consider F^ω to be explicitly typed and explicitly kinded. As a convention, we use a wildcard "_" when a type annotation is unambiguously determined by an immediate subexpression and may be omitted. This is just a syntactic convenience to avoid redundant type information and improve readability, but the underlying terms should always be understood as explicitly-typed F^ω terms. We write ω for kind variables, α and β for type variables of any kind, and φ and ψ for type variables known to be of higher-order kinds. Application of expressions $e\,\varsigma$ and types $\tau\,\varsigma$ to kinds are restricted to small kinds ς. In expressions and type expressions, we actually write kinds κ (and kind abstraction $\Lambda\omega.$) in pale color so that they are nonintrusive, and we often leave them implicit. We actually always do so in the elaboration typing rules below for conciseness.

For convenience, we use n-ary notations for homogeneous sequences of type-binders. We introduce let-binding $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ as syntactic sugar for $(\lambda(x : \_).\, e_2)\, e_1$; we define n-ary pack and unpack operators as follows:

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

3.4.2 Encoding of Signatures

M^ω signatures are actually F^ω types with some syntactic sugar. In F^ω, we see Y and A_I as term variables, similar to x's. We assume a collection ℓ_I of record labels indexed by identifiers I of the source language. Structural signatures $\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}$ are just syntactic sugar for record types $\{\overline{\mathcal{D}}\}$. A small trick is needed to represent type fields, which have no computational content, but cannot be erased during elaboration as they carry additional typing constraints. We reuse the solution of *F-ing* (Rossberg et al. [2014]), encoding them as identity functions with type annotations. For this, we introduce the following syntactic sugar for the term representing a type field (on the left). We overload the notation to also mean its type (on the right).

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

The type τ is used as argument of a higher-kinded type operator φ to uniformly handle the encoding of types of any kind. The key (and only useful) property is that two types (of the same kind) are equal if and only if their encodings are equal. Finally, declarations are syntactic sugar for record entries (distinguished by the category of the identifier):

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

3.4.3 Sharing Existential Types by Repacking

The encoding of module expressions as F^ω terms is slightly more involved than for signatures. Although structures and functors are simply encoded as records and functions, a difficulty arises from the need to *lift* existential types to extend their scope, as explained in Section 3.1.2.

Let us first consider the easier generative case. The only construct for handling a term with an abstract type is *unpack*, which allows using the term in a *subexpression*, hence with a limited scope, but not to make an abstract type accessible to the *rest of the program*. Yet, abstract type declarations inside modules have an *open* scope and are visible in the rest of the program. At a technical level, the difficulty comes from the representation of structures. To model them, one needs ordered records (also known as telescopes), where each component can introduce new abstract types accessible to the rest of the record, while standard F^ω only provides non-dependent records.

This observation was at the core of the design of *open existential types* Montagu [2010] and of *recursive type generativity* Dreyer [2007a]. Here, in order to stay in plain F^ω, we reuse and adapt the trick of *F-ing* (Rossberg et al. [2014]): structures are built field by field with a special *repacking* pattern: abstract types are unpacked, shared, but abstractly, with the rest of the structure, and then repacked. This allows the terms to mimic the existential lifting done in the types.

To capture this lifting of existentials out of records, we first introduce a combined syntactic form $\mathsf{repack}^{\blacktriangledown}\ \langle \bar\alpha, x\rangle = e_1 \ \mathsf{in}\ e_2$, which allows the abstract types of e_1 to appear in the type of e_2[3]:

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

Then, we use it to define a new construct to concatenate two records e_1 and e_2 with disjoint domains, but where e_2 might access the first record, via the bound name x_1, and reuse its abstract types, via the bound variables $\overline{\alpha}$:

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

It is better understood by the following derived typing rule and its use in the following example.

$$\frac{\Gamma \vdash e_1 : \exists^{\blacktriangledown}\bar\alpha.\ \{\overline{\ell_1 : \tau_1}\} \qquad \Gamma, \bar\alpha, x_1 : \{\overline{\ell_1 : \tau_1}\} \vdash e_2 : \exists^{\blacktriangledown}\bar\beta.\ \{\overline{\ell_2 : \tau_2}\} \qquad \overline{\ell_1} \mathbin{\#} \overline{\ell_2}}{\Gamma \vdash \mathsf{lift}\langle \bar\alpha, x_1 = e_1 \mathbin{@} e_2\rangle : \exists^{\blacktriangledown}\bar\alpha, \bar\beta.\ \{\overline{\ell_1 : \tau_1}; \overline{\ell_2 : \tau_2}\}}$$

Example 3.4.1. *A simple module M with three type fields, on the left-hand side. The raw encoding (after reduction of administrative let-bindings) on the right-hand side shows how abstract types are shared between components via lifting.*

Source

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

***Encoding of** e*

$$e = \mathit{lift}^{\blacktriangledown}\ \langle \alpha,\ x_1 = \mathit{pack}\,\langle (), \{\ell_t = \langle\!\langle () \rangle\!\rangle\}\rangle\ \mathit{as}\ \exists^{\blacktriangledown}\alpha.\ \{\ell_t : \langle\!\langle \alpha \rangle\!\rangle\}$$

$$\mathbin{@} \mathit{lift}^{\blacktriangledown}\ \langle \beta,\ x_2 = \mathit{pack}\,\langle (), \{\ell_u = \langle\!\langle () \rangle\!\rangle\}\rangle\ \mathit{as}\ \exists^{\blacktriangledown}\beta.\ \{\ell_u : \langle\!\langle \beta \rangle\!\rangle\}$$

$$\mathbin{@} \{\ell_v = \langle\!\langle \alpha \times \beta \rangle\!\rangle\}\rangle\rangle$$

***Signature of** e*

$$\mathcal{C} = \exists \alpha, \beta.\ \{\ell_t : \langle\!\langle \alpha \rangle\!\rangle\ ;\ \ell_u : \langle\!\langle \beta \rangle\!\rangle\ ;\ \ell_v : \langle\!\langle \alpha \times \beta \rangle\!\rangle\}$$

3.4.4 Transparent Existential Types and Their Lifting Through Function Types

The repacking pattern allows lifting existential types outside of record types. Unfortunately, this is insufficient for the applicative case, which uses skolemization to further lift abstract types out of the functor body to the front of the functor. This lifting of existential types though universal quantifiers by skolemization and through arrow types, as done in M^ω, is not definable in F^ω. More precisely, lifting through arrow types is *unsound*, while lifting through universal types is not expressible.

One solution is to avoid skolemization by *a-priori abstraction* over all possible type and term variables, i.e., the whole typing context. Doing so, existential types are always introduced at the front and need not be skolemized. This is the solution followed by the authors of *F-ing* (Rossberg et al. [2014]) and by Shan [2004]. While this suffices to prove soundness, the encoding is impractical for manual use of the pattern—as it requires frequently abstracting over the whole environment—and therefore does not provide a good intuition of what modules

[3]We leave the type implicit since the type of repacking is fully determined by the combination of $\bar\alpha$ and the type of e_2

really are. The encoding could be slightly improved by abstracting over fewer variables, without really solving the problem of a-priori abstraction.

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

We create a transparent existential type with the expression $\mathsf{pack}\ e\ \mathsf{as}\ \exists^{\triangledown\tau}(\alpha : \kappa).\,\sigma$, which behaves much as $\mathsf{pack}\ \langle\tau, e\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}(\alpha : \kappa).\,\sigma$, except that the witness type τ remains visible in the result type. A transparent existential type is thus weaker than a usual abstract type, as we still see the witness type. It is still abstract, as α cannot be turned back into its witness type τ and has to be treated abstractly. Two transparent existential types with different witnesses are incompatible. This could be seen as a weakness of transparent existentials, but it is actually a key to their lifting through arrow types.

Transparent existential types do not replace usual existential types, which we here call *opaque* existential types, but come in addition to them. Indeed, an expression of a transparent existential type can be further abstracted to become opaque, using the expression $\mathsf{seal}\ e$, which behaves as the identity but turns the expression e of type $\exists^{\triangledown\tau}(\alpha : \kappa).\,\sigma$ into one of type $\exists^{\blacktriangledown}(\alpha : \kappa).\,\sigma$.

Transparent existential types may also be used abstractly, with the expression $\mathsf{repack}^{\triangledown}\ \langle\alpha, x\rangle = e_1\ \mathsf{in}\ e_2$, which is the analog of the expression $\mathsf{repack}^{\blacktriangledown}\ \langle\alpha, x\rangle = e_1\ \mathsf{in}\ e_2$ but when e_1 is a transparent existential type $\exists^{\triangledown\tau}(\alpha : \kappa).\,\sigma_1$. In both cases, e_2 is typed in a context extended with the abstract types $\overline{\alpha}$ and a variable x of type σ_1. Crucially, e_2 cannot see the witnesses $\overline{\tau}$. However, the abstract type variables $\overline{\alpha}$ may still appear in the type σ_2 of the expression e_2, and therefore it is made transparent again in the result type of $\mathsf{repack}^{\triangledown}\ \langle\alpha, x\rangle = e_1\ \mathsf{in}\ e_2$, which is $\exists^{\triangledown\tau}(\alpha : \kappa).\,\sigma_2$. We do not need a primitive transparent version $\mathsf{unpack}^{\triangledown}\ \langle\alpha, x\rangle = e_1\ \mathsf{in}\ e_2$, since it can be defined as syntactic sugar for $\mathsf{unpack}\ \langle\alpha, x\rangle = \mathsf{seal}\ e_1\ \mathsf{in}\ e_2$.

Lifting through an arrow type So far, one may wonder what is the advantage of transparent existentials by comparison with opaque existentials. Their key advantage is that we can provide two key additional constructs for lifting transparent existentials across arrow types and universal types—the only reason to have introduced them in the first place. The lifting across an arrow type, written $\mathsf{lift}^{\rightarrow} e$, turns an expression of type $\sigma_1 \rightarrow \exists^{\triangledown\tau}(\alpha : \kappa).\,\sigma_2$ into one of type $\exists^{\triangledown\tau}(\alpha : \kappa).\,(\sigma_1 \rightarrow \sigma_2)$ as long as α is fresh for σ_1. Since we can observe the witness τ, we can ensure that the choice of the witness does not depend on the value (of type σ_1), allowing us to lift it outside of the function. While this operation seems easy, it crucially depends on existential types begin transparent—this transformation would be unsound with opaque existentials.

Unsoundness example For instance, let us consider the following example (adapted from the example of Dreyer et al. [2003]). We first define the type of a small record containing a value and a function that can be applied to the value:

$$\varphi \triangleq \lambda\alpha.\{\ell_x : \alpha, \ell_f : \alpha \rightarrow \mathsf{unit}\}$$

Then we consider the following expression:

$$\begin{aligned} m \triangleq \lambda x.\quad & \mathsf{if}\ x\ \mathsf{then}\ (\mathsf{pack}\ \langle\mathsf{int}, \{\ell_x = 42, \ell_f = \lambda x.()\}\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\alpha.\,\varphi\,\alpha) \\ & \qquad\quad \mathsf{else}\ (\mathsf{pack}\ \langle\mathsf{unit} \rightarrow \mathsf{unit}, \{\ell_x = \lambda u.(), \ell_f = \lambda x.x\,()\}\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\alpha.\,\varphi\,\alpha) \end{aligned}$$

It has type $\mathtt{bool} \rightarrow \exists^{\blacktriangledown}\alpha.\ \{\ell_x : \alpha, \ell_f : \alpha \rightarrow \mathsf{unit}\}$, but it would be unsound to consider it at the type $\exists^{\blacktriangledown}\alpha.\mathtt{bool} \rightarrow \{\ell_x : \alpha, \ell_f : \alpha \rightarrow \mathsf{unit}\}$. Indeed, assuming m has such type, the following

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

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

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

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

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

Figure 19: Typing rules for transparent existential types

well-typed code would get stuck:

$$\begin{aligned}\mathsf{unpack}\ \langle \alpha, M\rangle = m\ \mathsf{in}\ &\mathsf{let}\ M_1 = M\ \mathsf{true}\ \mathsf{in}\\ &\mathsf{let}\ M_2 = M\ \mathsf{false}\ \mathsf{in}\\ &M_2.f\ (M_1.x)\end{aligned}$$

The evaluation would get stuck on the application $42\ ()$. The takeaway is that function of type $\sigma_1 \to \exists^{\blacktriangledown}\alpha.\sigma_2$ can return an existential value whose witness type depends on the argument; it would thus be impossible to have a unique witness type to be used for all applications of the function, as required in a function of type $\exists^{\blacktriangledown}\alpha.(\sigma_1 \to \sigma_2)$. F^ω does not have dependent types, but have *existential types with dynamically chosen witnesses* (the opaque ones).

Lifting through an universal type Similarly, lifting across a universal type variable β of kind κ', written $\mathsf{lift}^{\forall} e$, turns an expression of type $\Lambda(\beta : \kappa').\exists^{\nabla\tau}(\alpha : \kappa).\sigma$ into one of type $\exists^{\nabla\lambda(\beta : \kappa').\tau}(\alpha' : \kappa' \to \kappa).\forall(\beta : \kappa').\sigma[\alpha \mapsto \alpha'\ \beta]$, provided β is fresh for τ, using skolemization of both the existential variable α and its witness type τ. It is a key for type soundness that β does not appear free in the witness type, hence that we *know* the witness type—this is why skolemization is not encodable with opaque existential types.

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

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

Their typing rules are given in Figure 19 and discussed below. Transparent existential types are introduced (just as opaque ones) by *packing*, by the Rule F-Hide. They can be transformed into opaque ones by *sealing*, where the witness is lost, in Rule F-Seal.

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

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

Transparent existential types do not have an unpack construct, only a repack construct, with the following rule:

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

This comes from the fact that, when unpacking $\mathsf{unpack}\ \langle \alpha, x\rangle = e_1\ \mathsf{in}\ e_2$ the left-hand-side expression e_2 only sees the right-hand-side e_1 through the variables α and x. Specifically, it could not pack the result back as a transparent existential, as it sees α only abstractly! Therefore, we only have a repack construct, and the normal unpack can be obtained by sealing and using the normal opaque unpack construct.

Finally, we have the two rules for the lifting operators F-LiftArr and F-LiftAll, written here without kinds for the sake of readability (the kinded version can be found in Figure 19):

$$\textsc{F-LiftArr}\quad \frac{\Gamma \vdash e : \sigma_1 \to \exists^{\nabla\tau}(\alpha).\,\sigma_2}{\Gamma \vdash \mathsf{lift}^{\to} e : \exists^{\nabla\tau}(\alpha).\,\sigma_1 \to \sigma_2} \qquad \textsc{F-LiftAll}\quad \frac{\Gamma \vdash e : \forall\beta.\,\exists^{\nabla\tau}(\alpha).\,\sigma}{\Gamma \vdash \mathsf{lift}^{\forall} e : \exists^{\nabla\lambda\beta.\,\tau}(\alpha').\,\forall\beta.\,\sigma[\alpha \mapsto (\alpha'\,\beta)]}$$

Semantics The added constructs have no additional computational content, namely repack behaves as a let-binding, while the other constructs behave as their underlying expression e.

Derived operators We add syntactic sugar for n-ary versions of transparent packing and repacking, as we did for opaque existentials. We write seal^n for n applications of seal. We can define a lifting operation $\mathsf{lift}^{\nabla}\langle\bar\alpha, x_1 = e_1 \mathbin{@} e_2\rangle$ for dependent record concatenation as the counterpart of the opaque version, by replacing opaque repacking by transparent repacking. Finally, we also define a new operation $\mathsf{lift}^{*} e$ that uses a combination of the primitive $\mathsf{lift}^{\to}$ and $\mathsf{lift}^{\forall}$ to turn an expression e of type $\forall\bar\alpha.\,\sigma_1 \to \exists^{\nabla\bar\tau}(\bar\beta).\,\sigma_2$ into one of type $\exists^{\nabla\overline{\lambda\bar\alpha.\,\tau}}(\bar\beta').\,\forall\alpha.\,\sigma_1 \to \sigma_2\big[\bar\beta \mapsto \overline{\beta'\,\bar\alpha}\,\big]$, which is the key transformation for lifting existentials out of applicative functor bodies. The operator lift^{*} is defined as $\mathsf{lift}_q^{\forall^p\triangleright}$ where p and q represent the size[4] of $\bar\alpha$ and $\bar\beta$, where $\mathsf{lift}_q^{\forall^p\triangleright}$ is itself inductively defined as follows:

$$\mathsf{lift}^{\diamond}_{q+1} e \triangleq \mathsf{repack}^{\nabla}\,\langle\alpha, x\rangle = \mathsf{lift}^{\forall^p}(\mathsf{lift}^{\triangleright} e)\ \mathsf{in}\ \mathsf{lift}^{\diamond}_q x \qquad\qquad \mathsf{lift}^{\diamond}_0 e \triangleq e$$

$$\mathsf{lift}^{\forall^{p+1}} e \triangleq \mathsf{lift}^{\forall}(\Lambda\alpha.\,\mathsf{lift}^{\forall^p}(e\,\alpha)) \qquad\qquad \mathsf{lift}^{\forall^0} e \triangleq e$$

- We allow $\mathsf{lift}^{\forall}$ to cross a sequence of quantifiers defining $\mathsf{lift}^{\forall(q+1)} e$ to be $\mathsf{lift}^{\forall}\big(\lambda\alpha.\,\mathsf{lift}^{\forall q} e\,\alpha\big)$ and $\mathsf{lift}\forall 0 e$ to be e.
- We allow $\mathsf{lift}^{\to}$ and $\mathsf{lift}^{\forall q}$ to be applied to a sequence of quantifiers $\exists^{\blacktriangledown}(\bar\alpha : \overline{\omega}).\,\tau$ instead of a single one, defining $\mathsf{lift}^{\diamond}_{n+1} e$ to be $\mathsf{repack}^{\nabla}\,\langle\alpha, x\rangle = \mathsf{lift}^{\diamond} e\ \mathsf{in}\ \mathsf{lift}^{\diamond}_n x$ with $\mathsf{lift}^{\diamond}_0 e$ equal to e.
- We define $\mathsf{lift}_n^{\forall q \to} e$ to be $\mathsf{lift}_n^{\forall q}(\mathsf{lift}_n^{\to} e)$.

We then simply write $\mathsf{lift}^{*} e$ for $\mathsf{lift}_n^{\forall q \to} e$, leaving the number of iterations n and q implicit from the type of the argument (taking the longest possible sequence). We have the following derived typing rule:

$$\textsc{F-LiftStar}\quad \frac{\Gamma \vdash e : \forall\alpha.\,\sigma \to \exists^{\nabla\bar\tau}(\beta).\,\sigma'}{\Gamma \vdash \mathsf{lift}^{*} e : \exists^{\nabla\overline{\lambda\alpha.\,\tau}}(\beta').\,\forall\alpha.\,\sigma \to \sigma'[\beta \mapsto (\beta'\,\overline{\alpha})]}$$

3.4.5 Implementation of Transparent Existential Types in F^ω

Interestingly, transparent existential types are completely definable in plain F^ω (with kind polymorphism). A concrete implementation is given on Figure 20, with and without syntactic sugar.

We first define a record e_0 with the constructs, which is actually pretty simple: most fields are η-expansions of the identity. Then, we define a type expression $\tau_{\mathbb{E}}$, and use normal (opaque) existentials of F^ω to pack e_0: $e_{\mathbb{E}} = \mathsf{pack}\,\langle\tau_0, e_0\rangle\ \mathsf{as}\ \tau_{\mathbb{E}}$ where τ_0 is the interface type that hides the implementation of the type $\mathbb{E}$. Using this definition, we may see a program e

[4] p and q are left implicit in $\mathsf{lift}^{*} e$ as they can be determined from the type of the argument e

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

(a) Implementation of transparent existentials as a library in F^ω with (predicative) kind polymorphism. The (kind-polymorphic) operator $\mathbb{E}$ should be understood as the transparent existential quantifier.

$$\begin{array}{lcl}
e_0 & \triangleq & \left\{\begin{array}{lcl}
\mathsf{Pack} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x:\varphi\,\alpha).\,x \\
\mathsf{Seal} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x:\varphi\,\alpha).\,\mathsf{pack}\,\langle\alpha,x\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\alpha.\ \varphi\,\alpha \\
\mathsf{Repack} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x:\mathbb{E}\,\alpha\,\varphi).\,\Lambda\psi.\,\lambda(f:\forall\alpha.\ \varphi\,\alpha\to\psi\,\alpha).\,(f\,\alpha\,x) \\
\mathsf{Lift}^{\to} & = & \Lambda\alpha.\,\Lambda\varphi.\,\Lambda\beta.\,\lambda(f:\beta\to\varphi\,\alpha).\,f \\
\mathsf{Lift}^{\forall} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x:(\forall\beta.\ \varphi\,\beta(\alpha\,\beta))).\,x
\end{array}\right\} \\
\tau_0 & \triangleq & \lambda\alpha.\,\lambda\varphi\,.\,\varphi\,\alpha \\
\tau_{\mathbb{E}} & \triangleq & \exists^{\blacktriangledown}(\exists^{\triangledown}.). \\
 & & \left\{\begin{array}{lcl}
\mathsf{Pack} & : & \forall\alpha.\,\forall\varphi\,.\,\varphi\,\alpha\to\exists^{\triangledown\alpha}\beta.\ \varphi_\beta \\
\mathsf{Seal} & : & \forall\alpha.\,\forall\varphi\,.\,\exists^{\triangledown\alpha}\beta.\ \varphi_\beta\to\exists^{\blacktriangledown}\beta.\ \varphi_\beta \\
\mathsf{Repack} & : & \forall\alpha.\,\forall\varphi\,.\,\exists^{\triangledown\alpha}\beta.\ \varphi_\beta\to\forall\psi\,.\,(\forall\alpha.\ \varphi\,\alpha\to\psi\,\alpha)\to\exists^{\triangledown\alpha}\beta.\ \psi_\beta \\
\mathsf{Lift}^{\to} & : & \forall\alpha.\,\forall\varphi\,.\,\forall\beta.\,(\beta\to\exists^{\triangledown\alpha}\gamma.\ \varphi_\gamma)\to\exists^{\triangledown\alpha}\gamma.\,(\beta\to\varphi\,\gamma)_\gamma \\
\mathsf{Lift}^{\forall} & : & \forall\alpha.\,\forall\varphi\,.\,(\forall\beta.\,\exists^{\triangledown\alpha\beta}\gamma.\,(\varphi\,\beta)_\gamma)\to\exists^{\triangledown\alpha}\gamma.\,(\forall\beta.\,\varphi\,\beta\,(\alpha\,\beta))_\gamma
\end{array}\right\} \\
e_{\mathbb{E}} & \triangleq & \mathsf{pack}\,\langle\tau_0,e_0\rangle\ \mathsf{as}\ \tau_{\mathbb{E}}
\end{array}$$

(b) Same implementation as above, but with omitted kinds and syntactic sugar for the transparent existential operator. We write a type variable as subscript as a shortcut to indicate that it is the free type variable (via an η-expension): φ_β for $\lambda\beta.\,(\varphi\,\beta)$.

Figure 20: Implementation of transparent existentials as a library in F^ω (with and without syntactic sugar)

using transparent existential types as a program $\mathsf{unpack}\ \langle \mathbb{E}\,, x_{\mathbb{E}}\,\rangle = e_{\mathbb{E}}\ \mathsf{in}\ e$ in plain F^ω, with the following additional syntactic sugar[5]:

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

$$\begin{array}{rcl}
\mathsf{seal}\ \ e & \triangleq & x_{\mathbb{E}}\,.\mathsf{Seal}\ \ \_\ \_\ \ e \\
\mathsf{lift}^{\rightarrow} e & \triangleq & x_{\mathbb{E}}\,.\mathsf{Lift}^{\rightarrow}\ \_\ \_\ \ e \\
\mathsf{lift}^{\forall}\ e & \triangleq & x_{\mathbb{E}}\,.\mathsf{Lift}^{\forall}\ \ \_\ \_\ \ e
\end{array}$$

We also write $\mathsf{repack}^{\Diamond}\,\langle\alpha, x\rangle = e_1\ \mathsf{in}\ e_2$ and $\mathsf{lift}^{\Diamond}\langle\bar{\alpha}, x_1 = e_1 \,@\, e_2\rangle$ where $\Diamond$ stands for either ∇ or $\blacktriangledown$.

Sound by construction The key take-away of this implementation is that it inherits soundness from F^ω: we do not need to do a soundness proof of the extended language. We verified this implementation in Coq via a shallow embedding.

3.4.6 Elaboration

As for M^ω, the elaboration relies on a subtyping judgment and a typing judgment for both signatures and modules. However, as M^ω signatures are already F^ω types, we can reuse the M^ω elaboration judgment (although we should now reread it with implicit kinds). Specifically, neither M^ω signatures nor its typing contexts mention transparent existential types. This is a key observation: transparent existential types may only appear in types of module expressions. This means that values of such types are never bound to a variable (during elaboration), which would otherwise force them to appear in the typing context. Instead, transparent existential types are always lifted to the top of the expression (using the lift operators).

There are two main elaboration judgments, for subtyping and typing.

- $\Gamma \vdash \mathtt{M} : \exists^{\Diamond}\overline{\alpha}.\mathcal{C} \rightsquigarrow e$ the typing of the module expressions and the declarations. In addition to the signature, an *evidence term* e is produced. The judgment is also defined for bindings $\Gamma \vdash_A \mathtt{D} : \exists^{\Diamond}\overline{\alpha}.\mathcal{D} \rightsquigarrow e$

- $\Gamma \vdash \mathcal{C} \prec: \mathcal{C}' \rightsquigarrow f$ the subtyping between M^ω signatures, extended to return the explicit coercion function f. The judgment is also defined for declarations $\Gamma \vdash \mathcal{D} \prec: \mathcal{D}' \rightsquigarrow f$

Properties of those two judgments are stated and proved in the next section, Theorem 7.

Subtyping

The rules are given in Figure 21 and discussed below.

Declarations As subtyping between declarations is restricted to the equality for type and value fields, the coercion functions of rules E-Sub-Decl-Val and E-Sub-Decl-Type are just the identity:

E-Sub-Decl-Val
$\Gamma \vdash (\mathsf{val}\ x : \tau) \prec: (\mathsf{val}\ x : \tau) \rightsquigarrow \lambda x.x$

E-Sub-Decl-Type
$\Gamma \vdash (\mathsf{type}\ t = \tau) \prec: (\mathsf{type}\ t = \tau) \rightsquigarrow \lambda x.x$

[5]Here, _ stands for kinds or types that are left implicit as they can be straightforwardly inferred from other arguments. We also extend transparent existentials with sequences of abstractions as we did for opaque existentials.

E-Sub-Sig-Struct
$$\frac{\overline{\mathcal{D}_0} \subseteq \overline{\mathcal{D}} \qquad \Gamma \vdash \overline{\mathcal{D}_0} \prec: \overline{\mathcal{D}'} \rightsquigarrow \overline{f} \qquad \overline{I'} = \mathsf{dom}(\overline{\mathcal{D}'})}{\Gamma \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \prec: \mathsf{sig}\ \overline{\mathcal{D}'}\ \mathsf{end} \rightsquigarrow \lambda x.\ \left\{\overline{\ell_{I'} = f\,(x.\ell_{I'})}\right\}}$$

E-Sub-Sig-GenFct
$$\frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} \prec: \mathcal{C}'[\overline{\alpha'} \mapsto \overline{\tau}] \rightsquigarrow f}{\Gamma \vdash () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \prec: () \to \exists^{\blacktriangledown}\overline{\alpha'}.\mathcal{C}' \rightsquigarrow \lambda x. \lambda u. \mathsf{unpack}\ \langle \overline{\alpha}, y\rangle = x\,()\ \mathsf{in}\ \mathsf{pack}\ \langle \overline{\tau}, f\,y\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\overline{\alpha'}.\mathcal{C}'}$$

E-Sub-Sig-AppFct
$$\frac{\Gamma, \overline{\alpha'} \vdash \mathcal{C}'_a \prec: \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow f \qquad \Gamma, \overline{\alpha'} \vdash \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] \prec: \mathcal{C}' \rightsquigarrow g}{\Gamma \vdash \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \prec: \forall\overline{\alpha'}.\mathcal{C}'_a \to \mathcal{C}' \rightsquigarrow \lambda x : (\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}). \Lambda\overline{\alpha'}. \lambda y : \mathcal{C}'_a. g\,(x\,\overline{\tau}\,(f\,y))}$$

(a) Rules for signatures

E-Sub-Decl-Val
$$\Gamma \vdash (\mathsf{val}\ x : \tau) \prec: (\mathsf{val}\ x : \tau) \rightsquigarrow \lambda x.x$$

E-Sub-Decl-Type
$$\Gamma \vdash (\mathsf{type}\ t = \tau) \prec: (\mathsf{type}\ t = \tau) \rightsquigarrow \lambda x.x$$

E-Sub-Decl-Mod
$$\frac{\Gamma \vdash \mathcal{C} \prec: \mathcal{C}' \rightsquigarrow f}{\Gamma \vdash (\mathsf{module}\ X : \mathcal{C}) \prec: (\mathsf{module}\ X : \mathcal{C}') \rightsquigarrow f}$$

E-Sub-Decl-ModType
$$\frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} \prec: \mathcal{C}' \rightsquigarrow f \qquad \Gamma, \overline{\alpha} \vdash \mathcal{C}' \prec: \mathcal{C} \rightsquigarrow g}{\Gamma \vdash (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}) \prec: (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}') \rightsquigarrow \lambda x.x}$$

(b) Rules for declarations

Figure 21: Subtyping with elaboration (outputing the coercion function)

For submodules, the coercion function is just returned directly:

E-Sub-Decl-Mod
$$\frac{\Gamma \vdash \mathcal{C} \prec: \mathcal{C}' \rightsquigarrow f}{\Gamma \vdash (\mathsf{module}\ X : \mathcal{C}) \prec: (\mathsf{module}\ X : \mathcal{C}') \rightsquigarrow f}$$

Finally, for module types, the coercion function is again the identity:

E-Sub-Decl-ModType
$$\frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} \prec: \mathcal{C}' \rightsquigarrow f \qquad \Gamma, \overline{\alpha} \vdash \mathcal{C}' \prec: \mathcal{C} \rightsquigarrow g}{\Gamma \vdash (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}) \prec: (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}') \rightsquigarrow \lambda x.x}$$

Signatures The rules for signature display coercion functions that are a bit more involved. First, for structural signatures, the coercion functions of each fields are gathered and used to produce the result:

E-Sub-Sig-Struct
$$\frac{\overline{\mathcal{D}_0} \subseteq \overline{\mathcal{D}} \qquad \Gamma \vdash \overline{\mathcal{D}_0} \prec: \overline{\mathcal{D}'} \rightsquigarrow \overline{f} \qquad \overline{I'} = \mathsf{dom}(\overline{\mathcal{D}'})}{\Gamma \vdash \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \prec: \mathsf{sig}\ \overline{\mathcal{D}'}\ \mathsf{end} \rightsquigarrow \lambda x.\ \left\{\overline{\ell_{I'} = f\,(x.\ell_{I'})}\right\}}$$

For applicative functors, we have two coercion functions: f for the domain and g for the codomain, as displayed in the following rule:

E-Sub-Sig-AppFct
$$\frac{\Gamma, \overline{\alpha'} \vdash \mathcal{C}'_a \prec: \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow f \qquad \Gamma, \overline{\alpha'} \vdash \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] \prec: \mathcal{C}' \rightsquigarrow g}{\Gamma \vdash \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \prec: \forall\overline{\alpha'}.\mathcal{C}'_a \to \mathcal{C}' \rightsquigarrow \lambda x : (\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}). \Lambda\overline{\alpha'}. \lambda(y : \mathcal{C}'_a). g\,(x\,\overline{\tau}\,(f\,y))}$$

The resulting coercion function takes the function $x : (\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C})$ as input, then a set of type parameters $\overline{\alpha}'$, and finally a parameter $(y : \mathcal{C}'_a)$. To call x, we first specialize it on the instantiation $\overline{\tau}$, then coerce y into the type $\mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}]$ by using f. Therefore, we have $x\ \overline{\tau}\ (f\ y)$, which is of type $\mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]$. This is coerced back into $\mathcal{C}'$ by calling g. Overall, it can be thought of as being more or less the composition $g \circ x \circ f$

For generative functors, we have only a single coercion function f for the codomain, but we also need to take care of the existential quantification on both sides, that might differ:

E-Sub-Sig-GenFct
$$\frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} \prec: \mathcal{C}'[\overline{\alpha}' \mapsto \overline{\tau}] \rightsquigarrow f}{\Gamma \vdash () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \prec: () \to \exists^{\blacktriangledown}\overline{\alpha}'.\mathcal{C}' \rightsquigarrow \lambda x.\lambda u.\mathsf{unpack}\ \langle \overline{\alpha}, y\rangle = x\ ()\ \mathsf{in}\ \mathsf{pack}\ \langle \overline{\tau}, f\ y\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\overline{\alpha}'.\mathcal{C}'}$$

The coercion function basically repacks the result of the functor with the right type, using the coercion function of the codomain.

Typing

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

The judgment $\Gamma \vdash^{\Diamond} \mathtt{M} : \exists^{\bar{\vartheta}}\bar{\alpha}.\mathcal{C} \rightsquigarrow e$ extends M^ω typing with the elaborated module term e. The mode $\Diamond$ must coincide with $\bar{\vartheta}$, and may be left implicit, as we did for the corresponding M^ω judgment. Hence, we usually just write $\Gamma \vdash \mathtt{M} : \exists^{\bar{\vartheta}}\bar{\alpha}.\mathcal{C} \rightsquigarrow e$. A similar, helper judgment $\Gamma \vdash^{\bar{\vartheta}}_{A} \mathtt{B} : \exists^{\bar{\vartheta}}\bar{\alpha}.\mathcal{D} \rightsquigarrow e$ is also defined for bindings. When reading an M^ω type environment Γ in F^ω, we must read $A.(\mathsf{val}\ x : \tau)$ and $A.(\mathsf{module}\ X : \mathcal{C})$ as $A_x : \tau$ and $A_X : \mathcal{C}$, etc.

The typing rules are given in Figure 22 and discussed bellow.

Sequences and structures The key rule for structures is the sequence rule that combines bindings. It may be concisely written as follows for generative and applicative modes:

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

E-Typ-Bind-SeqApp
$$\frac{\Gamma \vdash_A \mathtt{B} : \exists^{\triangledown\overline{\tau}_1}(\overline{\alpha}_1).\mathcal{D} \rightsquigarrow e_1 \qquad \Gamma, \overline{\alpha}_1, A.\mathcal{D} \vdash_A \overline{\mathtt{B}} : \exists^{\triangledown\overline{\tau}_2}(\overline{\alpha}_2).\overline{\mathcal{D}} \rightsquigarrow e_2}{\Gamma \vdash_A \mathtt{B}; \overline{\mathtt{B}} : \exists^{\triangledown\overline{\tau}_1\overline{\tau}_2}(\overline{\alpha}_1\overline{\alpha}_2).(\mathcal{D}, \overline{\mathcal{D}}) \rightsquigarrow \mathsf{lift}^{\triangledown}\langle \overline{\alpha}_1, x_1 = e_1\ @\ (\mathsf{let}\ A_{I_1} = x_1.\ell_{I_1}\ \mathsf{in}\ e_2)\rangle}$$

The single field of e_1 is concatenated with the fields of e_2 after lifting out their existential bindings. In both cases, the field of e_1 is made visible in e_2, as well as the existentials in front of e_1—but abstractly. As the those two rules are actually similar, they can be factored into an unified rule E-Typ-Bind-Seq:

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

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

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

E-Typ-Mod-Arg

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

E-Typ-Mod-Var

$$\frac{(A.X :\ \mathsf{module}\,\mathcal{C}) \in \Gamma}{\Gamma \vdash A.X : \mathcal{C} \rightsquigarrow A_X}$$

E-Typ-Mod-AppFct

$$\frac{\Gamma \vdash \texttt{S} : \lambda\overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a \vdash \texttt{M} : \exists^{\nabla\overline{\tau}}(\overline{\beta}).\mathcal{C} \rightsquigarrow e}{\Gamma \vdash (Y : \texttt{S}) \to \texttt{M} : \exists^{\nabla\overline{\lambda\overline{\alpha}.\tau}}(\overline{\beta'}).\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right] \rightsquigarrow \mathsf{lift}^*(\Lambda\overline{\alpha}.\,\lambda(Y : \mathcal{C}_a).\,e)}$$

E-Typ-Mod-AppApp

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

E-Typ-Mod-GenFct

$$\frac{\Gamma \vdash \texttt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash () \to \texttt{M} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow \lambda(\_\ : ()).\,e}$$

E-Typ-Mod-GenApp

$$\frac{\Gamma \vdash P : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash P() : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow e\,()}$$

E-Typ-Mod-Ascr

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

E-Typ-Mod-Seal

$$\frac{\Gamma \vdash \texttt{M} : \exists^{\nabla\overline{\tau}}(\overline{\alpha}).\mathcal{C} \rightsquigarrow e}{\Gamma \vdash \texttt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow \mathsf{seal}^{|\overline{\alpha}|}\ e}$$

E-Typ-Mod-Struct

$$\frac{\Gamma \vdash_A \overline{\texttt{B}} : \exists^{\bar{\vartheta}}\bar{\alpha}.\overline{\mathcal{D}} \rightsquigarrow e \qquad A \notin \Gamma}{\Gamma \vdash \texttt{struct}_A\ \overline{\texttt{B}}\ \mathsf{end} : \exists^{\bar{\vartheta}}\bar{\alpha}.\,\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \rightsquigarrow e}$$

E-Typ-Mod-Proj

$$\frac{\Gamma \vdash \texttt{M} : \exists^{\bar{\vartheta}}\bar{\alpha}.\,\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \rightsquigarrow e \qquad \mathsf{module}\ X : \mathcal{C} \in \overline{\mathcal{D}} \qquad \overline{\alpha}' = \mathsf{fv}(\mathcal{C}) \cap \overline{\alpha}}{\Gamma \vdash \texttt{M}.X : \exists^{\bar{\vartheta}}\bar{\alpha}.\mathcal{C} \rightsquigarrow \mathsf{clean}^{\diamondsuit}\langle\overline{\alpha}, \overline{\alpha}'\rangle\ (\mathsf{repack}^{\diamondsuit}\ \langle\overline{\alpha}, x\rangle = e\ \mathsf{in}\ x.\ell_X)}$$

(a) Rules for modules

E-Typ-Bind-Let

$$\frac{\Gamma \vdash \texttt{e} : \tau \rightsquigarrow e}{\Gamma \vdash_A (\texttt{let}\,x = \texttt{e}) : (\mathsf{val}\ x : \tau) \rightsquigarrow \{\ell_x = e\}}$$

E-Typ-Bind-Type

$$\frac{\Gamma \vdash \texttt{u} : \tau}{\Gamma \vdash_A (\texttt{type}\,t = \texttt{u}) : (\mathsf{type}\ t = \tau) \rightsquigarrow \{\ell_t = \langle\!\langle\tau\rangle\!\rangle\}}$$

E-Typ-Bind-ModType

$$\frac{\Gamma \vdash \texttt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\texttt{module type}\ T = \texttt{S}) : (\mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C}) \rightsquigarrow \{\ell_T = \langle\!\langle\lambda\overline{\alpha}.\mathcal{C}\rangle\!\rangle\}}$$

E-Typ-Bind-Empty

$$\Gamma \vdash_A \varnothing : \varnothing \rightsquigarrow \{\}$$

E-Typ-Bind-Mod

$$\frac{\Gamma \vdash_A \texttt{M} : \exists^{\bar{\vartheta}}\bar{\alpha}.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash_A (\texttt{module}\,X = \texttt{M}) : (\exists^{\bar{\vartheta}}\bar{\alpha}.\,\mathsf{module}\ X : \mathcal{C}) \rightsquigarrow \mathsf{repack}^{\diamondsuit}\ \langle\overline{\alpha}, x\rangle = e\ \mathsf{in}\ \{\ell_X = x\}}$$

E-Typ-Bind-Seq

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

(b) Rules for bindings

Figure 22: Typing rules with elaboration

Modes and sealing By default, elaboration is done in applicative mode, hence inferring transparent existentials, but it can be turned into generative mode when required, using the floating Rule E-Typ-Mod-Seal, which, unsurprisingly, uses the sealing operator:

$$\textsc{E-Typ-Mod-Seal}\quad \frac{\Gamma \vdash \mathtt{M} : \exists^{\nabla \overline{\tau}}(\overline{\alpha}).\mathcal{C} \rightsquigarrow e}{\Gamma \vdash \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow \mathsf{seal}^{|\overline{\alpha}|}\ e}$$

Since signature ascription is defined on paths, it is always applicative (rule E-Typ-Sig-App):

$$\textsc{E-Typ-Mod-Ascr}\quad \frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C} \qquad \Gamma \vdash P : \mathcal{C}' \rightsquigarrow e \qquad \Gamma \vdash \mathcal{C}' \prec: \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow f \qquad \Gamma \vdash \overline{\tau} : \overline{\overline{\varsigma}}}{\Gamma \vdash (P : \mathtt{S}) : \exists^{\nabla \overline{\tau}}(\overline{\alpha}).\mathcal{C} \rightsquigarrow \mathsf{pack}\ f\, e\ \mathsf{as}\ \exists^{\nabla \overline{\tau}}(\overline{\alpha}).\mathcal{C}}$$

That is, signature ascription $(P : \mathtt{S})$ may introduce new abstract types $\overline{\alpha}$ as prescribed by the (elaboration $\lambda\overline{\alpha}.\mathcal{C}$ of the) signature $\mathtt{S}$, but these are transparent existentials in the type of $(P : \mathtt{S})$. The fact that ascription actually does not introduce opaque existential is essential for sealing inside applicative functors, a feature often left out of formalization attempts.

Elaboration of functors At first glance, the elaboration of functors seems to differ quite significantly for the applicative case (Rule E-Typ-Mod-AppFct) and generative case (Rule E-Typ-Mod-GenFct):

$$\textsc{E-Typ-Mod-AppFct}\quad \frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a \vdash \mathtt{M} : \exists^{\nabla \overline{\tau}}(\overline{\beta}).\mathcal{C} \rightsquigarrow e}{\Gamma \vdash (Y : \mathtt{S}) \to \mathtt{M} : \exists^{\nabla \overline{\lambda\overline{\alpha}.\tau}}(\overline{\beta'}).\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\Big[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\Big] \rightsquigarrow \mathsf{lift}^*\Lambda\overline{\alpha}.\, \lambda(Y : \mathcal{C}_a).\, e}$$

$$\textsc{E-Typ-Mod-GenFct}\quad \frac{\Gamma \vdash \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash () \to \mathtt{M} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow \lambda(\_ : ()).\, e}$$

The body of an applicative functor is elaborated to transparent existentials which are lifted, while in the generative case, the existentials are opaque and cannot be lifted. However, this difference is largely artificial as a result of using a special argument () to enforce generativity. Otherwise, the main difference lies in enforcing the body of the functor to be typed in generative mode, hence with an opaque existential type. Since lift^* is neutral on terms that do not have transparent existential types, the elaboration of the generative case could also be written $\mathsf{lift}^*\lambda(\_ : ()).\, e$, so that the two cases only differ by the modes of elaboration of their bodies.

Functor applications The corresponding rules for applying applicative functors E-Typ-Mod-AppApp and generative functors E-Typ-Mod-GenFct are relatively straightforward, using the normal F^ω application:

$$\textsc{E-Typ-Mod-AppApp}\quad \frac{\Gamma \vdash P : \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \rightsquigarrow e \qquad \Gamma \vdash P' : \mathcal{C}' \rightsquigarrow e' \qquad \Gamma \vdash \mathcal{C}' \prec: \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow f}{\Gamma \vdash P(P') : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}] \rightsquigarrow e\, \overline{\tau}\ (f\, e')}$$

$$\textsc{E-Typ-Mod-GenFct}\quad \frac{\Gamma \vdash \mathtt{M} : \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash () \to \mathtt{M} : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \rightsquigarrow \lambda(\_ : ()).\, e}$$

Projection As expected, projection is elaborated into a record projection, but there is a catch: the projected module expression M can have existential types. The solution, displayed in the following simplified rule, is to unpack, project and pack again i.e., to repack over the projection:

E-Typ-Mod-Proj-SIMPLIFIED

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

As for structures and sequences, we have a single rule for both modes that relies on the mode of the $\mathsf{repack}^{\Diamond}$ construct. To exactly match the M^ω rule for projection M-Typ-Mod-Proj, we also need to "garbage-collect" the abstract variables that do not appear in the result type. This is done with a $\mathsf{clean}^{\Diamond}\langle \overline{\alpha}, \overline{\beta} \rangle\ e$ macro that removes the type variables in $\overline{\alpha}$ that do not appear in $\overline{\beta}$. It is defined as:

$$\begin{aligned}
\mathsf{clean}^{\Diamond}\langle \gamma\overline{\alpha}, \gamma\overline{\beta} \rangle\ e &\triangleq \mathsf{repack}^{\Diamond}\ \langle \gamma, x \rangle = e\ \mathsf{in}\ \mathsf{clean}^{\Diamond}\langle \overline{\alpha}, \overline{\beta} \rangle\ x \\
\mathsf{clean}^{\Diamond}\langle \gamma\overline{\alpha}, \gamma'\overline{\beta} \rangle\ e &\triangleq \mathsf{unpack}^{\Diamond}\ \langle \gamma, x \rangle = e\ \mathsf{in}\ \mathsf{clean}^{\Diamond}\langle \overline{\alpha}, \gamma\overline{\beta} \rangle\ x \\
\mathsf{clean}^{\Diamond}\langle \varnothing, \varnothing \rangle\ e &\triangleq e
\end{aligned}$$

Using this macro, we can define the proper projection rule:

E-Typ-Mod-Proj

$$\frac{\Gamma \vdash \mathtt{M} : \exists^{\vartheta} \bar{\alpha}. \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \rightsquigarrow e \qquad \mathsf{module}\ X : \mathcal{C} \in \overline{\mathcal{D}} \qquad \overline{\alpha}' = \mathsf{fv}(\mathcal{C}) \cap \overline{\alpha}}{\Gamma \vdash \mathtt{M}.X : \exists^{\vartheta} \bar{\alpha}. \mathcal{C} \rightsquigarrow \mathsf{clean}^{\Diamond}\langle \overline{\alpha}, \overline{\alpha}' \rangle\ (\mathsf{repack}^{\Diamond}\ \langle \overline{\alpha}, x \rangle = e\ \mathsf{in}\ x.\ell_X)}$$

Bindings The elaboration of bindings produces records with a single field. The rules for value, type fields and module type fields are straightforward. Using the type encoding for values, which we recall as:

$$\langle\!\langle (\tau : \omega) \rangle\!\rangle = \lambda(\beta : \omega \to \star).\, \lambda(x : (\beta\, \tau)).\, x$$

we get the following rules:

E-Typ-Bind-Let

$$\frac{\Gamma \vdash \mathtt{e} : \tau \rightsquigarrow e}{\Gamma \vdash_A (\mathtt{let}\, x = \mathtt{e}) : (\mathsf{val}\ x : \tau) \rightsquigarrow \{\ell_x = e\}}$$

E-Typ-Bind-Type

$$\frac{\Gamma \vdash \mathtt{u} : \tau}{\Gamma \vdash_A (\mathsf{type}\, t = \mathtt{u}) : (\mathsf{type}\, t = \tau) \rightsquigarrow \{\ell_t = \langle\!\langle \tau \rangle\!\rangle\}}$$

E-Typ-Bind-ModType

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

The rule for modules bindings displays an extrusion, which, like for the rule of sequences E-Typ-Bind-Seq, uses repacking:

E-Typ-Bind-Mod

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

Again, we can merge the two modes within one rule by using the mode-specific *repacking.*

3.4.7 Properties of elaboration

In this section we state and prove the soundness and correctness of the elaboration of M^ω.

Soundness

We start with a lemma regarding subtyping:

Lemma 6 (Soundness of subtyping)**.** *The coercion functions are well-typed in F^ω. For subtyping between signatures, we have:*

$$\Gamma \vdash \mathcal{C} \prec: \mathcal{C}' \rightsquigarrow f \implies \Gamma \vdash f : \mathcal{C} \to \mathcal{C}' \tag{3.1}$$

For subtyping between declarations, we have (without the syntactic sugar of declarations):

$$\Gamma \vdash (\ell_I : \tau) \prec: (\ell_I : \tau') \rightsquigarrow f \quad \wedge \implies \Gamma \vdash f : \tau \to \tau' \tag{3.2}$$

Proof. We proceed by induction on the typing derivation. The proof cases are immediate applications of the typing rules of F^ω.

- E-Sub-Decl-Val and E-Sub-Decl-Type are immediate (the coercion is the identity). E-Sub-Decl-Mod is immediate by induction hypothesis.
- E-Sub-Decl-ModType: for module-types bindings, we use the fact that the kernel of subtyping is a subset of type equivalence.
- E-Sub-Sig-Struct is immediate by induction hypothesis
- E-Sub-Sig-GenFct: we recall the rule:

  E-Sub-Sig-GenFct
  $$\frac{\Gamma, \overline{\alpha} \vdash \mathcal{C} \prec: \mathcal{C}'[\overline{\alpha}' \mapsto \overline{\tau}] \rightsquigarrow f}{\Gamma \vdash () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C} \prec: () \to \exists^{\blacktriangledown}\overline{\alpha}'.\mathcal{C}' \rightsquigarrow \lambda x.\lambda u.\,\mathsf{unpack}\ \langle \overline{\alpha}, y\rangle = x\,()\ \mathsf{in}\ \mathsf{pack}\ \langle \overline{\tau}, f\,y\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\overline{\alpha}'.\mathcal{C}'}$$

  We have $\Gamma, \overline{\alpha} \vdash f : \mathcal{C} \to \mathcal{C}'[\overline{\alpha}' \mapsto \overline{\tau}]$ by induction hypothesis. From there,

  $$\begin{aligned} &\Gamma, (x : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}), \overline{\alpha}, y : \mathcal{C} \vdash (f\,y) : \mathcal{C}'\left[\overline{\alpha}' \mapsto \overline{\tau}\right] && \text{(By F-App)}\\ \implies &\Gamma, (x : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}), \overline{\alpha}, y : \mathcal{C} \vdash \mathsf{pack}\ \langle \overline{\tau}, f\,y\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\overline{\alpha}'.\mathcal{C}' : \exists^{\blacktriangledown}\overline{\alpha}'.\mathcal{C}' && \text{(By F-Pack)}\\ \implies &\Gamma, (x : () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}) \vdash \mathsf{unpack}\ \langle \overline{\alpha}, y\rangle = x\,()\ \mathsf{in}\ \mathsf{pack}\ \langle \overline{\tau}, f\,y\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\overline{\alpha}'.\mathcal{C}' : \exists^{\blacktriangledown}\overline{\alpha}'.\mathcal{C}' \\ & && \text{(By F-Unpack)} \end{aligned}$$

  We conclude by two applications of F-Abs.
- E-Sub-Sig-AppFct is similar.

□

We then have the main soundness theorem:

Theorem 7: Soundness

When typing a module, the elaborated module term is well typed regarding F^ω typing, and the source module term is well typed regarding M^ω typing.

$$\begin{aligned} \Gamma \vdash \mathtt{M} : \exists^{\bar\vartheta}\bar\alpha.\mathcal{C} \rightsquigarrow e &\implies \Gamma \vdash e : \exists^{\bar\vartheta}\bar\alpha.\mathcal{C} \quad &&\wedge\ \Gamma \vdash \mathtt{M} : \exists^{\bar\vartheta}\bar\alpha.\mathcal{C} \\ \Gamma \vdash \overline{\mathtt{B}} : \exists^{\bar\vartheta}\bar\alpha.\overline{\mathcal{D}} \rightsquigarrow e &\implies \Gamma \vdash e : \exists^{\bar\vartheta}\bar\alpha.\left\{\overline{\mathcal{D}}\right\} &&\wedge\ \Gamma \vdash \mathtt{M} : \exists^{\bar\vartheta}\bar\alpha.\overline{\mathtt{D}} \end{aligned} \tag{3.3}$$

Proof We proceed by induction on the typing derivation. The cases a mostly immediate applications of F^ω rules.

- Immediate cases: E-Typ-Mod-Arg and E-Typ-Mod-Var are immediate by F-Var. The rules for value, type and module type bindings (E-Typ-Bind-Let, E-Typ-Bind-Type, E-Typ-Bind-ModType) are also immediate by F-Record.

- E-Typ-Mod-AppFct: we recall the typing rule:

  $$\textsc{E-Typ-Mod-AppFct}\quad \frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}_a\ (\mathbf{1}) \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a \vdash \mathtt{M} : \exists^{\nabla\overline{\tau}}(\overline{\beta}).\mathcal{C} \rightsquigarrow e\ (\mathbf{2})}{\Gamma \vdash (Y : \mathtt{S}) \to \mathtt{M} : \exists^{\nabla\overline{\lambda\overline{\alpha}.\tau}}(\overline{\beta'}).\forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right] \rightsquigarrow \mathsf{lift}^*(\Lambda\overline{\alpha}.\,\lambda(Y : \mathcal{C}_a).\,e)}$$

  By induction hypothesis on (2), we have:

  $$\Gamma, \overline{\alpha}, Y : \mathcal{C}_a \vdash e : \exists^{\nabla\overline{\tau}}(\overline{\beta}).\mathcal{C}$$

  By F-Abs and F-Tabs, we have

  $$\Gamma \vdash (\Lambda\overline{\alpha}.\,\lambda(Y : \mathcal{C}_a).\,e) : \forall\overline{\beta}.\mathcal{C}_a \to \exists^{\nabla\overline{\tau}}(\overline{\beta}).\mathcal{C}$$

  We conclude by F-LiftStar.

- E-Typ-Mod-GenFct and E-Typ-Mod-GenApp are immediate by induction hypothesis.

- E-Typ-Mod-AppApp, immediate by induction hypothesis, and using the subtyping lemma.

Theorem 8: Completeness

Well-typed M^ω terms and bindings can always be elaborated:

$$\begin{array}{rcl} \Gamma \vdash \mathtt{M} : \exists^{\Diamond}\bar{\alpha}.\mathcal{C} & \Longrightarrow & \exists e, \bar{\vartheta},\ \Gamma \vdash \mathtt{M} : \exists^{\bar{\vartheta}}\bar{\alpha}.\mathcal{C} \rightsquigarrow e\ \wedge\ \mathsf{mode}(\bar{\vartheta}) = \Diamond \\ \Gamma \vdash \overline{\mathtt{B}} : \exists^{\Diamond}\bar{\alpha}.\overline{\mathcal{D}} & \Longrightarrow & \exists e, \bar{\vartheta},\ \Gamma \vdash \overline{\mathtt{B}} : \exists^{\bar{\vartheta}}\bar{\alpha}.\overline{\mathcal{D}} \rightsquigarrow e\ \wedge\ \mathsf{mode}(\bar{\vartheta}) = \Diamond \end{array} \tag{3.4}$$

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

3.5 Abstract module-types

In this section we extend M^ω with abstract signatures. We restrict abstract signatures to *simple abstract signatures* (as presented in Section 1.4.3): abstract signatures can only be instantiated by signatures that do not contain abstract module-type bindings. Yet, "*simple*" is a bit misleading: the polymorphism that is provided by simple abstract signatures is not just at the level of types, but also at the structure of types themselves. In the rest of this section we refer to signatures that are not abstract as *concrete signatures* (they might contain abstract signatures as fields)

In Section 3.5.1, we introduce the key points of the encoding of abstract signatures via examples. In Section 3.5.2, we extend F^ω (and M^ω) with functions and tuples at the kind level to represent the form of polymorphism provided by abstract signatures. In Section 3.5.3, we present the extended typing rules of M^ω.

3.5.1 Key intuitions of abstract signatures

In this subsection we introduce the key intuitions of abstract signatures through examples. The extensions presented here (mostly to the kind system) are given formally in Section 3.5.2.

Ascription Abstract signatures in positive positions are easy: they acts as existentially quantified types. Informally, after ascription with the following left-hand side source signature, we would get the right-hand side M^ω signature (using the variable Ψ for an abstract signature variable):

```
sig_A
  module type T = A.T
  module X_1 : A.T
  module X_2 : A.T
end
```

```
∃(ℵ : ⋆).sig
  module type T = Ψ
  module X_1 : Ψ
  module X_2 : Ψ
end
```

An important point here is that we do not need existential kinds: once the signature has been abstracted away, the fact that it might have been parametric is not expressible anymore and therefore, does not matter.

Kind polymorphism and *flexroots* Let us consider the following functor:

```
module F = (Y : sig_A module type  T = A.T  module X : A.T end) → Y.X
```

As a first approximation, we could try an M^ω signature of F of the following form:

```
module F : ∀(ℵ : ⋆)
  sig module type T = Ψ module X : Ψ end → Ψ
```

Here, the abstract signature variable Ψ could be instantiated by any wellformed signature, but there is a catch: if cannot by instantiated by a *parametric* signature (which is not of kind $\star$), like $\lambda\alpha.\mathsf{sig}\ \mathsf{type}\ t = \alpha\ \mathsf{end}$. We could fix the signature of F to account for parametric signatures with only type parameter:

```
module F : ∀(Ψ : ⋆ → ⋆).∀(α : ⋆)
  sig module type T = λβ.(Ψ β)   module X : (Ψ α) end → (Ψ α)
```

Every occurrence of the abstract signature introduces a new type variable (here, only α as there is only once occurrence of T) that is the corresponding parameter of the occurrence. But what if the signature has several type parameters ? The functor should support *any* number of type parameters. We can achieve this by using kind polymorphism[6] $\forall\omega.$, where the kind variable ω can be instantiated to represent any arity. To represent arity at the kind level, we introduce *kind tuples* that range from the empty kind tuple $\varnothing$ to an n-tuple $\varsigma_1 \times \cdots \times \varsigma_n$. This gives us the following kind-polymorphic signature for F (that subsumes the two signatures presented before):

```
module F : ∀ω.(Ψ : ω → ⋆).∀(α : ω)
  sig module type T = λ(β : ω).(Ψ β)   module X : (Ψ α) end → (Ψ α)
```

The type variable α contains all the parameterized types of Ψ for the module X. This idea of having one variable to hold all the type components of a signature is inspired by the (similar) *flexroot* concept of Shao [1999]. In our setting, flexroots are not records but mere tuples, and we abstract over the kind of the flexroot. Every occurrence of the abstract signature introduces a corresponding flexroot, as every instance might be parameterized differently.

[6] We used kind polymorphism already in Section 3.4.5 for the implementation of transparent existentials as a library in F^ω. Then, it was only to have a completely internal presentation, without meta-notations. Here, it is not to circumvent meta-notations, but really to express the right polymorphism.

To illustrate this, we consider the following signature:

```
sig_A   module type $T = A.T$   module $X_1 : A.T$   module $X_2 : A.T$   end
```

Here, X_1 and X_2 have the same signature, but it might be parameterized by different types: each have its own flexroot. This gives the following signature:

```
$\Lambda\omega.\lambda\Psi, \alpha_1, \alpha_2$.sig  module type $T = \lambda\alpha.(\Psi\,\alpha)$ module $X_1 : (\Psi\,\alpha_1)$ module $X_2 : (\Psi\,\alpha_2)$ end
```

By instantiation, we can get a signature where X_1 and X_2 have different types:

```
sig
    module type $T = \lambda\alpha$.sig type $t = \alpha$ end
    module $X_1$ : sig type $t$ = int end
    module $X_2$ : sig type $t$ = bool end
end
```

Kind functions However, first-order kind polymorphism is not enough: due to the presence of higher-order functors, we need higher-order kinds to express all possible forms of sharing between an abstract signature in a functor's codomain and the functor's parameter. To see why, let us consider this higher-order functor:

```
module type $TA$ = sig_A module type  $T = A.T$  module $X : A.T$ end
module $F = (Y : ((Y' : TA) \to TA)) \to$ struct_B  end
```

The functor F could be applied to a functor that either produces a new abstract signature as output or produces a concrete signature that depends on the parameter signature. Therefore, the following signature would be incorrect:

```
module $F$ :$\forall\omega_0.\forall(\Psi_0 : \omega_0 \to \star).\forall(\alpha_0 : \omega_0)$.
           $\forall\omega_1.\forall(\Psi_1 : \omega_1 \to \star).\forall(\alpha_1 : \omega_1)$.
           (sig module type $T = \lambda(\beta : \omega_1).(\Psi_1\,\omega_1)$   module $X : (\Psi_1\,\alpha_1)$ end $\to$
              sig module type $T = \lambda(\beta : \omega_0).(\Psi_0\,\omega_0)$   module $X : (\Psi_0\,\alpha_0)$ end) $\to$ sig  end
```

Indeed, with such signature the functor F could not take as input a functor that produces a concrete signature. Here, we need to skolemize the kind variable to cover all possible cases, turning ω_0 into an higher-order kind Ω. Skolemization happens both at the kind and type levels: Ψ_0 and α_0 are turned into kind-polymorphic higher-order types. This gives:

```
module $F$ :$\forall\Omega.\forall(\Psi_0 : \forall\omega.(\omega \to \star) \to \omega \to (\Omega\,\omega) \to \star).\forall(\alpha_0 : \forall\omega.(\omega \to \star) \to \omega \to (\Omega\,\omega))$.
           $\forall\omega\,.\forall(\Psi_1 : \omega \to \star).\forall(\alpha_1 : \omega)$.
           (sig module type $T = \lambda(\beta : \omega).(\Psi_1\,\omega)$   module $X : (\Psi_1\,\alpha_1)$ end $\to$
              sig module type $T = \lambda(\beta : (\Omega\,\omega)).\Psi_1\,\omega\,\Psi_1\,\alpha_1\,\beta$
                 module $X : (\Psi_0\,\omega\,\Psi_1\,\alpha_1\,(\alpha_0\,\omega\,\Psi_1\,\alpha_1))$ end) $\to$ sig  end
```

Skolemization This introduction of higher-order kinds to treat higher-order functors via skolemization strongly resembles Biswas [1995]'s introduction of higher-order types to treat higher-order functors. While his type system does not have higher-order types (nor applicative functors), he used skolemization to represent all the possible ways a functor's codomain might depend on its domain. This effectively introduces polymorphism over higher-order types, while no higher order types are actually introduced. Similarly, we introduce higher-order kinds to represent all the possible ways the kind of an abstract signatures in a functor's codomain might depend on its domain, while we do not have kind-parametric signatures.

3.5.2 Extension of F^ω

The extended syntax of F^ω is summed up in Figure 23. We extended the system in two ways:

$$\begin{array}{rll} \aleph & := \bullet \mid \aleph \Rightarrow \aleph & \text{(meta-kinds)} \\ \varsigma & := \cdots \mid \varnothing \mid (\varsigma \times \cdots \times \varsigma) & \text{(small kinds)} \\ \kappa & := \cdots \mid \lambda(\omega : \aleph).\kappa \mid \kappa\,\kappa & \text{(large kinds)} \\ \tau & := \cdots \mid \varnothing \mid (\tau \times \cdots \times \tau) & \text{(types)} \\ \Gamma & := \cdots \mid (\omega : \aleph) & \text{(environments)} \end{array}$$

Figure 23: Extension of F^ω with kind functions, kind tuples and meta-kinds.

- **Kind tuples**: We add n-ary kind tuples, with the empty tuple being written $\varnothing$. At the level of types, we add the corresponding empty-tuple type $\varnothing$, and the n-ary constructor for type tuples. Those type tuples are not the type of tuples of core language! They are not at the base kind $\star$, i.e., not term has a type that has a tuple kind.
- **Kind functions and meta-kinds**: We add kind-level lambda and kind applications. In addition, we add an extra layer to "type kind expressions", which we call *meta-kinds* and are denoted with $\aleph$. The base meta-kind is $\bullet$, its the type of plain kinds. Kind function have an arrow meta-kind, written $\aleph \Rightarrow \aleph$. It is not the meta-kind of arrow kinds $\kappa \to \kappa'$ (the kind of type operators), which are of the base meta-kind $\bullet$, but of *kind-level* functions. Kind tuples have a special meta-kind, which indicates the arity of the tuple. It is written $\times_n$.

We add the corresponding typing rules, along with a new judgment of kind-wellformedness, written $\Gamma \vdash \kappa : \aleph$, in Figure 24. The kind-wellformedness rules are very similar to the type-wellformedness rules of F^ω, just pushed one level higher. All the kinds previously considered (the base kind $\star$, kind arrows and polymorphic kinds) are at the base meta-kind $\bullet$. Only kind functions are at an higher meta-kind. The rule for kind application is standard. There are no rules for wellformedness of meta-kinds, they are always wellformed. We introduce β-reduction at the kind level to reduce kind applications:

$$(\lambda(\omega : \aleph).\kappa)\kappa' \rightsquigarrow^\beta \kappa[\omega \mapsto \kappa']$$

We consider kinds up the equivalence defined by the reflexive, symmetric and transitive cloture of β-reduction and renaming of bound kind variables. We extend type-equivalence to inject kind equivalence. The other rules of F^ω typing are unchanged, except for the kind variables that appear with a meta-kind. Finally, we preferably use the variable Ψ for abstract signatures variables for the sake of readability, but it should be read as a normal type variable of the collection α.

Module-type declarations of M^ω For a technical reason that we detail in the next section, we need to change the encoding of the module-type fields of signatures and structures. Up until now, signatures where stored in module-type fields as parametric types, following Russo [2004]:

```
module type $T = \lambda\overline{\alpha}.\mathcal{C}$
```

We switch back to existentially quantified signatures, in the style of Rossberg et al. [2014]:

```
module type $T = \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}$
```

The two presentation are more-or-less equivalent. The only difference lies in the fact that, with existential signatures, the transformation of the binder from $\exists$ to $\forall$ when the signature is used for a functor parameter did not have a logic interpretation. This is a minor technical detail.

$$\Gamma \vdash \star : \bullet \qquad \frac{(\omega : \aleph) \in \Gamma}{\Gamma \vdash \omega : \aleph} \qquad \frac{\Gamma \vdash \kappa : \bullet \qquad \Gamma \vdash \kappa' : \bullet}{\Gamma \vdash \kappa \to \kappa' : \bullet} \qquad \frac{\Gamma, (\omega : \aleph) \vdash \kappa : \bullet}{\Gamma \vdash \forall(\omega : \aleph).\kappa : \bullet}$$

$$\frac{\forall i \in [\![1, n]\!] \,.\, \Gamma \vdash \varsigma_i : \bullet}{\Gamma \vdash \varsigma_1 \times \cdots \times \varsigma_n : \bullet} \qquad \frac{\Gamma, (\omega : \aleph) \vdash \kappa : \aleph'}{\Gamma \vdash \lambda(\omega : \aleph).\kappa : \aleph \Rightarrow \aleph'} \qquad \frac{\Gamma \vdash \kappa : \aleph' \Rightarrow \aleph \qquad \Gamma \vdash \kappa' : \aleph'}{\Gamma \vdash (\kappa\,\kappa') : \aleph}$$

(a) Wellformedness of kinds

$$\frac{\forall i \in [\![1, n]\!] \,.\, \Gamma \vdash \tau_i : \kappa_i}{\Gamma \vdash \tau_1 \times \cdots \times \tau_n : \kappa_1 \times \cdots \times \kappa_n} \qquad \Gamma \vdash \varnothing : \varnothing$$

(b) Extension to type-wellformedness of F^ω (in addition to the rules of Figure 18)

Figure 24: Extension of F^ω wellformedness

3.5.3 Typing rules

Equipped with this extension of the type system, we can present the typing rules for M^ω. For the sake of readability, we leave meta-kinds and most kinds implicits. The core change is in the signature elaboration, where we use higher-order kinds to represent all the possible instantiation of a signature. The key mechanisms for kind variables are similar to type variables: the variables are introduced, extruded and skolemized. In addition, there is a specific mechanism of *lowering* that removes kind parametricity in positive position. We discuss the key rules and mechanisms below.

Introduction The only point where kind-parametric abstract module types are introduced is at an abstract module-type binding:

M-Typ-Decl-ModTypeAbs

$$\Gamma \vdash_A (\texttt{module type } T = A.T) : \Lambda\omega.\,\lambda(\Psi : \omega \to \star).\,\mathsf{module\ type}\ T = \lambda(\alpha : \omega).\,(\Psi\,\alpha)$$

The abstract signature variable Ψ is lifted, while the set of type parameters α is left inside the module type declaration. As we can see on this rule, elaboration of signatures (and declarations) not only a parametric signature but a kind-parametric, type parametric signature of the form $\Lambda\overline{\omega}.\,\lambda\overline{\alpha}.\mathcal{C}$.

Extrusion When combining declarations together in a sequence, the kind variables are merged together in front of the declarations as displayed by the following rule:

M-Typ-Decl-Seq

$$\frac{\Gamma \vdash_A \mathtt{D}_1 : \Lambda\overline{\omega}_1.\,\lambda\overline{\alpha}_1.\mathcal{D}_1 \qquad \Gamma, \overline{\omega}_1, \overline{\alpha}_1, A.\mathcal{D}_1 \vdash_A \overline{\mathtt{D}_2} : \Lambda\overline{\omega}_2.\,\lambda\overline{\alpha}_2.\overline{\mathcal{D}_2}}{\Gamma \vdash_A \mathtt{D}_1, \overline{\mathtt{D}_2} : \Lambda\overline{\omega}_1, \overline{\omega}_2.\,\lambda\overline{\alpha}_1, \overline{\alpha}_2.\ \mathcal{D}_1, \overline{\mathcal{D}_2}}$$

Skolemization When elaborating the signature of an applicative functor, we skolemize both the kind variables and the abstract type variables (which contain abstract signature variables) of the functor's body (written here with kinds to explicit the skolemization):

M-Typ-Sig-AppFct

$$\frac{\Gamma \vdash \mathtt{S}_1 : \Lambda\overline{\omega}_1.\,\lambda\overline{\alpha}_1.\mathcal{C}_1 \qquad \Gamma, \overline{\omega}_1, \overline{\alpha}_1, (Y : \mathcal{C}_1) \vdash \mathtt{S}_2 : \Lambda\overline{\omega}_2.\,\lambda\overline{(\alpha_2 : \kappa_2)}.\mathcal{C}_2}{\Gamma \vdash (Y : \mathtt{S}_1) \to \mathtt{S}_2 : \Lambda\overline{\omega}'_2.\,\lambda\overline{(\alpha'_2 : \forall\omega.\,\kappa_2)}.\,\Big(\forall\overline{\omega}_1.\forall\overline{\alpha}_1.\mathcal{C}_1 \to \Big(\mathcal{C}_2\Big[\overline{\omega_2 \mapsto (\omega'_2\,\overline{\omega}_1)}\Big]\Big[\overline{\alpha_2 \mapsto (\alpha'_2\,\overline{\omega}_1\,\overline{\alpha}_1)}\Big]\Big)\Big)}$$

The kind variables coming from the functor's body $\overline{\omega}_2$ are skolemized into $\overline{\omega}'_2$, and each occurrence ω_2 is replaced by $(\omega'_2\,\overline{\omega}_1)$. Abstract type variables coming from the functor's body $\overline{(\alpha_2 : \kappa_2)}$ are turned into kind-polymorphic higher-order types $\overline{(\alpha_2' : \forall\omega.\,\kappa_2)}$; each occurrence α_2 is replaced by $(\alpha_2'\,\overline{\omega}_1\,\overline{\alpha}_2)$. Finally, we can see that the lambda quantification of the functor domain is turned into an universal quantification.

Lowering A new mechanism specific to kind variables is *lowering*. If we consider type parameters of signature that are bound by a lambda binder, they are either turned into universal quantification when the signature is used as functor parameter, or into existential quantification when the signature is used for ascription or as the codomain of a generative functor. Kind variables behave differently. While we still want kind-parametricity for functors (as displayed above), we do not want existential kinds when using a kind-parametric signature for ascription or the body of generative functors. We use instead *lowering* to remove kind parameters, and "lower" them to the base kind $\star$. Let us start with the rule for signature of generative functors:

$$\textsc{M-Typ-Sig-GenFct}\quad \frac{\Gamma \vdash \mathtt{S} : \mathcal{S} \qquad \mathcal{S} = \overline{\Lambda\omega.\,\Lambda\Psi.\,\lambda\overline{\alpha}}.\lambda\overline{\beta}.\mathcal{C} \qquad \mathcal{S}^{\downarrow} = \exists^{\blacktriangledown}\overline{(\Psi' : \star)}, \overline{\beta}.\mathcal{S}\,\overline{(\varnothing\,(\lambda(\gamma : \varnothing).\,\Psi')\,\overline{\varnothing})}\,\overline{\beta}}{\Gamma \vdash () \to \mathtt{S} : \mathcal{S}^{\downarrow}}$$

First, the codomain signature $\mathtt{S}$ is elaborated. We reorder the types and kind variables to group together each kind variable ω with the associated signature variable $(\Psi : \omega \to \star)$ and the flexroots $\overline{\alpha}$. The other type variables $\overline{\beta}$ are left untouched. Then, we introduce a new abstract signature variable $(\Psi' : \star)$ at the base kind for each abstract signature variable Ψ (which might be at an higher kind). Finally, we get the lowered signature $\mathcal{S}^{\downarrow}$ by instantiating in $\mathcal{S}$:

- each kind variables ω by the empty tuple kind $\varnothing$
- each abstract signature variable Ψ by the constant type function $(\lambda(\gamma : \varnothing).\,\Psi')$
- each flexroot α by the empty tuple type $\varnothing$; higher-order flexroots are instantiated by constant type functions that return the empty tuple: $\lambda(\_ : \varnothing).\,\varnothing$

Lowering and subtyping The rule for ascription in module expressions M-Typ-Mod-Ascr also features lowering:

$$\textsc{M-Typ-Mod-Ascr}\quad \frac{\begin{array}{c}\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathtt{S} : \mathcal{S} \qquad \mathcal{S} = \overline{\Lambda\omega.\,\Lambda\Psi.\,\lambda\overline{\alpha}}.\lambda\overline{\beta}.\mathcal{C} \\ \mathcal{S}^{\downarrow} = \lambda\overline{\Psi'}, \overline{\beta}.\mathcal{S}\,\overline{(\varnothing\,(\lambda(\gamma : \varnothing).\,\Psi')\,\overline{\varnothing})}\,\overline{\beta} \qquad \Gamma \vdash \mathcal{C} < (\mathcal{S}^{\downarrow}\,(\overline{\exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}'})\,\overline{\tau})\end{array}}{\Gamma \vdash^{\triangledown} (P : \mathtt{S}) : \exists^{\triangledown}\overline{\Psi'}, \overline{\beta}.\,\left(\mathcal{S}^{\downarrow}\,\overline{\Psi'}\,\overline{\beta}\right)}$$

The elaborated signature $\mathcal{S}$ is lowered into $\mathcal{S}^{\downarrow}$ similarly to M-Typ-Sig-GenFct. Then, the subtyping is checked between the signature $\mathcal{C}$ of the path P and the lowered signature, instantiated by a list of concrete types $\overline{\tau}$ and a list of existential signatures $\exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}'$. The returned signature is the lowered one $\mathcal{S}^{\downarrow}$, with an eta-expansion to turn the lambda binder into an existential one. Subtyping between the lowered signature $\mathcal{S}^{\downarrow}$ rather than the kind-polymorphic signature $\mathtt{S}$ is more restrictive – there are programs that would not typecheck – but we do it to prevent the introduction of existential kinds. To understand why the instantiation with existential signatures works, we show the rule in action on an example.

Example 3.5.1. *We show the Rule* M-Typ-Mod-Ascr *in action. Let us assume that the path P has the following signature $\mathcal{C}$:*

```
𝒞 ≜ sig
        module type T = ∃▼α.sig type t = α end
        module X_1 : sig type t = int end
        module X_2 : sig type t = bool end
        module F : ∀β.sig type t = β end → sig type t = (β list) end
    end
```

Now, we consider the signature `S` *and its elaboration $\mathcal{S}$:*

```
S ≜ sig_A
      module type T =
          sig_B type t = B.t end
      module X_1 : A.T
      module X_2 : A.T
      module F : (Y : A.T) → A.T
    end
```

```
𝒮 ≜ Λω.λ(Ψ : ω → ⋆).λα_1, α_2, φ.sig
        module type T = ∃▼α.(Ψ α)

        module X_1 : (Ψ α_1)
        module X_2 : (Ψ α_2)
        module F : ∀β.(Ψ β) → (Ψ (φ β))
    end
```

Here, we do no want to instantiate the signature $\mathcal{S}$ directly, as we would get as output a signature with an existential kind. Instead, we first lower the signature to $\mathcal{S}^{\downarrow}$, defined as :

$$\mathcal{S}^{\downarrow} \triangleq \lambda(\Psi' : \star).(\mathcal{S}\ \varnothing\ \varnothing\ \varnothing\ (\lambda(\_ : \varnothing).\varnothing))$$

After β-reduction of the empty-tuple type arguments, we obtain the following:

```
𝒮↓ ≜ λ(Ψ' : ⋆).sig
        module type T = Ψ'
        module X_1 : Ψ'
        module X_2 : Ψ'
        module F : Ψ' → Ψ'
    end
```

Finally, before subtyping $\mathcal{C}$ and $\mathcal{S}^{\downarrow}$, we instantiate the latter with $(\exists^{\blacktriangledown}\alpha.\,\textsf{sig type}\ t = \alpha\ \textsf{end})$. The subtyping is then made field by field. For the module fields, we have:

$$\textsf{module}\ X_1 : \textsf{sig type}\ t = \texttt{int}\ \textsf{end} < \exists^{\blacktriangledown}\alpha.\,\textsf{module}\ X_1 : \textsf{sig type}\ t = \alpha\ \textsf{end}$$
$$\textsf{module}\ X_2 : \textsf{sig type}\ t = \texttt{bool}\ \textsf{end} < \exists^{\blacktriangledown}\alpha.\,\textsf{module}\ X_2 : \textsf{sig type}\ t = \alpha\ \textsf{end}$$

This is shown easily by instantiating α with `int` *and* `bool` *respectively. For the functor field, we have the following successive subtyping relations:*

$$\begin{aligned} & \textsf{module}\ F : \forall\beta.\textsf{sig type}\ t = \beta\ \textsf{end} \to \textsf{sig type}\ t = (\beta\ \textsf{list})\ \textsf{end} \\ < \ & \textsf{module}\ F : \forall\beta.\textsf{sig type}\ t = \beta\ \textsf{end} \to (\exists^{\blacktriangledown}\alpha.\,\textsf{sig type}\ t = \alpha\ \textsf{end}) && (1) \\ < \ & \textsf{module}\ F : (\exists^{\blacktriangledown}\beta.\,\textsf{sig type}\ t = \beta\ \textsf{end}) \to (\exists^{\blacktriangledown}\alpha.\,\textsf{sig type}\ t = \alpha\ \textsf{end}) && (2) \end{aligned}$$

The subtyping relation (1) is normal covariance and instantiation, while (2) is justified by the fact the parameter's signature is universally quantified, and that β does not appear free in the codomain:

$$(\forall\alpha.\sigma \to \tau) < (\exists\alpha.\sigma) \to \tau \qquad (\alpha \notin \textsf{fv}(\tau))$$

Finally, the result signature is $\mathcal{S}^{\downarrow}$ with a (transparent) existential quantifier:

$$\Gamma \vdash (P : \texttt{S}) : \exists^{\triangledown}\Psi'.(\mathcal{S}^{\downarrow}\ \Psi')$$

Overall, the instantiation by an existential signature creates a signature where abstract types have been "un-extruded" and "un-skolemized", and this signature is a supertype of $\mathcal{C}$. But those abstract types are hidden away in the resulting signature, abstracted by Ψ'. If "un-extrusion" and "un-skolemization" is not possible, then the subtyping fails and there is a typechecking *error. This would happen for instance with the following signature (inside $\mathcal{C}$):*

```
module F : sig type t = int end → sig type t = (int list) end
```

The subtyping step (1) would still work, but (2) would fail: it would make the functor more polymorphic than it really is!

Functor application The last technical point is the following rule for functor application:

$$\frac{\Gamma \vdash P : \forall \omega'. \forall \overline{\alpha}. \mathcal{C}_a \to \mathcal{C} \qquad \Gamma \vdash P' : \mathcal{C}' \qquad \Gamma \vdash \mathcal{C}' < \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}]\,[\overline{\omega} \mapsto \overline{\varsigma}]}{\Gamma \vdash P(P') : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]\,[\overline{\omega} \mapsto \overline{\varsigma}]}\ \textsc{M-Typ-Mod-AppApp}$$

Here, we extend the *instantiation* mechanism of subtyping to support both instantiation of kinds and instantiation of types. The instantiation of kinds is guided by the module-type fields and does not pose a problem for decidability.

Discussion We believe this extension of M^ω supports the main use-cases for abstract signatures. It adds some complexity, but overall the treatment of kind variables is quite similar to type variables. However, if we where to relax the *simple abstract signatures* criterion and allow instantiation of abstract signatures by signatures containing abstract module-types, the system would become much more involved. Simple abstract signatures can be seen as the level 1 of predicative instantiation. At level 2, we would already need higher-order meta-kinds and several more quantified variables. We believe that the added complexity is not justified regarding the absence of known use-cases.

3.6 Discussion

M^ω signatures are more expressive than source signature, but they may also keep *too much information*, revealing the history of the module operations. This may lead to an inferred signature that is not anchorable, while intuitively providing the same type-sharing information as a simpler, anchorable signature. This typically happens when a type variable has become "unreachable", only appearing in sub-expressions. We have identified two such patterns.

In the following, we use the notation $\mathcal{C}[\alpha, \dots]$ to indicate that α appears freely in $\mathcal{C}$ and the notation $\mathcal{C}[(\varphi\,\alpha), \dots]$ to indicate that α appears only in the subexpression $(\varphi\,\alpha)$. In particular, $\mathcal{C}[\varphi, (\varphi\,\alpha)]$ means that α only appears as an argument of φ in $\mathcal{C}$.

Loss of a type argument. The signature $\exists\,\varphi, \alpha.\ \mathcal{C}[\varphi, (\varphi\,\alpha)]$, could be obtained by exporting a functor (providing φ) along with a type obtained by applying this functor to an argument that has latter been hidden. The application $\varphi\,\alpha$ keeps trace that the type was obtained by applying φ to α. However, since the argument α is not accessible, this information became useless. By subtyping, we could safely give the module the simpler signature $\exists\,\varphi, \beta.\ \mathcal{C}[\varphi, \beta]$ cutting the (original) link to the functor , which we can state as a subtyping relationship:

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

Anchoring the left-hand side will fail since α cannot be anchored, while anchoring the right-hand might succeed. We do not currently allow this simplification during anchoring, since both signatures are not isomorphic in F^ω as there is no coercion going in the opposite direction.

Loss of a type operator. Similarly, the application of a functor may be exported while the functor itself became unreachable. For instance, with two applications of the same functor, we may have a signature of the form $\exists\,\varphi, \alpha, \beta.\ \mathcal{C}[\alpha, \beta, (\varphi\,\alpha), (\varphi\,\beta)]$, which is a subtype of, but not isomorphic to $\exists \alpha, \beta, \alpha', \beta'.\ \mathcal{C}[\alpha, \beta, \alpha', \beta']$. In the special case where the functor is called only once, the signature would be of the form $\exists\,\varphi, \alpha.\, \mathcal{C}[\alpha, (\varphi\,\alpha)]$, which is actually isomorphic to $\exists \alpha, \alpha'.\, \mathcal{C}[\alpha, \alpha']$.

Chapter 4

ZipML

Overview In this paper, we propose ZipML, a fully-syntactic specification of an OCaml-like module system that supports both generative and applicative (higher-order) functors, opaque and transparent ascription, type and module-type definitions, extended with transparent signatures and the new concept of signature zippers.

Floating fields follow the way the user would solve signature avoidance manually, by adding extra fields in structures. However, while this pollutes the namespace with fields that were not meant to be visible, ZipML floating fields are added automatically and can only be used internally: they are not accessible to the user and are absent at runtime. We also propose an internal mechanism to simplify floating fields and drop them after they became unreferenced. This mechanism is actually an internalized, improved counterpart of the signature avoidance resolution of M^ω.

Our contributions are:

- The introduction of a syntactic type system for a full-fledged OCaml-like module language, including both generative and applicative functors, and extended with transparent ascription.
- A new concept of *floating fields*, implemented as *signature zippers*, that enables to internalize and avoid or delay signature avoidance.
- An equivalence criterion for signature avoidance resolution without loss of type sharing, along with the description of an algorithm to compute such resolution.
- A formal treatment of type and module type definitions that are kept during inference and in inferred signatures.
- A regular, delayed treatment of strengthening that prevents useless inlining of module type definitions and increases sharing inside the typechecker.
- A soundness proof by elaboration of ZipML into M^ω.

4.1 Elaboration vs Fully Syntactic systems

Yet, ML modules system remained a case of advanced theoretical work that has not yet made it into real-world implementation: the SML and OCaml compilers are not based on an elaboration in F^ω. Notably, OCaml relies on a *path-based* system, initially described by Leroy [1995, 1994], which has not been extended to more complex constructs. The SML compiler has an official specification Milner et al. [1997] that has also evolved over the years, but not towards an elaboration in F^ω (see MacQueen et al. [2020] for more details). Furthermore, the *F-ing* (Rossberg et al. [2014]) approach lead to a significant gap between the user writable "source" signatures and their internal representations in F^ω. Users are either, required to think in terms of the elaboration, while still writing types in the surface language, or rely only on the syntactic types as the internal representation is hidden away with a reverse translation. The first option seemed unsatisfactory, as the elaboration can be quite involved and may pollute the typing environment with a lot of variables. For the second option, the reverse translation called *anchoring* Blaudeau et al. [2024] is non trivial: it must undo the extrusion and the skolemization of existential types to rebuild abstract type fields. In practice, it poses some challenges:

- The reverse translation can fail because the elaboration language is more expressive than the source ML signatures, and some inferred signatures are not expressible in the source signature syntax. This issue, called *signature avoidance*, is discussed in more details below. Those cases would trigger a specific class of typechecking errors which might be tricky to grasp for the user, as it might require to expose the internal representation.

- For printing messages in case of other typechecking errors (not due to avoidance), the reverse translation should be extended to invalid signatures, which might again be problematic.

- Even when the typechecking succeeds, the M^ω Blaudeau et al. [2024] type-system combined with the anchoring is not *fully syntactic* in the sense of Shao [1999]: if a module expression admits a signature, not all sub-expressions necessarily do. This might be counter-intuitive for the user, as the simple act of exposing a sub-module would make the system fail in non-trivial ways.

However, it is not surprising that the elaboration approach was appealing: purely syntactic systems for ML modules are known to be tricky to design, often leading to large systems with numerous rules, and hard to prove sound, often requiring complex semantic objects. Among other, we refer the reader to Harper et al. [1989]; Harper and Lillibridge [1994]; Dreyer et al. [2003]. We identify four main challenges, of increasing difficulty. 1. Prevent inlining of intermediate type and module type definitions. This is crucial to print concise interfaces respecting user intent. This aspect has been left behind in the whole *F-ing* (Rossberg et al. [2014]) line of works so far. 2. Handle applicative functors and module identities. 3. Maintain proper sharing of types, notably via a rewriting rule called *strengthening* Leroy [1994]. 4. Solve the *signature avoidance* problem.

In the rest of this introduction, we briefly discuss our design for a new syntactic module system with an OCaml-like syntax called ZipML that solves those issues, starting with avoidance.

4.2 An introduction to floating fields

The main novelty of ZipML is the introduction of floating fields as a way of dealing with the signature avoidance problem. Floating fields provide additional expressiveness that allows to describe all inferred signatures.

We use OCaml-like syntax Leroy et al. [2023] for examples. However, we use self-references for signatures[1] to refer to other components of themselves. That is, while we would write in OCaml, e.g.,:

```
sig type t type u = t → t val f : u end
```

we instead write:

```
sig (A) type t = A.t type u = A.t → A.t val f : A.u end
```

The first occurrence of `A` is a binder that refers to the whole signature, so that all internal references to a field of that signature go through this self-reference. This is also used for abstract types who are represented as aliases to themselves as `type t = A.t`.

For sake of readability and conciseness, we also extend the syntax with the following syntactic sugar that is not available in OCaml. Since OCaml only allows projection on paths $P.X$[2], the general case M`.X` must be encoded as `let Z = M in Z.X` where `let Z =` M_1 `in` M_2 may itself be encoded in OCaml as `struct open (struct module Z =` M_1 `end) include` M_2 `end`.

Signature avoidance is best illustrated on projections, using these syntactic extensions. Typically, signature avoidance occurs when a type `t` that appears in the inferred signature become out of scope after projection:

```
let module M = struct
    type t module Z = struct let l : M.t list = [] end
  end in M.Z
```

```
  The value "l" has no valid type if "M" is hidden.
```

Instead, ZipML will return the following signature:[3]

```
< B : type t = B.t > sig (A) val f : B.t end
```

The highlighted expression is what we call a floating field; it is not a signature, but the content of a signature that is bound to the self-reference `B`.

Intuitively, this type is obtained as follows. Before projection, the signature of `M` is:

```
sig (B) type t = B.t module Z : sig (A) val l : B.t list end end
```

When projecting on field `Z`, we would like to return `sig (A) val l : B.t list end`, which is however ill-formed as it refers to field `t` that has been lost. The solution is to keep the type of `t` as a floating field, which gives exactly the signature just above.

Another typical situation of signature avoidance is when a hidden type as `t` is used deeper in another type definition as in:

```
(struct type t module Z = struct type u = t list type v = t list end end).Z
```

```
sig (A) type u type v end                                  (* over-abstaction *)
```

Here, instead of failing, OCaml returns two abstract types `u` and `v`, not only loosing their list structure, but also also forgetting that these are actually equal types. We say that this is erroneously solved by *over-abstraction*. In ZipML, we would first return the following signature:

```
< B : type t = B.t > sig (A) val type u = B.t list type v = B.t list end
```

This is a correct answer. In particular, no information has been lost. The signature may be simplified using signature equivalence: since `A.v` is equal to `A.u` it can be made an alias to `A.u`.

[1]ZipML also uses self-references in structures, but we do not in examples to keep closer to OCaml syntax.

[2]This limitation encourages users to name intermediate structures, circumventing the avoidance problem.

[3]By default, input programs are colored in blue; errors in red; output signatures in green with zippers in yellow. We may still temporarily used other colors for to emphasize specific subexpressions.

```
< B : type t = B.t > sig (A) val type u = B.t list type v = A.u end
```

Still, we cannot get rid of the floating component here. This does not matter however as typechecking may continue with floating components, which are integrated into the type system.

4.2.1 Typechecking with floating fields

We demonstrate the process of typechecking with floating fields on a concrete example showcased below. In this example, we define a module `M` with some nested submodules and type aliases. Finally, we project a deeply nested module $M.Y.Z$. If type inference proceeds too rashly, this projection will lead to loss of type information, as type aliases might be forgotten. Indeed, this is exactly what happens in current OCaml, which erroneously solves signature avoidance by forgetting the information that fields `v` and `w` are actually the same abstract type.

```
let M = struct
  module X = struct type t end
  module Y = struct type u = X.t list module Z = struct type v = X.t type w = u end
end in
M.Y.Z
```

```
sig type v type w end
```

In ZipML, we first return the following signature $\mathtt{S}_0$ with floating fields:

```
< D : module X : sig (A) type t = A.t > < B : type u = D.X.t list >
sig (C) type v = D.X.t type w = B.u end
```

which we then simplify to this equivalent final form $\mathtt{S}_1$:

```
sig (C) type v type w = C.v list end
```

We explain this process step by step. Let us first look at the signature $\mathtt{S}_0$: the first line introduces two local typing environments of *floating fields* that may be directly referenced from the signature that is returned on the second line. Floating fields are not present at runtime. Hence, their order does not matter except for well-formedness, and they may be dropped when not referenced anymore.

To understand how $\mathtt{S}_0$ was generated, let us look at the signature of M, before the projection:

```
M : sig (D)
  module X : sig (A) type t = A.t end
  module Y : sig (B)
    type u = D.X.t list
    module Z : sig (C) type v = D.X.t type w = B.u end
  end
end
```

It is of the form $S_D\left[S_B\left[\mathtt{S}_Z\right]\right]$, where the signature $\mathtt{S}_Z$ is placed in the context $S_D\left[S_B\left[\cdot\right]\right]$ where S_D and S_B are the outer and inner contexts. Unfortunately, the signature $\mathtt{S}_Z$ that we would like to return for the projection `M.Y.Z` is ill-formed, as it refers to both outer contexts S_D and S_B. This situation is an instance of the signature avoidance problem: we want to return $\mathtt{S}_Z$ while *avoiding* to mention the unreachable fields from S_D and S_B. It is usually solved by rewriting the signature $\mathtt{S}_Z$ into an equivalent signature that does not depend on the context—whenever this is possible and fail otherwise.

We take a different stance: we type the projection by returning a *zipper* of the surrounding context $\langle S_D\left[S_B\right]\rangle\,\mathtt{S}_Z$, or equivalently, $\langle D : S_D\rangle\,\langle B : S_B\rangle\,\mathtt{S}_Z$. Thus, projection never fails, avoiding (or delaying) the signature avoidance problem. The zipper notation gives an immediate access to $\mathtt{S}_Z$ but still allowing $\mathtt{S}_Z$ to refer to its local contexts S_D and S_B.

Let us proceed and type projections one after the other. For the first projection `M.Y`, we return the signature $\mathtt{S}_2$ equal to $\langle D : S_D\rangle\,(S_B\,[\mathtt{S}_Z])$, which is, in concrete syntax:

```
< D : module X : sig (A) type t = A.t end > S_B [S_Z]
```

We actually dropped the access to the hole in `module X =` $[\cdot]$ as well as the field following the hole, since they all become useless after the projection. For the second projection `(M.Y).Z` we proceed similarly. Projecting `M.Y` of type $S_B\,[\mathtt{S}_Z]$ on Z, we get $\langle B : S_B\rangle\,\mathtt{S}_Z$. Popping the local environment S_D, we then return the signature $\langle S_D\rangle\,\langle S_B\rangle\,\mathtt{S}_Z$, which exactly corresponds to the signature $\mathtt{S}_0$ returned by ZIPML before simplification (given above).

Interestingly, each component of the zipped context S_D and S_B can be directly accessed from $\mathtt{S}_Z$ via their self-reference names, respectively D and B. Hence $\mathtt{S}_Z$ need not be changed—provided we have chosen disjoint self-references while zipping.

The next step to is transform $\mathtt{S}_0$ into an equivalent but simpler signature, that is, one with fewer floating fields. We may first inline `B.u` in the field `w` of $\mathtt{S}_Z$, leading to the signature:

```
< D : module X : sig (A) type t = A.t end > < B : type u = D.X.t list >
sig (C) type v = D.X.t type w = D.X.t list end
```

The floating component `B` is now unreferenced from the signature C and can be dropped. Since the type `C.v` is equal to `D.X.t`, the field `C.w` can be rewritten as `C.v list`. We obtain the signature $\mathtt{S}_3$:

```
< D : module X : sig (A) type t = A.t end > sig (C) type v = D.X.t type w = C.v list end
```

The signature can be read back as the projection of the unzipped signature (using some reserved field $\mathbb{Z}$ for the lost projection path).

```
(sig (D) module X : sig (A) type t = R.v end
         module ℤ : sig (C) type v = D.X.t type w = C.v list end  end).ℤ
```

Currently `C.v` is an alias to the floating abstract type `D.X.t`, which comes first. However, since the module X should be understood as a floating field not present at runtime, we may move it after field $\mathbb{Z}$, making `C.v` become the defining occurrence for the abstract type and let `D.X.t` be an alias to `C.v`. The key here is that the two unzipped signatures are subtype of one another, hence equivalent.

```
(sig (D) ℤ : sig (C) type v type w = C.v end  module X : sig (A) type t = R.v end end).ℤ
```

We can represent the displacement of fields directly on zippers, provided we enrich them to contain two parts, which we separated by `/`: before and after the position of interest:

```
< D : / module X : sig (A) type t = R.v end >  sig (C) type v type w = C.v end
```

The key is that floating fields that come after the hole can always be dropped as they are not present at runtime and no longer referenced from the signature. Here, the resulting zipper D contains no more fields and can be dropped altogether. We then obtain exactly the signature $\mathtt{S}_1$ (repeated below), which is thus equivalent to $\mathtt{S}_3$ and then to $\mathtt{S}_0$.

```
sig (C) type v type w = C.v list end
```

In this case, we were able to eliminate all floating fields and therefore successfully and correctly resolve signature avoidance, while OCAML incorrectly removes the equality between `v` and `w` types. In general, the simplification may remove some but not all floating fields. This is fine, as we are able to pursue typechecking in presence of floating fields, which may perhaps be dropped later on. For example, while $\mathtt{S}_3$ could be simplified by removing its floating field, this was not the case of the signature $\mathtt{S}_2$, since the floating field `D.X.t` is referenced in $S_B\,[\mathtt{S}_Z]$

Path and Prefix

$$\begin{array}{lll} P ::= & Q.X & \text{(Module)} \\ & \mid Y & \text{(Module Parameter)} \\ & \mid P(P) & \text{(Applicative application)} \\ & \mid P.A & \text{(Zipper access)} \end{array}$$

$$Q ::= A \mid P$$

Module Expression

$$\begin{array}{lll} \mathtt{M} ::= & P & \text{(Path)} \\ & \mid \mathtt{M}.X & \text{(Anonymous projection)} \\ & \mid (P : \tilde{\mathtt{S}}) & \text{(Ascription)} \\ & \mid P() & \text{(Generative application)} \\ & \mid () \rightarrow \mathtt{M} & \text{(Generative functor)} \\ & \mid (Y : \tilde{\mathtt{S}}) \rightarrow \mathtt{M} & \text{(Applicative functor)} \\ & \mid \mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end} & \text{(Structure)} \end{array}$$

Binding

$$\begin{array}{lll} \mathtt{B} ::= & \mathtt{let}\, x = \mathtt{e} & \text{(Value)} \\ & \mid \mathtt{type}\, t = \mathtt{u} & \text{(Type)} \\ & \mid \mathtt{module}\, X = \mathtt{M} & \text{(Module)} \\ & \mid \mathtt{module\ type}\ T = \tilde{\mathtt{S}} & \text{(Module type)} \end{array}$$

Identifier

$$I ::= x \mid t \mid X \mid T \mid Y$$

Contexts

$$\Gamma ::= \varnothing \mid \Gamma, A.I : S \mid \Gamma, Y : S \mid \Gamma, \gamma$$

Zippers

$$\gamma ::= \varnothing \mid A : \overline{\mathtt{D}} \mid \gamma \,;\, \gamma$$

Signature

$$\begin{array}{lll} \mathtt{S} ::= & \langle \gamma \rangle\, \mathtt{S} & \text{(Zipped type)} \\ & \mid \tilde{\mathtt{S}} & \text{(Unzipped type)} \\ \tilde{\mathtt{S}} ::= & (= P < \tilde{\mathtt{S}}) & \text{(Transparent signature)} \\ & \mid Q.T & \text{(Module type)} \\ & \mid () \rightarrow \mathtt{S} & \text{(Generative functor)} \\ & \mid (Y : \tilde{\mathtt{S}}) \rightarrow \mathtt{S} & \text{(Applicative functor)} \\ & \mid \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end} & \text{(Structural signature)} \end{array}$$

Declaration

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

Core language types and expression

$$\mathtt{e} ::= \cdots \mid Q.x \quad \text{(Qualified value)} \qquad \mathtt{u} ::= \cdots \mid Q.t \quad \text{(Qualified type)}$$

Figure 25: Syntax of ZipML

as `D.X.t` before being aliased. If we disallowed floating fields, we would have failed at that program point—or used over abstraction as in OCaml in this case and probably failed later on as a consequence of over-abstraction.

Interestingly, when typechecking an ascription ($\mathtt{M} : \mathtt{S}$), the type returned in case of success will be the elaboration of the *source signature* `S`, which never contains floating fields. Thus, an OCaml library defined by an implementation file `M` together with an interface file `S` will never return floating fields in ZipML, even if internally some signatures will carry floating fields. In other cases, we may return an answer with floating fields, giving the user the possibility to remove them via a signature ascription. As a last resort, we may still keep them until link time—or fail, leaving both options to the language designer—or the user.

4.3 Formal presentation

4.3.1 Grammar

The syntax of ZipML is given in Figure 25. It reuses the syntax of OCaml, but with a few differences in notation, the addition of transparent ascriptions, and our key contribution: signatures with *floating fields*, also called *zippers*.

As meta-syntactic conventions, we use lowercase letters (x, t, ...) for elements of the core language, and uppercase letters (X, T, ...) for modules. We also use slanted letters (I, A, Q, ...) for identifiers and paths and upright letters (`M`, `e`, `D`, ...) for syntactic categories. Finally, we designate lists with an overbar, such as a list of bindings $\overline{\mathtt{B}}$. Let us now walk through these syntactic categories.

Classical Module Constructs As in OCaml, the module language is based on structures, signatures, functors and projections. These can be found in the **Module Expressions** ($\texttt{M}$) and **Signatures** ($\texttt{S}$), and the usual module fields (expression, types, modules, module types) in **Bindings** ($\texttt{B}$) and **Declarations** ($\texttt{D}$). As in OCaml we use a special unit argument to distinguish generative functors from applicative functors, in both module expressions and signatures. Both structures and signatures are annotated with a self-reference A, explained below. Signatures contain a new form $(= P < \tilde{\texttt{S}})$ for a transparent signature which has them the identity of P but with the interface of $\texttt{S}$. Transparent signatures allow to express both aliasing and transparent ascription.

Self-references A special class of identifiers A range over self-references, which are variables used in both module structures $\texttt{struct}_A\ \overline{\texttt{B}}\ \texttt{end}$ and structural signatures $\texttt{sig}_A\ \overline{\texttt{D}}\ \texttt{end}$ to refer to the structure or the signature itself from fields $\overline{\texttt{B}}$ or $\overline{\texttt{D}}$: The annotation is the binding occurrence, their scope extends to $\overline{\texttt{B}}$ or $\overline{\texttt{D}}$, and they can be consistently renamed. This avoids the standard telescope notation and facilitates renaming operations. Thanks to self-references, field names are no longer binders and behaves rather as record fields. We do not allow overriding of fields in signatures, which is standard in module systems. By contrast with common usage, we also avoid overriding in structures and therefore require all field names to be disjoint in the same structure, for sake of simplicity. This restriction is just a matter of name resolution that has little interaction with typechecking and could easily be relaxed.

Self-references are also used to represent abstract types: all type fields are of the form $\texttt{type}\, t = \texttt{u}$ where $\texttt{u}$ may be a core language type or an alias $P.t$ including the case $A.t$ where A is the self-reference of the field under consideration, which in this case means that $A.t$ is an abstract type.

We may omit the self-reference and just write $\texttt{sig D end}$ for $\texttt{sig}_A\ \texttt{D end}$ when A does not appear free in $\texttt{D}$. Conversely, when we write $\texttt{sig D end}$, we should read $\texttt{sig}_A\ \texttt{D end}$ where the anonymous self-reference A is chosen fresh for $\texttt{D}$.

Identifiers and Paths Paths P are a hybrid mechanism to access modules, statically. By static, we mean that we always know statically the identity of the module a path refers to. They may access the environment directly, either functor parameters Y or module fields $A.I$ where I spans over any identifier. They may also access module fields by projection $P.X$. In the absence of applicative functors and floating fields, this would be sufficient. With applicative functors, a path may also designate the result of an immediate module application $P(P)$. Finally, floating environments are accessed with $P.A$. The letter Q designates paths that are either local (A) or distant (P). We also extend the core language with qualified values $Q.x$ and types $Q.t$.

Note that we distinguish module names X from module parameters Y. The latter are variables, can be renamed (when in binding position), and will be substituted during type-checking. By contrast, names are never used in a binding position.

Contexts Typing contexts Γ bind module fields to declarations and functor parameters to signatures. In typing contexts module fields are always prefixed by a self-reference. As we disallow shadowing, every $A.I$ in Γ must be unique. For convenience, we may write $\Gamma, A.\overline{\texttt{D}}$ for the sequence $\Gamma, \overline{A.\texttt{D}}$. Finally, context, may also contain floating fields Γ, γ. By associativity, we identify $\Gamma, (A.\overline{\texttt{D}}, A'.\overline{\texttt{D}}')$ and $(\Gamma, A, \overline{\texttt{D}}), A'.\overline{\texttt{D}}'$.

Zippers Finally, the novelty is the introduction of zipped signatures $\langle\gamma\rangle\,\texttt{S}$ used to introduce zippers γ, which are sequences of floating fields $A : \overline{\texttt{D}}$. In a zipped signature $\langle A : \texttt{D}\rangle\,\texttt{S}$, the self-reference A, which is used to access fields of $\texttt{D}$ from $\texttt{S}$, can be renamed consistently. That is, we identify $\left\langle A : \overline{\texttt{D}}\right\rangle \texttt{S}$ and $\left\langle A' : \overline{\texttt{D}}[A \mapsto A']\right\rangle \texttt{S}[A \mapsto A']$ provided A' is fresh for $\overline{\texttt{D}}$ and $\texttt{S}$. Of course, a zipper taken alone cannot be renamed. The concatenation of zippers $\gamma_1 ; \gamma_2$ is only defined when the domains of γ_1 and γ_2 are disjoint where their domain is defined as follows:

$$\mathsf{dom}(\emptyset) = \emptyset \qquad \mathsf{dom}(A : \overline{\mathtt{D}}) = \{A\} \qquad \mathsf{dom}(\gamma_1 ; \gamma_2) = \mathsf{dom}(\gamma_1) \uplus \mathsf{dom}(\gamma_2)$$

We may then see a zipper as a map from self-references to declarations and define $\gamma(A)$ accordingly. The concatenation of zippers ";" is associative and the empty zipper $\emptyset$ is a neutral element. We identify $\mathtt{S}$ and $\langle\emptyset\rangle\,\mathtt{S}$ and $\langle\gamma_1\rangle\,\langle\gamma_2\rangle\,\mathtt{S}$ and $\langle\gamma_1 ; \gamma_2\rangle\,\mathtt{S}$ whenever γ_1 and γ_2 have disjoint domains. Therefore a signature S can always be written as $\langle\gamma\rangle\,\tilde{\mathtt{S}}$ where $\tilde{\mathtt{S}}$ is an unzipped signature and γ concatenates all consecutive zippers or is $\emptyset$ if none. The introduction of zippers requires a new form of path $P.A$, invalid in source programs, to access floating fields.[4] Finally, we define $\tilde{\mathtt{S}}$ as signature without an initial zipper but where subterms may contain zippers. Note that zipper may not appear under a transparent ascription or a functor parameter.

Invariants We also define several syntactic subcategories of signatures to capture some invariants.[5] The head **v**alue form $\mathtt{S}^{\mathrm{v}}$ of a signature gives the actual shape of a signature, which is either a structural signature or a functor. The head **n**ormal form $\mathtt{S}^{\mathrm{n}}$ is similar, but still contains the *identity* of a signature, if it has one, via a transparent ascription.

$$\mathtt{S}^{\mathrm{v}} ::= \mathtt{sig}_A\,\overline{\mathtt{D}}\,\mathtt{end} \mid (Y : \mathtt{S}) \to \mathtt{S} \mid () \to \mathtt{S} \qquad \mathtt{S}^{\mathrm{n}} ::= \mathtt{S}^{\mathrm{v}} \mid (= P < \mathtt{S}^{\mathrm{v}})$$

Notice that head normal forms (and value forms) are superficial and signatures appearing under value forms may themselves be any signature. Hence, they may contain inner zippers.

A *Transparent* signature $\mathbb{S}$ is a generalization of the syntactic form $(= P < \tilde{\mathtt{S}})$ that also allows transparent ascriptions to be placed inside zippers. The definition is the following:

$$\mathbb{S} ::= \langle\mathbb{G}\rangle\,\mathbb{S} \mid \tilde{\mathbb{S}} \qquad \tilde{\mathbb{S}} ::= (= P < \tilde{\mathtt{S}}) \qquad \mathbb{G} ::= A : \overline{\mathbb{D}} \mid \mathbb{G} ; \mathbb{G} \mid \emptyset$$
$$\mathbb{D} ::= \mathtt{module}\,X : \mathbb{S} \mid \mathtt{val}\,x : \mathtt{u} \mid \mathtt{module\ type}\ T = \mathtt{S} \mid \mathtt{type}\,t = \mathtt{u}$$

Finally, we let $\tilde{P}$ stand for paths P that do not contain any (direct or recursive) zipper access (of the form $P'.A$). Consistently, $\tilde{Q}$ means $\tilde{P}$ or A and $\tilde{\mathtt{u}}$ means $\mathtt{u}$ where all Q's are actually $\tilde{Q}$'s.

Syntactic choices and syntactic sugar Our grammar has some superficial syntactic restrictions, i.e., that simplifies the presentation without reducing expressiveness. In particular, we only allow applications of paths to paths. The more general application $\mathtt{M}_1(\mathtt{M}_2)$ may be encoded as syntactic sugar for $(\mathtt{struct}_A\ \mathtt{module}\ X_1 = \mathtt{M}_1\ \mathtt{module}\ X_2 = \mathtt{M}_2\ \mathtt{module}\ X = A.X_1(A.X_2)\ \mathtt{end}).X$. Indeed, our projection $\mathtt{M}.X$ is unrestricted. By contrast, OCaml requires $\mathtt{M}$ to be a path P and must encode general projection with an application. Our choice is more general, as it does not require any additional type annotation.

Similarly, $\mathtt{M}$ is currently restricted to be a path $\tilde{P}$ in ascription $(\mathtt{M} : \tilde{\mathtt{S}})$, but the general case can be encoded as $(\mathtt{struct}_A\,\mathtt{module}\ X_1 = \mathtt{M}\,\mathtt{module}\ X = (A.X_1 : \tilde{\mathtt{S}})\ \mathtt{end}).X$.

Note some grammatical ambiguity: $P.X$ may be a projection from a path P or from a module expression $\mathtt{M}$ which may itself be a path. This is not an issue as their typing will be the same.

[4]In the absence of floating fields, self references could only be at the origin of a path Q.

[5]For sake of readability, we do not always use the most precise syntactic categories and sometimes just write $\mathtt{S}$ when $\mathtt{S}$ may actually be of a more specific form. Conversely, we may use subcategories to restrict the application of a rule that only applies for signatures of a specific shape.

4.3.2 Full zippers

In fact, the zippers we have defined are *partial* zippers, a special case of *full* zippers. Full zippers are only used in Section 4.4, but introducing them now helps illustrate the role and treatment of floating fields as zippers. In full zippers, floating components are of of the form $A : \mathtt{D} \cdot X \cdot \mathtt{D}'$. That is, the field they store are split into before-fields and after-fields separated by a module name. We may define an *unzip* operation on full zippers that rebuilds a structural signature from a zipped signature:

$$\begin{aligned} \mathsf{unzip}\ \left\langle A : \overline{\mathtt{D}} \cdot X \cdot \overline{\mathtt{D}}' \right\rangle \mathtt{S} &\triangleq \mathtt{sig}_A\ \overline{\mathtt{D}}, \mathtt{module}\ X : \mathtt{S}, \overline{\mathtt{D}}'\ \mathtt{end} \\ \mathsf{unzip}\ \langle \gamma_1 \mathbin{;} \gamma_2 \rangle\, \mathtt{S} &\triangleq \mathsf{unzip}\ \langle \gamma_1 \rangle\, (\mathsf{unzip}\ \langle \gamma_2 \rangle\, \mathtt{S}) \\ \mathsf{unzip}\ \langle \emptyset \rangle\, \mathtt{S} &\triangleq \mathtt{S} \end{aligned}$$

Conversely, given a structural signature $\mathtt{S}$, $\mathtt{S} \cdot X$ is the zipper positioned at the submodule X:

$$(\mathtt{sig}_A\ \overline{\mathtt{D}}, \mathtt{module}\ X : \mathtt{S}, \overline{\mathtt{D}}'\ \mathtt{end}) \cdot X \triangleq \left\langle A : \overline{\mathtt{D}} \cdot X \cdot \overline{\mathtt{D}}' \right\rangle \mathtt{S} \qquad \mathtt{S} \cdot (X.\overline{X}) \triangleq (\mathtt{S} \cdot X) \cdot \overline{X} \qquad \mathtt{S} \cdot \emptyset \triangleq \mathtt{S}$$

We may also define the path of a zipper as the sequence of projections that returns the zipped signature from its unzipped one:

$$\mathsf{path}\ (A : \overline{\mathtt{D}} \cdot X \cdot \overline{\mathtt{D}}') = X \qquad\qquad \mathsf{path}\ (\gamma \mathbin{;} \gamma') = (\mathsf{path}\ \gamma) \cdot (\mathsf{path}\ \gamma')$$

Then, we have $(\mathsf{unzip}\ (\langle \gamma \rangle\, \mathtt{S})) \cdot (\mathsf{path}\ \gamma)$ equal to $\langle \gamma \rangle\, \mathtt{S}$.

A partial zipper $A : \overline{\mathtt{D}}$ is a full zipper $A : \overline{\mathtt{D}} \cdot X \cdot \mathtt{D}'$ where we have dropped both the projection-field name X and the after-fields $\mathtt{D}'$, because we do not need them. The idea is to directly hold the signature $\mathtt{S}$ that was at a field X in some structural signature and still have access to the fields preceding X. We do not care about the field name X nor the fields following X, on which $\mathtt{S}$ does not depend. Partial zippers are smaller, slightly easier to manipulate, and sufficient to define most typing judgments. We may also view a partial zipper $A : \overline{\mathtt{D}}$ as the full zipper $\mathtt{D} \cdot \mathbb{Z} \cdot \emptyset$, using a reserved field $\mathbb{Z}$, and extend unzipping to partial zippers.

4.3.3 Strengthening and typing environment

Strengthening is a key operation in path-based module systems: intuitively, it is used to give module signatures and abstract types an identity that will then be preserved by aliasing.

In ZipML, we may use transparent ascription $(= P < \mathtt{S})$ to express strengthening of the signature $\mathtt{S}$ by the path P, which effectively gives an identity, i.e., path P, to an existing signature $\mathtt{S}$—under a subtyping condition between the signature of P and $\mathtt{S}$. In other words, transparent ascription allows strengthening to be directly represented in the syntax of signatures. This can be advantageously used to implement strengthening lazily, by contrast with OCaml's eager version.[6]

Given a signature $\mathtt{S}$ and a path P, we consider two flavors of strengthening: delayed strengthening $\mathtt{S} /\!/ P$ and shallow strengthening $\mathtt{S} / P$, defined on Figure 26. Shallow strengthening is only defined on signatures in head normal forms and is called during normalization to push strengthening just one level down. It then delegates the work to delayed strengthening $\mathtt{S} /\!/ P$, which will insert a transparent ascription in a signature, if there is not one already, and push it under zippers if any. Since the very purpose of strengthening is to make signatures transparent, both form of strengthening indeed return a transparent signature $\mathbb{S}$. The rules for delayed signature strengthening should be read in order of appearance, as they pattern match on the head of the signature:

- Delayed strengthening stops at a transparent ascription, since it is already transparent.

[6]There is actually a proposal to add lazy strengthening in OCaml for efficiency purposes. **XDR [Should we give a URL?]**

Delayed strengthening

$$\begin{aligned}
(= P' < \mathtt{S}) \mathbin{//} P &\triangleq (= P' < \mathtt{S}) \\
\langle \gamma \,;\, \gamma' \rangle\, \mathtt{S} \mathbin{//} P &\triangleq (\langle\gamma\rangle\,(\langle\gamma'\rangle\,\mathtt{S})) \mathbin{//} P \\
\langle A : \overline{\mathtt{D}} \rangle\, \mathtt{S} \mathbin{//} P &\triangleq \langle A : \overline{\mathtt{D}} \mathbin{//} P\rangle\,(\mathtt{S}[A \mapsto P.A] \mathbin{//} P) \\
\mathtt{S} \mathbin{//} P &\triangleq (= P < \mathtt{S})
\end{aligned}$$

Shallow strengthening

$$\begin{aligned}
\mathtt{sig}_A\, \overline{\mathtt{D}}\, \mathtt{end} \,/\, P &\triangleq \mathtt{sig}_A\, \overline{\mathtt{D}[A \mapsto P] \mathbin{//} P}\, \mathtt{end} \\
(Y : \mathtt{S}_a) \to \mathtt{S} \,/\, P &\triangleq (Y : \mathtt{S}_a) \to (\mathtt{S} \mathbin{//} P(Y)) \\
() \to \mathtt{S} \,/\, P &\triangleq () \to \mathtt{S}
\end{aligned}$$

Declaration strengthening

$$\begin{aligned}
(\mathtt{val}\, x : \mathtt{u}) \mathbin{//} P &\triangleq \mathtt{val}\, x : \mathtt{u} &\qquad (\mathtt{module}\, X : \mathtt{S}) \mathbin{//} P &\triangleq \mathtt{module}\, X : (\mathtt{S} \mathbin{//} P.X) \\
(\mathtt{type}\, t = \mathtt{u}) \mathbin{//} P &\triangleq \mathtt{type}\, t = \mathtt{u} &\qquad (\mathtt{module\ type}\ T = \mathtt{S}) \mathbin{//} P &\triangleq \mathtt{module\ type}\ T = \mathtt{S}
\end{aligned}$$

Context strengthening

$$\begin{aligned}
\Gamma \uplus (Y : \mathtt{S}) &\triangleq \Gamma, Y : (\mathtt{S} \mathbin{//} Y) &\qquad \Gamma \uplus (\mathtt{module}\, A.X : \mathtt{S}) &\triangleq \Gamma, \mathtt{module}\, A.X : (\mathtt{S} \mathbin{//} A.X) \\
\Gamma \uplus (A : \overline{\mathtt{D}} \,;\, \gamma) &\triangleq (\Gamma \uplus A : (\overline{\mathtt{D}} \mathbin{//} A)), \gamma
\end{aligned}$$

Figure 26: Strengthening (delayed by default) – $\mathtt{S} \,/\, P$ and $\mathtt{S} \mathbin{//} P$

- It strengthens zipped signatures step by step. For a compound zipper, we decompose it as two successive zipping. For a simple zipper, we strengthen both the zipper and the signature in which we replaced the self-reference A by the strengthened path. We ignored the empty zipper, which is neutral for zipping. Notice that strengthening commutes with unzipping. That is, $\mathtt{unzip}\ (\langle\gamma\rangle\,\mathtt{S} \mathbin{//} P)$ is equal to $(\mathtt{unzip}\ \langle\gamma\rangle\,\mathtt{S}) \mathbin{//} P$.

- Otherwise, delayed strengthening just inserts a transparent ascription, which is actually the materialization of the delaying.

We also defined delayed strengthening on declarations, which is called by strengthening on zippers, shallow strengthening, and context strengthening: it pushes delayed strengthening inside module declarations and does nothing on other fields. (Module type definitions never have an identity, so they are not strengthened.)

Finally, Figure 26 defines a helper binary operation $\uplus$ that strengthens bindings as they enter the context. We use it to maintain the invariant that all signatures entering the typing environment are transparent signatures $\mathbb{S}$, which can be duplicated without lost of sharing.

Strengthening strengthen identities but not types. This is a key.

For example, if $\Gamma \vdash \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end}$, then $\Gamma \uplus \Gamma A.X \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end}$ is equal to

4.3.4 Path typing

Since paths include projections and applications, which require some type checking, their types are not immediate to deduce. Moreover, module types may themselves be definitions that must sometimes be inlined to be analyzed.

The path typing judgment $\Gamma \vdash P : \mathbb{S}$ is defined on Figure 27. A key invariant is that the signature $\mathbb{S}$ is transparent.

Rules Typ-P-Arg and Typ-P-Module are straightforward lookup. Rule Typ-P-Zip accesses a zipper through its self-reference. It may be surprising that the signature $\mathtt{sig}_A\ \overline{\mathbb{D}}\ \mathtt{end}$ of $P.A$ need not be put back inside the zipper $\mathbb{G}$. The key here is that the signature $\langle\mathbb{G}\rangle\,\mathbb{S}$, and hence the declaration $\mathbb{D}$, are transparent and no longer depend on the local zipper context $\mathbb{G}$, but only on the environment Γ.

Rule Typ-P-Norm means that path typing is defined up to signature normalization (see below): we may, but need not, normalize the signature to progress, e.g., when the signature $\mathtt{S}$ is a module type.

$$\textsc{Typ-P-Arg}\ \frac{Y : \mathbb{S} \in \Gamma}{\Gamma \vdash Y : \mathbb{S}} \qquad \textsc{Typ-P-Module}\ \frac{\texttt{module}\, A.X : \mathbb{S} \in \Gamma}{\Gamma \vdash A.X : \mathbb{S}} \qquad \textsc{Typ-P-Zip}\ \frac{\Gamma \vdash P : \langle \mathbb{G} \rangle\, \mathbb{S} \quad \mathbb{G}(A) = \overline{\mathbb{D}}}{\Gamma \vdash P.A : \texttt{sig}\ \overline{\mathbb{D}}\ \texttt{end}} \qquad \textsc{Typ-P-Norm}\ \frac{\Gamma \vdash P : \mathbb{S} \quad \Gamma \vdash \mathbb{S} \downarrow \mathbb{S}'}{\Gamma \vdash P : \mathbb{S}'}$$

$$\textsc{Typ-P-Proj}\ \frac{\Gamma \vdash P \triangleright \texttt{sig}\ \overline{\mathbb{D}}\ \texttt{end} \quad \texttt{module}\, X : \mathbb{S} \in \overline{\mathbb{D}}}{\Gamma \vdash P.X : \mathbb{S}} \qquad \textsc{Typ-P-AppA}\ \frac{\Gamma \vdash P \triangleright (Y : \mathtt{S}_a) \to \mathbb{S} \quad \Gamma \vdash P' : \mathbb{S}' \quad \Gamma \vdash \mathbb{S}' \leq \mathtt{S}_a}{\Gamma \vdash P(P') : \mathbb{S}[Y \mapsto P']}$$

Figure 27: Path typing – $\Gamma \vdash P : \mathtt{S}$

$$\textsc{Res-P-Id}\ \frac{\Gamma \vdash P : \langle \mathbb{G} \rangle\, (= P' < \tilde{\mathtt{S}})}{\Gamma \vdash P \triangleright P'} \qquad \textsc{Res-P-Val}\ \frac{\Gamma \vdash P : \langle \mathbb{G} \rangle\, (= P' < \tilde{\mathtt{S}}) \qquad \tilde{\mathbb{S}} = \tilde{\mathtt{S}} \,/\, P'}{\Gamma \vdash P \triangleright \tilde{\mathbb{S}}}$$

Figure 28: Path Resolution – $\Gamma \vdash P \triangleright P'$ and $\Gamma \vdash P \triangleright \tilde{\mathbb{S}}$

The two remaining rules use path resolution to analyze the signature. In Rule Typ-P-Proj, we ignored the self-reference to the signature of P since we know it is transparent, hence $\mathbb{S}$ does not use its self-reference and is well-formed in Γ. Typing of functors (Rule Typ-P-AppA) requires the domain signature to be a super type of the argument signature. We then return the codomain signature after substitution of the argument P by the parameter Y.

Notice that path typing is not deterministic: there may be two signatures $\mathbb{S}_1$ and $\mathbb{S}_2$ such that $\Gamma \vdash P : \mathbb{S}_1$ and $\Gamma \vdash P : \mathbb{S}_2$, where $\mathbb{S}_1$ and $\mathbb{S}_2$ are not α-equivalent. First, one may contain a zipper that the other will have dropped. Then, one may or may not call normalization. Finally, normalization may be partial, inlining some module type definitions, but not necessarily all of them.

Path resolution A signature contains dual information, a structure and an identity. Path resolution is composed of two independent single-rule helper judgments, defined on Figure 28, that extract this information. Path resolution accesses information inside the signature, so we must skip the zippers that surrounds it if any. In both rules, $\mathbb{G}$ is meant for that, assuming that zippers have been flattened and defaulting to the empty zipper. The judgment $\Gamma \vdash P \triangleright P'$ (Res-P-Id) just returns the path P' of the transparent signature of P. The judgment $\Gamma \vdash P \triangleright \tilde{\mathbb{S}}$ returns the unzipped (hence the tilde) part of the signature of P, but strengthened with its identity P'. The judgment does not require $\tilde{\mathbb{S}}$ to be in head normal form, but may reach a normal form when asked to do so. We thus might use resolution to pattern match on the resulting signature, forcing computation of a head normal form.

4.3.5 Normalization

The judgment $\Gamma \vdash \mathtt{S} \downarrow S'$ *allows* the (head) normalization of a signature $\mathtt{S}$ into $\mathtt{S}'$, which inlines module type definitions, but only one step at a time. Hence, to achieve the head normal form, we may call normalization repeatedly. Rule Norm-S-Zip normalizes the signature part of a zipped signature. We never normalize the zipper itself, as we will first access the zipper and normalize it afterwards.**XDR [Perhaps, we should, which would allow simplification on the zipper side.]** Rules Norm-S-Trans-Some and Norm-S-Trans-None allows normalization under a transparent ascription. If the (partial) normal form of $\mathtt{S}$ is itself a transparent ascription, we return it as is; otherwise, we return its strengthened version by the resolved path P'. Finally, rules Norm-S-LocalModType and Norm-S-PathModType expand a module type definition.

$$\textsc{Norm-S-Zip}\ \frac{\Gamma \uplus \gamma \vdash \mathtt{S} \downarrow \mathtt{S}'}{\Gamma \vdash \langle\gamma\rangle\, \mathtt{S} \downarrow \langle\gamma\rangle\, \mathtt{S}'} \qquad \textsc{Norm-S-Trans-Some}\ \frac{\Gamma \vdash \mathtt{S} \downarrow (= P' < \mathtt{S})}{\Gamma \vdash (= P < \mathtt{S}) \downarrow (= P' < \mathtt{S}')} \qquad \textsc{Norm-S-Trans-None}\ \frac{\Gamma \vdash P \triangleright P' \qquad \Gamma \vdash \mathtt{S} \downarrow \mathtt{S}'}{\Gamma \vdash (= P < \mathtt{S}) \downarrow (= P' < \mathtt{S}')}$$

$$\textsc{Norm-Typ-Res}\ \frac{\Gamma \vdash P \triangleright P'}{\Gamma \vdash P.t \downarrow P'.t} \qquad \textsc{Norm-S-LocalModType}\ \frac{\mathtt{module\ type}\ A.T = \mathtt{S} \in \Gamma}{\Gamma \vdash A.T \downarrow \mathtt{S}} \qquad \textsc{Norm-S-PathModType}\ \frac{\Gamma \vdash P \triangleright \mathtt{sig}\ \overline{\mathtt{D}}\ \mathtt{end} \quad \mathtt{module\ type}\ T = \mathtt{S} \in \overline{\mathtt{D}}}{\Gamma \vdash P.T \downarrow \mathtt{S}} \qquad \textsc{Norm-Typ-Local}\ \frac{\mathtt{type}\, A.t = \mathtt{u} \in \Gamma}{\Gamma \vdash A.t \downarrow \mathtt{u}}$$

$$\textsc{Norm-Typ-Path}\ \frac{\Gamma \vdash P \triangleright \mathtt{sig}\ \overline{\mathtt{D}}\ \mathtt{end} \quad \mathtt{type}\, t = \mathtt{u} \in \overline{\mathtt{D}}}{\Gamma \vdash P.t \downarrow \mathtt{u}}$$

Figure 29: Signature and type normalization – $\Gamma \vdash \mathtt{S} \downarrow \mathtt{S}'$

$$\textsc{Sub-S-Norm}\ \frac{\Gamma \vdash \mathtt{S}_1 \downarrow \mathtt{S}_1' \qquad \Gamma \vdash \mathtt{S}_2 \downarrow \mathtt{S}_2' \qquad \Gamma \vdash \mathtt{S}_1' \leqslant \mathtt{S}_2'}{\Gamma \vdash \mathtt{S}_1 \leqslant \mathtt{S}_2} \qquad \textsc{Sub-S-Zipper}\ \frac{\Gamma \uplus \gamma \vdash \mathtt{S}_1 \leqslant \mathtt{S}_2}{\Gamma \vdash \langle\gamma\rangle\, \mathtt{S}_1 \leqslant \mathtt{S}_2} \qquad \textsc{Sub-S-TrAscr}\ \frac{\Gamma \vdash \mathtt{S}_1 \,/\, P \leqslant \mathtt{S}_2 \,/\, P}{\Gamma \vdash (= P < \mathtt{S}_1) \leqslant (= P < \mathtt{S}_2)}$$

$$\textsc{Sub-S-LooseAlias}\ \frac{\Gamma \vdash \mathtt{S}_1 \,/\, P \leqslant \mathtt{S}_2}{\Gamma \vdash (= P < \mathtt{S}_1) \leqslant \mathtt{S}_2} \qquad \textsc{Sub-S-FctG}\ \frac{\Gamma \vdash \mathtt{S}_1 \leqslant \mathtt{S}_2}{\Gamma \vdash () \to \mathtt{S}_1 \leqslant () \to \mathtt{S}_2} \qquad \textsc{Sub-S-FctG}\ \frac{\Gamma \vdash \mathtt{S}_{a_2} \leqslant \mathtt{S}_{a_1} \qquad \Gamma \uplus Y : \mathtt{S}_{a_2} \vdash \mathtt{S}_1 \leqslant \mathtt{S}_2}{\Gamma \vdash (Y : \mathtt{S}_{a_1}) \to \mathtt{S}_1 \leqslant (Y : \mathtt{S}_{a_2}) \to \mathtt{S}_2}$$

$$\textsc{Sub-S-Sig}\ \frac{\overline{\mathtt{D}_0} \sqsubset_{\leq} \overline{\mathtt{D}_1} \qquad \Gamma \uplus A : \overline{\mathtt{D}_1} \vdash \overline{\mathtt{D}_0} \leqslant \overline{\mathtt{D}_2}}{\Gamma \vdash \mathtt{sig}_A\ \overline{\mathtt{D}_1}\ \mathtt{end} \leqslant \mathtt{sig}_A\ \overline{\mathtt{D}_2}\ \mathtt{end}} \qquad \textsc{Sub-S-DynEq}\ \frac{\Gamma \vdash \mathtt{S} \lesssim \mathtt{S}' \qquad \Gamma \vdash \mathtt{S}' \lesssim \mathtt{S}}{\Gamma \vdash \mathtt{S} \approx \mathtt{S}'} \qquad \textsc{Sub-T-Norm}\ \frac{\Gamma \vdash \mathtt{u}_1 \downarrow \mathtt{u} \qquad \Gamma \vdash \mathtt{u}_2 \downarrow \mathtt{u}}{\Gamma \vdash \mathtt{u}_1 \leqslant \mathtt{u}_2}$$

$$\textsc{Sub-D-Mod}\ \frac{\Gamma \vdash \mathtt{S}_1 \leqslant \mathtt{S}_2}{\Gamma \vdash \mathtt{module}\, X : \mathtt{S}_1 \leqslant \mathtt{module}\, X : \mathtt{S}_2} \qquad \textsc{Sub-D-Val}\ \frac{\Gamma \vdash \mathtt{u}_1 \leqslant \mathtt{u}_2}{\Gamma \vdash \mathtt{val}\, x : \mathtt{u}_1 \leqslant \mathtt{val}\, x : \mathtt{u}_2}$$

$$\textsc{Sub-D-ModType}\ \frac{\Gamma \vdash \mathtt{S}_1 \approx \mathtt{S}_2}{\Gamma \vdash \mathtt{module\ type}\ T = \mathtt{S}_1 \leqslant \mathtt{module\ type}\ T = \mathtt{S}_2} \qquad \textsc{Sub-D-Type}\ \frac{\Gamma \vdash \mathtt{u}_1 \approx \mathtt{u}_2}{\Gamma \vdash \mathtt{type}\, t = \mathtt{u}_1 \leqslant \mathtt{type}\, t = \mathtt{u}_2}$$

Figure 30: Subtyping – $\Gamma \vdash \mathtt{S} \leqslant \tilde{\mathtt{S}}$

The judgment $\Gamma \vdash P.t \downarrow \mathtt{u}$ allows normalization of types. Rules Norm-Typ-Res allows the resolution of the path P while rules Norm-Typ-Local and Norm-Typ-Path inlined the type definition $Q.t$.

4.3.6 Subtyping judgments

The subtyping judgment $\Gamma \vdash \mathtt{S} \leqslant \tilde{\mathtt{S}}$ is only defined when the target signature is an elaborated signature $\tilde{\mathtt{S}}$ and therefore is well-formed, does not contain zippers nor zipper accesses (nor chains of transparent ascriptions). The left-hand side signature $\mathtt{S}$ is also a well-formed signature resulting either from the elaboration of a source signature or from typechecking. Hence, it may contain zippers. The same invariants extend to auxiliary subtyping judgments for declarations and paths.

Signature subtyping is defined on Figure 30. This is actually a parametric definition that depends on the $\sqsubset$ binary operation between declarations, used to parameterized the width subtyping Rule Sub-S-Sig. This allows us to decide whether and how fields can be rearranged during subtyping. For simplicity, we wrote $\leqslant$ instead of $\leqslant_{\sqsubset}$ in the definition of the typing rules.

Rules must be read by case analysis on the left-hand side signature. Rule Sub-S-Norm is not syntax directed and can be applied anywhere and repeatedly to perform normalization before subtyping. For example, there is no rule matching a signature $Q.T$ on the left-hand side. But normalization allows inlining the definition before checking for subtyping. A zipper may only occur on the left-hand side. Rule Sub-S-Zipper pushes the zipper in the environment (as the signature $\mathtt{S}_1$ may not be transparent) and pursues with subtyping. For other cases, we require the left-hand side to be in head normal form. When the left-hand side is a transparent ascription the right-hand side may also be a transparent ascription, in which case, we check subtyping between the respective signatures, but after pushing strengthening one level-down, lazily (Sub-S-TrAscr). Otherwise, we drop the transparent ascription only the left-hand side, which amounts to loose its transparency, hence increase abstraction, as allowed by subtyping (Sub-S-LooseAlias). In the remaining cases, the left-hand side is a head value form and the right-hand side must have the same shape. Functor types are contravariant. For subtyping explicit signatures (Sub-S-Sig), we may sometimes forget and reorder some fields, as specified by the parameter relation $\sqsubset_{\leqslant}$. Namely, we must find a sequence $\overline{\mathtt{D}_0}$ related to $\overline{\mathtt{D}_1}$ by $\sqsubset_{\leqslant}$, and subtyped pointwise with $\overline{\mathtt{D}_2}$, but in an environment extended with $A.\overline{\mathtt{D}_1}$.

The main subtyping relation is $\leqslant$ defined as $\leqslant_{\subseteq}$, i.e., using $\subseteq$ for $\sqsubset_{\leqslant}$. Namely, $\overline{\mathtt{D}_0}$ can be any subset of $\overline{\mathtt{D}_1}$ where fields may appear in a different order. This relation is (usually) not code-free. Indeed, to ensure fast accesses, the representation of a structure (usually) follows the structure of its signature. That is, its runtime fields (modules and values) should be exactly the same and appear in the same order as in its signature. Thus, reordering a signature requires reordering the structure's representation accordingly. We also define a code-free version of subtyping, $\lesssim$, where the relation $\sqsubset_{\lesssim}$ is defined as $\overline{\mathtt{D}_0} \sqsubset_{\lesssim} \overline{\mathtt{D}_1} \triangleq \overline{\mathtt{D}_0} \subseteq \overline{\mathtt{D}_1} \wedge \mathsf{dyn}(\overline{\mathtt{D}_0}) = \mathsf{dyn}(\overline{\mathtt{D}_1})$ and $\mathsf{dyn}(\overline{\mathtt{D}})$ returns the subsequence of $\overline{\mathtt{D}}$ composed of dynamic fields (modules and values) only (appearing in the same order). The subtyping relation $\lesssim$ is used in Rule Sub-S-DynEq to defined $\approx$ on signatures as the kernel of $\lesssim$.

Subtyping uses two other helper judgments, for type and declaration subtyping. There is a single rule Sub-T-Norm for type subtyping that injects head type normalization into the subtyping relation, which is the pending of Rule Sub-S-Norm. In fact, this rule should also be made available in the subtyping relation of the core language, which should be a congruent preorder. For module declarations (Rule Sub-D-Mod), we just require subtyping covariantly. Rule Sub-D-Val for core language values is similar, requiring subtyping in the core language. In OCaml, this would reduce to core-language type-scheme specialization, which we haven't formalized.

Since module types may be used in both covariant and contravariant positions, the rule Sub-D-Modtype requests subtyping in both directions. Moreover, since this subtyping could occur deeply inside terms, we use code-free type equivalence. Notice that if signatures were fully inlined, subtyping would never see the names of definitions but their original inlined expansion and covariance would suffice. **XDR [Concrete example is needed]** The same remark applies to core-language type fields, which are also used as definitions and may appear both co- and contravariantly. Hence, Rule Sub-D-Type requires $\Gamma \vdash \mathtt{u}_1 \approx \mathtt{u}_2$ which means code-free subtyping in both directions, i.e., $\Gamma \vdash \mathtt{u}_1 \leqslant \mathtt{u}_2$ and $\Gamma \vdash \mathtt{u}_2 \leqslant \mathtt{u}_1$.

4.3.7 Signature typing

Source signatures provided by users are not necessarily well formed, and thus must be typed, using the judgment $\Gamma \vdash \tilde{\mathtt{S}}$, defined on Figure 31. The resulting source signature $\tilde{\mathtt{S}}$ is zipper free. Since this is also enforced by not having a typing rule for zipped signatures, we have not enforced the syntactic subcategories and just use $\mathtt{S}$ and $\mathtt{D}$ for signatures and declarations,

$$\textsc{Typ-S-ModType}\ \frac{\Gamma \vdash \tilde{Q}.T : \mathtt{S}}{\Gamma \vdash \tilde{Q}.T} \qquad \textsc{Typ-S-GenFct}\ \frac{\Gamma \vdash \mathtt{S}}{\Gamma \vdash () \to \mathtt{S}} \qquad \textsc{Typ-S-AppFct}\ \frac{\Gamma \vdash \mathtt{S}_a \qquad \Gamma \uplus (Y : \mathtt{S}_a) \vdash \mathtt{S}}{\Gamma \vdash (Y : \mathtt{S}_a) \to \mathtt{S}}$$

$$\textsc{Typ-S-Ascr}\ \frac{\Gamma \vdash \mathtt{S} \qquad \Gamma \vdash \tilde{P} : \mathbb{S} \qquad \Gamma \vdash \mathbb{S} \leqslant \mathtt{S}}{\Gamma \vdash (= \tilde{P} < \mathtt{S})} \qquad \textsc{Typ-S-Str}\ \frac{\Gamma \vdash_A \overline{\mathtt{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end}} \qquad \textsc{Typ-D-Val}\ \frac{\Gamma \vdash \tilde{\mathtt{u}}}{\Gamma \vdash_A (\mathtt{val}\, x : \tilde{\mathtt{u}})} \qquad \textsc{Typ-D-Type}\ \frac{\Gamma \vdash \tilde{\mathtt{u}}}{\Gamma \vdash_A (\mathtt{type}\, t = \tilde{\mathtt{u}})}$$

$$\textsc{Typ-D-TypeAbs}\ \Gamma \vdash_A (\mathtt{type}\, t = A.t) \qquad \textsc{Typ-D-Mod}\ \frac{\Gamma \vdash \mathtt{S}}{\Gamma \vdash_A (\mathtt{module}\, X : \mathtt{S})} \qquad \textsc{Typ-D-ModType}\ \frac{\Gamma \vdash \mathtt{S}}{\Gamma \vdash_A (\mathtt{module\ type}\ T = \mathtt{S})} \qquad \textsc{Typ-D-Empty}\ \Gamma \vdash_A \varnothing$$

$$\textsc{Typ-D-Seq}\ \frac{\Gamma \vdash_A \mathtt{D}_0 \qquad \Gamma \uplus A.\mathtt{D}'_0 \vdash_A \overline{\mathtt{D}}}{\Gamma \vdash_A (\mathtt{D}_0, \overline{\mathtt{D}})}$$

Figure 31: Signature typing (all signatures are $\tilde{\mathtt{S}}$) – $\Gamma \vdash \tilde{\mathtt{S}}$

for the sake of readability. The signature $\tilde{\mathtt{S}}$ should not have zipper accesses either, which is enforced by using the restricted syntactic categories $\tilde{Q}$ and $\tilde{P}$ for paths.

Rule Typ-S-ModType uses path typing to check the well-formedness of paths. Rule Typ-S-Ascr must also check that the signature of path $\tilde{P}$ is a subtype of the signature $\mathtt{S}$. Rule Typ-S-Str for structural signatures delays most of the work to the elaboration judgment $\Gamma \vdash_A \overline{\mathtt{D}}$ for declarations, which carries the self-reference variable A which should be chosen fresh for Γ, as it now appears free in declarations $\overline{\mathtt{D}}$. Rule Typ-D-Seq for sequence of declarations pushes $A.\mathtt{D}_0$ in the context while typing the remaining sequence $\overline{\mathtt{D}}$. All the other rules are straightforward.

In practice, typing of signatures could simplify them on the fly, typically removing chains of transparent ascriptions if any. This would then require to replace the typing judgment $\Gamma \vdash \mathtt{S}$ by an elaboration judgment $\Gamma \vdash \mathtt{S} : \mathtt{S}'$ that returns an elaborated signature $\mathtt{S}'$. In fact, this would be necessary if we allowed declarations `open` $\mathtt{S}$ and `include` $\mathtt{S}$ that should always be elaborated. We have not included them, but the type system has been designed to allow them.

4.3.8 Module typing

The typing judgment $\Gamma \overset{\diamondsuit}{\vdash} \mathtt{M} : \mathtt{S}$ for module expressions is given on Figure 32. The $\diamondsuit$ symbol is a metavariable for modes that ranges over the applicative (or *transparent*) mode $\triangledown$ and the generative (or *opaque*) mode $\blacktriangledown$. Rule Typ-M-Mode means that we may always consider an applicative judgment as a generative one. This is a floating rule that can be applied at any time. Judgments for pure module expressions can be treated either as applicative or generative, hence they use the $\diamondsuit$ metavariable. Many rules use the same metavariable $\diamondsuit$ in premises and conclusion, which then stand for the same mode. This implies that if the premise can only be proved in generative mode, it will also be the case for the conclusion.

Typing a path $\tilde{P}$ (which should not contain zipper accesses) as a module expression (Rule Typ-M-Path) calls the path typing rule defined earlier (with no mode symbol), which always returns a transparent signature $(= P' < \mathtt{S})$. However, we return the "more recent" identity $(= \tilde{P} < \mathtt{S})$, as this is probably the one the user would like to see; besides, the older identities can always be recovered by path resolution, while the reverse is not be possible.

A signature ascription $(\tilde{P} : \tilde{\mathtt{S}})$ gives the source path $\tilde{P}$ the (elaborated) signature of the source signature $\tilde{\mathtt{S}}$ after checking that is it indeed a supertype of the signature of $\tilde{P}$ alone (Rule Typ-M-Path). This ascription is opaque since it returns the elaboration $\tilde{\mathtt{S}}'$ of $\tilde{\mathtt{S}}$ which

Typ-M-Norm
$$\frac{\Gamma \vdash^{\Diamond} \mathtt{M} : \mathtt{S} \qquad \Gamma \vdash \mathtt{S} \downarrow \mathtt{S}'}{\Gamma \vdash^{\Diamond} \mathtt{M} : \mathtt{S}'}$$

Typ-M-Mode
$$\frac{\Gamma \vdash^{\triangledown} \mathtt{M} : \mathtt{S}}{\Gamma \vdash^{\blacktriangledown} \mathtt{M} : \mathtt{S}}$$

Typ-M-Ascr
$$\frac{\Gamma \vdash \tilde{\mathtt{S}} \qquad \Gamma \vdash \tilde{P} : \mathtt{S}'' \qquad \Gamma \vdash \mathtt{S}'' \leqslant \tilde{\mathtt{S}}}{\Gamma \vdash^{\Diamond} (\tilde{P} : \tilde{\mathtt{S}}) : \tilde{\mathtt{S}}}$$

Typ-M-Path
$$\frac{\Gamma \vdash \tilde{P} : (= P' < \mathtt{S})}{\Gamma \vdash^{\Diamond} \tilde{P} : (= \tilde{P} < \mathtt{S})}$$

Typ-M-FctG
$$\frac{\Gamma \vdash^{\blacktriangledown} \mathtt{M} : \mathtt{S}}{\Gamma \vdash^{\Diamond} () \to \mathtt{M} : () \to \mathtt{S}}$$

Typ-M-AppG
$$\frac{\Gamma \vdash P \triangleright () \to \mathtt{S}}{\Gamma \vdash^{\blacktriangledown} P() : \mathtt{S}}$$

Typ-M-FctA
$$\frac{\Gamma \vdash \mathtt{S}_a \qquad \Gamma \uplus Y : \mathtt{S}_a \vdash^{\triangledown} \mathtt{M} : \mathtt{S}}{\Gamma \vdash^{\Diamond} (Y : \mathtt{S}_a) \to \mathtt{M} : (Y : \mathtt{S}'_a) \to \mathtt{S}}$$

Typ-M-Str
$$\frac{\Gamma \vdash_A \overline{\mathtt{B}} : \overline{\mathtt{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end} : \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end}}$$

Typ-M-Proj
$$\frac{\Gamma \vdash^{\Diamond} \mathtt{M} : \langle\gamma\rangle\, \mathtt{S} \qquad \Gamma \uplus \gamma \vdash \mathtt{S} \triangleright \mathtt{sig}_A\ \overline{\mathtt{D}}; \mathtt{module}\ X : \mathtt{S}'; \overline{\mathtt{D}}'\ \mathtt{end}}{\Gamma \vdash^{\Diamond} \mathtt{M}.X : \langle \gamma\, ; A : \overline{\mathtt{D}} \rangle\, \mathtt{S}'}$$

Typ-B-Seq
$$\frac{\Gamma \vdash_A^{\Diamond} \mathtt{B}_0 : \mathtt{D}_0 \qquad \Gamma \uplus A.\mathtt{D}_0 \vdash_A^{\Diamond} \overline{\mathtt{B}} : \overline{\mathtt{D}}}{\Gamma \vdash_A^{\Diamond} A, \mathtt{B}_0, \overline{\mathtt{B}} : \mathtt{D}_0, \overline{\mathtt{D}}}$$

Typ-B-Typ-Bind
$$\frac{\Gamma \vdash \tilde{\mathtt{u}} : \mathtt{u}'}{\Gamma \vdash_A^{\Diamond} (\mathtt{type}\, t = \tilde{\mathtt{u}}) : (\mathtt{type}\, t = \tilde{\mathtt{u}})}$$

Typ-B-Empty
$$\Gamma \vdash_A^{\Diamond} \varnothing : \varnothing$$

Typ-B-AbsType
$$\Gamma \vdash_A^{\Diamond} (\mathtt{type}\, t = A.t) : (\mathtt{type}\, t = A.t)$$

Typ-B-Let
$$\frac{\Gamma \vdash^{\Diamond} \mathtt{e} : \mathtt{u}}{\Gamma \vdash_A^{\Diamond} (\mathtt{let}\, x = \mathtt{e}) : (\mathtt{val}\, x : \mathtt{u})}$$

Typ-B-Mod
$$\frac{\Gamma \vdash^{\Diamond} \mathtt{M} : \mathtt{S}}{\Gamma \vdash_A^{\Diamond} (\mathtt{module}\, X = \mathtt{M}) : (\mathtt{module}\, X : \mathtt{S})}$$

Typ-B-ModType
$$\frac{\Gamma \vdash \tilde{\mathtt{S}} : \mathtt{S}'}{\Gamma \vdash_A^{\Diamond} (\mathtt{module\ type}\ T = \tilde{\mathtt{S}}) : (\mathtt{module\ type}\ T = \mathtt{S}')}$$

Figure 32: Typing rules

is not strengthened by $\tilde{P}$. However, we can also use this construct to implement transparent ascription $(\tilde{P} < \tilde{\mathtt{S}})$ as syntactic sugar for $(P : (= \tilde{P} < \tilde{\mathtt{S}}))$, which then returns the view $\mathtt{S}$ but with the identity of P.

A generative functor Typ-M-FctG is just an evaluation barrier: the functor itself is applicative while the body is generative. Correspondingly, applying a generative functor, which amounts to evaluating its body is then generative. An applicative functor is typed in the obvious way.

Rule Typ-M-Str for structures delays the work to the typing rules for bindings, which carry the self-variable A of the structure as an annotation that should be chosen fresh for the context Γ. The remaining rules are for typing of bindings, which work as expected. In particular, Rule Typ-B-Seq pushes the declaration $A.\mathtt{D}_0$ into the context while typing the $\overline{\mathtt{D}}$, much as Typ-D-Seq for signatures.

Finally, Rule Typ-M-Proj for typing a projection $\mathtt{M}.X$ is the key rule that leverages zippers. Thanks to the use floating components, we can easily deal with the general case when $\mathtt{M}$ is not a path, without running into signature avoidance. Intuitively, it just returns the signature $\mathtt{S}'$ of the field X of the signature $\mathtt{S}$ of $\mathtt{M}$ zipped in the prefix of $\mathtt{S}$ before the field X. Still, we have to consider that the signature of $\mathtt{M}$ may itself be in a zipper γ, which is then composed with the prefix zipper, resulting in $\gamma\,;A : \overline{\mathtt{D}}$. Notice also that we must push γ in the context while resolving the type of $\mathtt{S}$ since $\mathtt{S}$ may not be transparent. Finally, note that since subtyping is not code free, it is not a floating rule and is only allowed at specific places, namely signature ascriptions and functor applications.

4.4 Signature avoidance by zipper simplification

In the type system of the previous section, the only rule that introduces floating components is the projection rule Typ-M-Proj. From there, zippers are transported by the other typing rules. They may be eliminated only by subtyping at a signature ascription or an application. Yet, zippers are a typechecking tool to delay signature avoidance, but should not actually be present at runtime. Indeed, the typing rules prevent direct access to zippers, and internal zippers accesses are only done by type declarations, which are not present at runtime. In this section, extend subtyping to allow *zipper subtyping*. The simultaneously extends dynamic equivalence to allow simplification of floating fields and their removal when they become unreachable, but without loosing any (visible) type equalities. Dynamic equivalence is too expressive and would allow zipper transformations that would be difficult to compute and not very intuitive for the user. We give an effective simplification algorithm that works along a simpler version of dynamic equivalence.

4.4.1 Subtyping with zippers

We first extend ZipML to allow full zippers instead of partial zippers. This extension is done by direct replacement in the previous definitions. **XDR [Should we change the projection rule? We do not need to as we may reify partial zippers as full zippers]**

Code-free equivalence $\approx$, defined in Rule Sub-S-DynEq, Section 4.3.6, could be used as a floating rule since it doesn't change static information nor dynamic representation. However, the underlying code-free subtyping $\lesssim$ has only been defined on zipper-free signatures $\tilde{\mathtt{S}}$. We now extend the definition of subtyping to allow full zippers on both sides. We just add the following subtyping rule for zippers (keeping all the previous rules, including Rule Sub-S-

Zipper):

Sub-S-Zip
$$\frac{\Gamma' = \Gamma \uplus A : \mathtt{D}_1 ; \mathtt{module}\, X : \mathtt{S}_1 ; \overline{\mathtt{D}_1}' \qquad \overline{\mathtt{D}_0} \cdot \overline{\mathtt{D}_0}' \sqsubset_{\leqslant} \overline{\mathtt{D}_1} \cdot \overline{\mathtt{D}_1}' \qquad \Gamma' \vdash \overline{\mathtt{D}_0} \leqslant \overline{\mathtt{D}_2} \qquad \Gamma' \vdash \mathtt{S}_1 \leqslant \mathtt{S}_2 \qquad \Gamma' \vdash \overline{\mathtt{D}_0}' \leqslant \overline{\mathtt{D}_2}'}{\Gamma \vdash \left\langle A : \overline{\mathtt{D}_1} \cdot X \cdot \overline{\mathtt{D}_1}' \right\rangle \mathtt{S}_1 \leqslant \left\langle A : \overline{\mathtt{D}_2} \cdot X \cdot \overline{\mathtt{D}_2}' \right\rangle \mathtt{S}_2}$$

XGR [Check explanation below] The rule has been designed so that it commutes with zipping, i.e., $\Gamma \vdash \langle\gamma\rangle\, \mathtt{S} \leqslant \langle\gamma'\rangle\, \mathtt{S}'$ holds if and only if**XGR [Check that]** $\Gamma \vdash \mathsf{unzip}\ \langle\gamma\rangle\, \mathtt{S} \leqslant \mathsf{unzip}\ \langle\gamma'\rangle\, \mathtt{S}'$. It allows dropping fields, moving fields between sides of the zipper, and subtyping between fields, as long at it preserves well-formedness. Dropping is enabled by the intermediary declarations $\overline{\mathtt{D}_0} ; \overline{\mathtt{D}_0}'$ and the $\sqsubset_{\leqslant}$ relation. By instantiating this rule with $\sqsubset_{\lessapprox}$, the relation $\approx$ applies to all signatures, even with (partial or full) zippers. Since it is defined as an equivalence, no type information (no sharing) has been lost. The interest of moving fields on the right is to prepare for them to be removed—provided the resulting signature is well-formed, that is, provided the to-be-removed fields are never accessed. This may be formalized by introducing a new code-free subtyping relation $\preceq$ called zipper-subtyping where $\sqsubset_{\preceq}$ is defined as:[7]

$$\overline{\mathtt{D}_0} \cdot \overline{\mathtt{D}_0}' \sqsubset_{\preceq} \overline{\mathtt{D}_1} \cdot \overline{\mathtt{D}_1}' \triangleq \overline{\mathtt{D}_0} = \overline{\mathtt{D}_1} \wedge \overline{\mathtt{D}_0}' \subseteq \overline{\mathtt{D}_1}' \qquad\qquad \overline{\mathtt{D}_0} \sqsubset_{\preceq} \overline{\mathtt{D}_1} \triangleq \overline{\mathtt{D}_0} = \overline{\mathtt{D}_1}$$

This allows to drop any subset of fields on the right-hand side of zippers but keep them in order both on the left-hand side of zippers and on signature declarations. This still requires both signatures to be well-formed.

We define a new relation called *simplification* as the relation $(\preceq ; \approx)$ that composes code-free equivalence with zipper-subtyping. This relation is still code-free and commutes with subtyping, that is, $(\precsim ; \leq) \subseteq (\leq ; \precsim)$. This indirectly implies that typechecking will never fail because of $\precsim$-simplifications. **XDR [I am confident that the commutation is correct when the target is in zipper-free signatures, not sure of the more general case]**

Since $\precsim$ is a code-free subtyping relation, we may extend the typing judgment with the following floating (i.e., non syntactic) typing rule:

Typ-M-Simplify
$$\frac{\Gamma \vdash \mathtt{M} : \mathtt{S} \qquad \Gamma \vdash \mathtt{S} \precsim \mathtt{S}'}{\Gamma \vdash \mathtt{M} : \mathtt{S}'}$$

This allows to replace a zipper by an equivalent one with fewer fields as long as we are not loosing any sharing. This can be done anytime, during or after typechecking.

4.4.2 Simplification algorithm

In this section, we propose a simplification algorithm that transforms a signature along $\precsim$, which we can use to reduce the size of the zipper. When we can remove the zipper altogether, this will coincides with solving signature avoidance. When some floating fields remain, this is just delaying signature avoidance.

We first present a simple algorithm that only deals with a single zipper composed of a single abstract type definition. We then extend it several times to deal with more complex zippers: first to treat a single submodule, then a single functor. Finally, we extend it to zippers with any number of components.

This algorithm serves a purpose similar to the *anchoring* of Blaudeau et al. [2024], and we reuse their terminology (anchoring point). However, while their algorithm worked on already-extruded existential types, we present here an algorithm that works directly on the source signatures with zippers, which are technically quite different.

[7]Formally, we should now see $\sqsubset_{\preceq}$ as an overloaded relation defined both between pairs of sequences, to be used in Rule Sub-S-Zip, and between sequences, to be used in Rule Sub-S-Sig.

Preliminary simplifications We may use normalization eagerly to always inline floating module type definitions, floating type definitions, and module type definitions that refer to floating fields. Those could needlessly prevent anchoring. (Other definitions need not—and therefore should not—be inlined as these will not help with further simplifications.)

Anchoring a single type

Anchoring points As a simple starting point, let us consider the avoidance of a single type field. It corresponds to the transformation of $\langle A : \texttt{type}\ t = A.t\rangle\, \mathsf{S}$ reified as $\langle A : \texttt{type}\ t = A.t \cdot \mathbb{Z} \cdot \emptyset\rangle\, \mathsf{S}$ into $\langle A : \emptyset \cdot \mathbb{Z} \cdot \texttt{type}\ t = \mathbb{Z}.P.t'\rangle\, \mathsf{S}'$, where some field of S serves as a new *anchoring point* to avoid $A.t$: all occurrences of $A.t$ are rewritten into $\mathbb{Z}.P.t'$ in S'. An anchoring point for $A.t$ must therefore validate two conditions: (1) it should be of the form $\texttt{type}\, t' = A.t$ and (2) it should be *accessible* from the root of the signature $\mathbb{Z}$. By contrast, all other occurrences of $A.t$ are *usage points*. If a usage point *occurs* before the first anchoring point, then the floating field cannot be moved.

Computing the anchoring points and usage points of $A.t$ can be easily done with a mutable map φ. For now, the map $\varphi(A.t)$ will be either Empty, indicating that no occurrences of $A.t$ have been seen yet, or one of the following *final* states:

- Used if the type has been used in a non anchoring definition;
- $\mathsf{Anchorable}\, P_0.t_0$: if an anchoring point $\texttt{type}\, t_0 = A.t$ has been found at position $\mathbb{Z}.P_0.t_0$ in S.

We then define a $\mathsf{visit}_P\, \mathsf{S}'$ function that recursively goes through the signature S while updating the map. The optional path P indicates the path to the root $\mathbb{Z}$ if still accessible, or none. When visiting without a path, all type declarations are considered usage points, they cannot be anchoring points. The path is reset to none when entering:

- a generative functor or a module type, as no path goes inside;
- an applicative functor, as it would erroneously make the type dependent;[8]
- a submodule with a transparent signature, as paths reaching inside the submodule are normalized away;
- a submodule with a module type signature.

The initial call is $\mathsf{visit}_{\mathbb{Z}}\, \mathsf{S}$. At the end of the visit, the map $\varphi(A.t)$ can be:

- Empty, meaning that the type $A.t$ has not been used;
- Used, in which case no simplification can be made; or
- $\mathsf{Anchorable}\, \mathbb{Z}.P.t'$.

In the latter case, we need to substitute, intuitively, $A.t$ by $\mathbb{Z}.P.t'$.

Contextual path substitution However, due to the structure of submodules in the signature S, we need to revisit the signature to replace all occurrences of $A.t$ found at a path $\mathbb{Z}.P'$ not by path $\mathbb{Z}.P.t'$ itself, but by stripped off its common prefix with $\mathbb{Z}.P'$ and rewired at the enclosing self-reference: For instance, with $\mathsf{Anchorable}\, \mathbb{Z}.X_1.X_2.t'$, occurrences of $A.t$ should be replaced by

[8]For instance, we cannot anchor type $A.t$ inside the body of a functor F as different applications of F would produce different abstract types.

- $A_2.t'$ inside X_2, where A_2 is the self-reference of the signature of X_2;
- $A_1.X_2.t'$ inside X_1, where A_1 is the self reference of the signature of X_1;
- $A.X_1.X_2$ in the rest of the signature $\mathtt{S}$.

Note that rewiring at the closest self-reference is mandatory to ensure well-formedness of the resulting signature. For the rest of this section, we refer to this technique as *contextual path substitution*.

Submodule constraint

We now extend the previous approach to a zipper that contains a single submodule with any number of type declarations or submodules. As an example, let us consider:

$$\langle A : \mathtt{module}\ X : \mathtt{sig}_B\ (\mathtt{type}\ t_1 = B.t_1\ \mathtt{type}\ t_2 = B.t_2)\ \mathtt{end}\rangle\ \mathtt{sig}_C\ \ (\mathtt{type}\ t_3 = A.X.t_1\ \mathtt{val}\ x : A.X.t_2)\ \mathtt{end}$$

We just extend the previous algorithm using a multi-point map φ. At the end of the visiting phase, it would give here:

$$A.X.t_1 \mapsto \mathsf{Anchorable}\, \mathbb{Z}.t_3\ ;\ A.X.t_2 \mapsto \mathsf{Used}$$

However, by contrast with the single-point map, we cannot always move the anchorable fields. Indeed, the whole submodule X cannot be moved to the right of $\mathtt{S}$, if one of its field $X.t_2$ is used but not anchorable (hence in the final state Used). We therefore introduce the *submodule constraint*: fields inside a submodule are anchorable only if no field of the submodule is in Used final state. This can easily be computed on the map after the visit by looking at prefixes.

Module identities

We further extend the algorithm to treat transparent signatures and identity sharing between modules. The overall principle remains the same as for abstract types. The anchoring points for a submodule $\mathtt{module}\, X : \mathtt{S}$ are accessible module declarations of the form $\mathtt{module}\, X' : (= X < \mathtt{S}')$ with the additional constraint that $\mathtt{S}'$ is a subtype of $\mathtt{S}$. Other occurrences of $A.X$ (not as a prefix) are usage points, as in $F(A.X).t$ for instance. We extend the anchoring map φ to support both types and paths. A submodule $A.X$ can eventually be moved only $\varphi(A.X)$ final state is Empty or Anchorable $P.X'$.

After visiting the signature, while respecting the submodule constraint, we do the contextual path substitution. When substituting at the anchoring point, we must get a module declaration $\mathtt{module}\, X' : (= X' < \mathtt{S}')$ that is equal to itself, which we rewrite in $\mathtt{module}\, X' : \mathtt{S}'/X'$; then we continue with the substitution inside $\mathtt{S}$. If X appears as a prefix or inside a functor application, it is also substituted.

Let us consider a small example:

$$\langle A : \mathtt{module}\ X : (\mathtt{sig}_B\ \mathtt{type}\ t_1 = B.t_1\ \mathtt{type}\ t_2 = B.t_2\ \mathtt{end})\rangle$$
$$\mathtt{sig}_C\ \mathtt{type}\ t_3 = A.X.t_1\ \mathtt{module}\ X' : (= A.X < (\mathtt{sig}_B\ \mathtt{type}\ t_1 = B.t_1\ \mathtt{type}\ t_2 = B.t_2\ \mathtt{end}))\ \mathtt{end}$$

After the visit of the signature, we get the following map:

$$A.X.t_1 \mapsto \mathsf{Anchorable}\, \mathbb{Z}.t_3\ ;\ A.X \mapsto \mathsf{Anchorable}\, \mathbb{Z}.X'\ ;\ A.X.t_2 \mapsto \mathsf{Anchorable}\, \mathbb{Z}.X'.t_2\ ;$$

After the substitution, we would**XDR [this is not exactly what we get, right?]** get the signature:

$$\mathtt{sig}_C\ \mathtt{type}\ t_3 = C.t_3\ \mathtt{module}\ X' : (= C.X' < (\mathtt{sig}_B\ \mathtt{type}\ t_1 = C.t_3\ \mathtt{type}\ t_2 = C.X.t_2\ \mathtt{end}))\ \mathtt{end}$$

Actually, reinstalling self-references during the substitution, we finally get:

$$\mathtt{sig}_C\ \mathtt{type}\ t_3 = C.t_3\ \mathtt{module}\ X' : (\mathtt{sig}_B\ \mathtt{type}\ t_1 = C.t_3\ \mathtt{type}\ t_2 = B.t_2\ \mathtt{end})\ \ \mathtt{end}$$

Functor applications

Supporting functor application does not require modifying the map. As done in Blaudeau et al. [2024], we rely on a simple decidable criterion for functors, which is the same as for submodules: functors can be moved only if it has been used and anchored. Whereas we allowed to anchor individual types of submodules, we do not try to anchor individual types out of applicative functors, as they would *suggest* a computation that did not happen. That is why the visit_P continues with an empty path inside applicative functors, which prevent anchorability.

Hence. the only update to the algorithm is to let contextual path substitution go inside functor applications and rewrite them if necessary.

Generalization

Finally, we extend the algorithm to support multiple zippers with multiple fields. A naive generalization would treat every zipper declaration independently, from the innermost to the outermost. This would however be quadratic in the size of the signature, requiring a whole visit of the signature for every zipper field. We can improve on this solution and compute everything in a single pass by considering a third type of point (besides anchoring and usage): *dependency*. A dependency point is a usage point of a zipper field inside the rest of the zipper, not inside the signature. A dependency point becomes a true usage point if the corresponding field turns out to not be anchorable inside the signature.

For instance, consider the following signature:

$$\langle A : \texttt{type}\ t = A.t\ \texttt{module}\ F : (Y : \texttt{sig type}\ u = A.t\ \texttt{end}) \to \ldots\rangle\, \texttt{S}$$

The occurrence of $A.t$ inside the signature of F is a dependency point: if F can be moved after the signature, then the dependency disappears. If not, it becomes a true usage point and prevents $A.t$ from being anchored.

To support dependency points, we extend the anchorable state to $\mathsf{Anchorable}\,(L)\,P$ where L is a list of paths to dependency points, all rooted inside zippers. We then just take the list of dependencies into account after the visit when computing the anchorable fields before applying the contextual substitution.

4.4.3 Restructuring zippers

In fact, while $\precsim$ is sufficient to remove useless zippers, it don't allow changing the structure of zippers. We could actually inverse generalization and allow zippers to be introduced. We may do this by defining the equivalence $\precsim\!\!\!\succsim$ as $(\succsim\, ; \precsim)$ and inject this as an additional typing rule Typ-M-Generalize or, equivalently, replace Typ-M-Simplify by Typ-M-Equiv. This allows to completely reorganize the structure of zippers, provided they bind the same essential fields.

$$\textsc{Typ-M-Generalize}\quad \frac{\Gamma \vdash \texttt{M} : \texttt{S} \qquad \Gamma \vdash \texttt{S} \succsim \texttt{S}'}{\Gamma \vdash \texttt{M} : \texttt{S}'} \qquad\qquad \textsc{Typ-M-Equiv}\quad \frac{\Gamma \vdash \texttt{M} : \texttt{S} \qquad \Gamma \vdash \texttt{S} \precsim\!\!\!\succsim \texttt{S}'}{\Gamma \vdash \texttt{M} : \texttt{S}'}$$

4.5 Properties

4.5.1 Soundness of ZipML by Elaboration in M^ω

We now present the elaboration from ZipML to M^ω Blaudeau et al. [2024]. We expect the reader to be familiar with elaboration of ML modules into F^ω, ideally M^ω Blaudeau et al. [2024] or *F-ing* (Rossberg et al. [2014]) Rossberg et al. [2014]. This elaboration serves as a proof of soundness of the system. Elaborating in M^ω rather than directly in F^ω allows to benefit from the sound extrusion and skolemization of M^ω. The goal is to show that every module expression that typechecks in ZIPML also typechecks in M^ω (as they have the same source language):

$$\vdash^{\Diamond} \texttt{M} : \texttt{S} \quad \Longrightarrow \quad \vdash \texttt{M} : \exists^{\Diamond}\overline{\alpha}.\mathcal{C} \qquad \text{(For some } \overline{\alpha}, \mathcal{C}\text{)}$$

We write $\vdash$ for judgments in M^ω, as opposed to $\vdash$ for judgments in ZIPML. However, to prove this result by induction on the typing derivations we need to extend it to a non-empty typing environment and link the two output signatures. To that aim, we introduce an elaboration of source signatures $\Delta \vdash \texttt{S} : \mathcal{S}$ where Δ and $\mathcal{S}$ are M^ω typing environments and signatures. We may then define $[\![\Gamma]\!]$ by folding the elaboration of signatures over Γ and $[\![\Gamma \vdash \texttt{S}]\!]$ for the signature $\mathcal{S}$ such that $[\![\Gamma]\!] \vdash \texttt{S} : \mathcal{S}$. In standalone M^ω, signatures are of the form $\lambda\overline{\alpha}.\mathcal{C}$, but they are turned into (either transparent or opaque) existential signatures $\exists^{\Diamond}\overline{\alpha}.\mathcal{C}$ when used as the signature of a module. We write $(\lambda\overline{\alpha}.\mathcal{C})^{\Diamond}$ to turn $\lambda\overline{\alpha}.\mathcal{C}$ into $\exists^{\Diamond}\overline{\alpha}.\mathcal{C}$. Then, the desire soundness statement is:

$$\Gamma \vdash^{\Diamond} \texttt{M} : \texttt{S} \quad \Longrightarrow \quad [\![\Gamma]\!] \vdash \texttt{M} : [\![\Gamma \vdash \texttt{S}]\!]^{\Diamond} \qquad \text{(Soundness statement)}$$

In the rest of this section, we first explain two key points: the treatment of zippers and the elaboration of abstract types in the environment. Then, we show how the auxiliary judgments of ZIPML relate in M^ω. **XDR [I guess we will need to cut this]**

Treatment of zippers

The first difficulty in the soundness statement is that the language of inferred signatures of ZIPML is larger than the source signatures of M^ω, with the introduction of zippers in signatures and zipper accesses in paths. Intuitively, zippers are removed at runtime, and could be removed in M^ω. However, to establish a one-to-one correspondence on inferred signatures, we need to keep zippers in both inferred signatures and the environment. Therefore, we extend M^ω signatures and environments with zippers. We add zippers to M^ω signatures with the following signature elaboration rule, along with appropriate path typing extension.

$$\frac{\Gamma \vdash_A \overline{\texttt{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}} \qquad \Gamma, A : \overline{\mathcal{D}} \vdash \texttt{S} : \lambda\overline{\beta}.\mathcal{C}}{\Gamma \vdash \langle A : \overline{\texttt{D}}\rangle \texttt{S} : \lambda\overline{\alpha},\overline{\beta}.\langle A : \overline{\mathcal{D}}\rangle \mathcal{C}}\ \textsc{Typ-S-Zip}$$

However, and this is a key point, those zippers are only used during the proof by induction for typing the inferred signatures of ZIPML. They should be seen as *decorations* on types and contexts to build a typing derivation in M^ω, after which zippers can be erased. If we were to produce the erased typing derivation of M^ω directly, it would be difficult to capture the right induction invariants.

Elaboration of environments and strengthening

When elaborating environments $[\![\Gamma]\!]$ of ZIPML, another technical point arises from the representation of abstract types as self-referring signatures in ZIPML typing environments, since M^ω uses existential type variables instead. For instance, consider an environment extended with a submodule: $\Gamma; \texttt{module}\, A.X : \texttt{S}$. All type declarations inside S are concrete, either referring to other modules or to $A.X$ itself. To help establish the correspondence between the two representations, we define the interpretation of environments *by the property* that the signature refers to itself:

$$\frac{[\![\Gamma]\!]; \overline{\alpha}; \texttt{module}\, A.X : \mathcal{C} \vdash \texttt{S} : \mathcal{C}}{[\![\Gamma; \texttt{module}\, A.X : \texttt{S}]\!] = [\![\Gamma]\!]; \overline{\alpha}; \texttt{module}\, A.X : \mathcal{C}}$$

XGR [This whole paragraph is not very understandable] This is not an algorithmic rule, as it requires to *guess* the signature $\mathcal{C}$ and the set of abstract types $\overline{\alpha}$ to show that some environment is indeed the result of the interpretation. In the statement, when we write $[\![\Gamma]\!]$ we implicitly mean that there exists Γ' such that $\Gamma' = [\![\Gamma]\!]$. However, during the proof we can exhibit such a Γ', as objects are added in the environment via the operator $\uplus$ defined in Section 4.3.3. From $(\Gamma \uplus \texttt{module}\, A.X : \texttt{S})$, we can "catch" the signature $\texttt{S}$ before its strengthening and use it to find $\mathcal{C}$. Then, the elaboration of $\texttt{S}$ is $\Gamma \vdash \texttt{S} : \lambda\overline{\alpha}.\mathcal{C}$. From there, we identify the set of abstract types $\overline{\alpha}$ and the signature $\mathcal{C}$. This is justified by the fact that elaboration of a strengthened signature $\texttt{S} \,/\!/\, P$ is the same as elaboration of the signature $\texttt{S}$ before strengthening is applied to some type variables:

Lemma 9 (Elaboration of strengthening)**.** *Given a path P and a signature $\texttt{S}$, such that $[\![\Gamma]\!] \vdash P : \mathcal{C}'$ and $[\![\Gamma]\!] \vdash \texttt{S} : \lambda\overline{\alpha}.\mathcal{C}$ and $[\![\Gamma]\!] \vdash \mathcal{C}' < \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]$, we have $[\![\Gamma]\!] \vdash \texttt{S} \,/\!/\, P : (\lambda\overline{\alpha}.\mathcal{C})\,\overline{\tau}$.*

Elaboration of judgments

As normalization, resolution, path typing, and subtyping are mutually recursive, we state four combined properties that are proven by mutual induction. Type and module type definitions are kept as such and only inlined on demand in ZipML, while they are immediately inlined in M^ω. Therefore, ZipML normalization becomes the identity in M^ω.

Lemma 10 (Elaboration of normalization)**.** *If S normalizes to $\texttt{S}'$ in ZipML ($\Gamma \vdash \texttt{S} \downarrow \texttt{S}'$), then both signatures have the same elaboration in M^ω. Namely, $[\![\Gamma]\!] \vdash \texttt{S} : \lambda\overline{\alpha}.\mathcal{C}$ implies $[\![\Gamma]\!] \vdash \texttt{S}' : \lambda\overline{\alpha}.\mathcal{C}$.*

Technical detail The introduction of module identities is done by a separate source-to-source transformation in M^ω, which, as the authors suggest, could be merged with typing. This is what we do. For the sake of readability, we use pairs to represent their (id, Val) structures:

$$(\tau, \mathcal{C}) \triangleq \mathsf{sig\ type}\ id = \tau; \mathsf{module}\ Val : \mathcal{C}\ \mathsf{end}$$

Path typing and resolution in ZipML can be understood as accessing either the full module, the identity type, or the value of a module at path P in M^ω:

Lemma 11 (Elaboration of path-typing and resolution)**.** *If $\Gamma \vdash P : \texttt{S}$ then we have both $[\![\Gamma]\!] \vdash P : (\tau, \mathcal{C})$ and $[\![\Gamma]\!] \vdash \texttt{S} : (\tau, \mathcal{C})$. Moreover,*

- *If $\Gamma \vdash P \rhd \texttt{S}$ then P and S have the same signature in M^ω: $[\![\Gamma]\!] \vdash P : (\tau, \mathcal{C})$ and $[\![\Gamma]\!] \vdash \texttt{S} : (\tau, \mathcal{C})$.*
- *If $\Gamma \vdash P \rhd P'$ then P and P' have the same identity in M^ω: $[\![\Gamma]\!] \vdash P : (\tau, \_\ )$ and $[\![\Gamma]\!] \vdash P' : (\tau, \_\ )$.*

Lemma 12 (Elaboration of subtyping)**.** *The elaboration preserves subtyping relationship: If $[\![\Gamma]\!] \vdash \texttt{S} : \lambda\overline{\alpha}.\mathcal{C}$ and $[\![\Gamma]\!] \vdash \texttt{S}' : \lambda\overline{\alpha}'.\mathcal{C}'$ then $\Gamma \vdash \texttt{S} \leqslant \texttt{S}'$ implies $[\![\Gamma]\!] \vdash \lambda\overline{\alpha}.\mathcal{C} < \lambda\overline{\alpha}.\mathcal{C}'$.*

XDR [I would prefer to formulate it as announced]

Theorem 13: Soundness

If $\Gamma \vdash^{\diamondsuit} \texttt{M} : \texttt{S}$ then $\exists \mathcal{S}$ such that $[\![\Gamma]\!] \vdash \texttt{S} : \mathcal{S}$ and $[\![\Gamma]\!] \vdash \texttt{M} : \mathcal{S}^{\diamondsuit}$.

Erasing of zippers In the soundness theorem above, the decorated environment is only used for the elaboration of the inferred signature. The left-hand side part is just normal typing in M^ω: no zipper signature and no zipper access inside $\texttt{M}$. Therefore, we can erase the decoration in the context and the resulting signature, as it was only useful for the proof, to obtain a plain M^ω typing derivation.

4.5.2 Completeness of ZipML with respect to M^ω

Conversely, one might wonder if the type system of ZipML is powerful enough to simulate M^ω. We conjecture that it is the case. In this section we present the key arguments supporting it. The main mechanism present in M^ω that differs from ZipML is the introduction and extrusion of existentially quantified types out of submodules and applicative functors via skolemization. However, we show that, using the generalized equivalence defined in Section 4.4.3, we can simulate the extrusion mechanism with floating components. Contrarily to quantified variables, those components are not alpha-convertible, but we show that it does not matter as they can be freely renamed at any point.

Simulating extrusion of abstract types with floating components

Floating fields can be used to encode extrusion of existential types. For instance, if we consider a type component inside a submodule, we can introduce a full zipper with a new type component on the right-hand side, and use equivalence to commute it. This would give:

$$\begin{array}{l}
\mathtt{sig}\ \mathtt{module}\ X : (\mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end})\ \mathtt{end} \\
\quad \stackrel{\star}{\approx}\ \langle B : \emptyset \cdot \mathbb{Z} \cdot \mathtt{type}\ t_0 = \mathbb{Z}.X.t \rangle\ \mathtt{sig}\ \mathtt{module}\ X : (\mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end})\ \mathtt{end} \\
\quad \stackrel{\star}{\approx}\ \langle B : \mathtt{type}\ t_0 = B.t_0 \cdot \mathbb{Z} \cdot \emptyset \rangle\ \mathtt{sig}\ \mathtt{module}\ X : (\mathtt{sig}_A\ \mathtt{type}\ t = B.t_0\ \mathtt{end})\ \mathtt{end}
\end{array}$$

Similarly, floating fields can be used to encode extrusion of higher-order types, by introducing unary functors that take an empty argument as input:

$$\begin{array}{l}
(Y : \mathtt{S}) \to \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{val}\ x : A.t\ \mathtt{end} \\
\quad \stackrel{\star}{\approx}\ \langle B : \emptyset \cdot \mathbb{Z} \cdot \mathtt{module}\ F_0 : (Y : \mathtt{sig}\ \ \mathtt{end}) \to \mathtt{sig}_A\ \mathtt{type}\ t = \mathbb{Z}(Y).t\ \mathtt{end} \rangle \\
\qquad\qquad (Y : \mathtt{S}) \to \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{val}\ x : A.t\ \mathtt{end} \\
\quad \stackrel{\star}{\approx}\ \langle B : \mathtt{module}\ F_0 : (Y : \mathtt{sig}\ \ \mathtt{end}) \to \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end} \cdot \mathbb{Z} \cdot \emptyset \rangle \\
\qquad\qquad (Y : \mathtt{S}) \to \mathtt{sig}_A\ \mathtt{type}\ t = B.F_0(Y).t\ \mathtt{val}\ x : A.t\ \mathtt{end}
\end{array}$$

This shows that floating fields have at least the same expressiveness as existentially quantified signatures with skolemization, except that they introduce names.

Alpha-conversion of floating components

Using again the generalized equivalence of Section 4.4.3, we can always rename a floating component by introducing a new component on the right-hand side that copies it and then permute them. For instance, we have:

$$\begin{array}{lcl}
\langle A : \mathtt{type}\ t = \mathtt{u} \rangle\ \mathtt{S} & \stackrel{\star}{\approx} & \langle A : \mathtt{type}\ t = \mathtt{u} \cdot \mathbb{Z} \cdot \mathtt{type}\ t' = A.t \rangle\ \mathtt{S} \\
& \stackrel{\star}{\approxeq} & \langle A : \mathtt{type}\ t' = \mathtt{u} \cdot \mathbb{Z} \cdot \mathtt{type}\ t = A.t' \rangle\ \mathtt{S}[t \mapsto t'] \\
& \stackrel{\star}{\approx} & \langle A : \mathtt{type}\ t' = \mathtt{u} \rangle\ \mathtt{S}[t \mapsto t']
\end{array}$$

Combining these two facts, we see that floating components can play the role of extruded existential types of M^ω: a canonical signature $\exists^{\blacktriangledown}\alpha.\mathcal{C}$ can be encoded as a zipper $\langle A : \mathtt{type}\ t_\alpha = A.t_\alpha \rangle\ \mathtt{S}$ where the source signature $\mathtt{S}$ is obtained by substituting α for $A.t_\alpha$. The other quantifications (universal and lambda) are always used for signatures that come from elaboration of the source, and can therefore also be represented in ZipML. We leave the technical details for future works.

4.5.3 Other properties

Normalization is a floating typing rule that can be called anytime. Normalization itself may be performed by need, but also in a strict manner. It is therefore left to the implementation to normalize just as necessary—as one would typically do with β-reduction.

As a result, the inferred signature is not unique, returning different syntactic answers according to the amount of normalization that has been done. Hence, we may have $\Gamma \vdash \mathtt{M} : \mathtt{S}$ and $\Gamma \vdash \mathtt{M} : \mathtt{S}'$ when $\mathtt{S}$ and $\mathtt{S}'$ syntactically differ—even a lot! as one may contain a signature definition expanded in the other.

Still, we should then have $\Gamma \vdash \mathtt{S}' \approx \mathtt{S}''$. That is, the inferred signatures should only differ up to their presentation, but remain inter-convertible—and otherwise simplified in the same manner.

One might expect a stronger result, stating that there is a best presentation where module names would have been expanded as little as possible. This would be worth formalizing, although a bit delicate. In particular, we probably wish to keep names introduced by the user, but not let the algorithm reintroduce a name when it recognized an inferred signature that was not named, but just happens to be equivalent to one with a name.

4.6 Missing features and Conclusion

We have presented ZipML, a source system for ML modules which uses a new feature, signature zippers, to solve the avoidance problem. ZipML also models transparent ascription, delayed strengthening, applicative and generative functors and parsimonious inlining of signatures.

Several essential features are still missing to bring ZipML on par with all features included in OCaml.

The **open** and **include** constructs allow users to access or inline a given module. While not problematic when used on paths, OCaml also allows opening structures Li and Yallop [2017], which easily triggers signature avoidance, as we have shown in **??** on a restricted case. We expect floating fields to easily model the opening of structures although some adjustments will be needed. In particular, type checking of signatures will have to become an elaboration judgment as mentioned in Section 4.3.7.

OCaml allows *abstract signatures* which amounts to quantify over signatures in functors. This feature, while rarely used in practice, unfortunately makes the system undecidable Rossberg [1999]; White [2015]. In our context, as ZipML expands module type names only by need. We conjecture that abstract signatures could be added to the system, as they should not impact zippers. However, the undecidability of subtyping should be addressed, maybe by restricting their instantiation.

Finally, the typechecking of recursive modules raises the question of *double vision* Dreyer [2007a]; Nakata and Garrigue [2006]. By contrast, OCaml requires full type annotations, along with an initialization semantics which can fail at runtime. All these solutions are compatible with ZipML. Another potential proposal would be to rely on Mixin modules Rossberg and Dreyer [2013], which could fit well with floating fields.

We leave these explorations for future works. An implementation of floating components into OCaml, as well as transparent ascription, should not be difficult, now that we have a detailed formalization that also fits well with the actual OCaml implementation. This remains to be done to appreciate the gain in expressiveness and verify that we do not loose in typechecking speed. A mechanization of ZipML metatheory for which we only have paper-sketched proofs would also be worth doing and would fit well with other efforts towards a mechanized specification of OCaml and formal proofs of OCaml programs.

4.7 Fixes

Currently, the typing rules are still odd.

4.7.1 Do we have those Properties?

- If $\Gamma \vdash P : \mathtt{S}$ and $\Gamma \vdash \mathtt{S} \approx \mathtt{S}'$, does $\Gamma \vdash P : \mathtt{S}'$ hold?

  It does not seem so. But we have $\Gamma \vdash^{\diamond} P : \mathtt{S}'$.

- $\Gamma \vdash \mathtt{S} \downarrow \mathtt{S}$

  No we can only normalize transparent signatures, module types. possibly under zippers.

- The following desirable property does not hold:

  If $\Gamma \vdash \mathtt{S} \downarrow \mathtt{S}'$ then $\Gamma \vdash \mathtt{S} \approx \mathtt{S}'$

  That is, normalization is not in the kernel of subtyping.

4.7.2 Desired properties

Rule Sub-S-Norm suggests that we could normalize on either side, and thus that normalization is reflexive, one side doing nothing, but normalization is not reflexive! We should make it reflexive—and transitive.

Or at least, Sub-S-Norm should be split into two rules

$$\textsc{Sub-S-Norm}\quad \frac{\Gamma \vdash \mathtt{S}_1 \downarrow \mathtt{S}_1' \qquad \Gamma \vdash \mathtt{S}_1' \leqslant \mathtt{S}_2}{\Gamma \vdash \mathtt{S}_1 \leqslant \mathtt{S}_2} \qquad\qquad \textsc{Sub-S-Norm}\quad \frac{\Gamma \vdash \mathtt{S}_2 \downarrow \mathtt{S}_2' \qquad \Gamma \vdash \mathtt{S}_1 \leqslant \mathtt{S}_2'}{\Gamma \vdash \mathtt{S}_1 \leqslant \mathtt{S}_2}$$

If we do this, then we have:

$$\textsc{Prop}\quad \frac{\Gamma \vdash \mathtt{S} \downarrow \mathtt{S}'}{\Gamma \vdash \mathtt{S} \approx \mathtt{S}'}$$

That is normalization would be in the kernel of subtyping.

More generally, I think we should allow replacing P by P' whenever $\Gamma \vdash P \triangleright P'$, including by congruence. This should preserve well-typedness, identities, and signatures. This would also allow to define a normal form for paths.

4.7.3 Structural components inside floating ones

We may inline module type definitions of floating components. But we may not be able to remove them from floating components, as they may appear in signature of functors argument not only as a signature which could be inlined, but as a module type component.

4.8 Simplification algorithm

The simplification is presented incrementally but informally in Section 4.4.2. Here we, describe the simplification of one zipper $\langle\gamma\rangle\,\mathtt{S}$ where γ is $(A_i : \overline{\mathtt{D}}_i)I_1$. We asume $\mathbb{Z} : \langle\gamma\rangle\,\mathtt{S}$, using $\mathbb{Z}$ as a path to refer to both floating fields $\mathbb{Z}.A_i$ and structural fields of $\mathtt{S}$ (if any).

The simplification will perform a structual (top-down, left–to-right) visit of $\langle\gamma\rangle\,\mathtt{S}$ and will track floating fields $\mathbb{Z}.A_i$ and their submodules recursively, i.e., all paths of the form $\mathbb{Z}.A_i.\overline{X}$, since only those fields may be an

4.9 Global Simplification algorithm

XDR [I think I am giving up the global simplification]

The algorithm we presented in Section 4.4.2 simplifies one zipped signature at a time. Thus to fully simplify a signature we would need to call it recursively on all zipped signatures, inner signature first, which would the algorithm quadratic. The algorithm can be adapted to perform the simplification of inner zippers altogether. The presentation is however more involved.

We still asume that a pre-simplification pass reduces zippers to contain only essential floating declarations. The algorithm takes as input a signature $\mathtt{S}$ that is well-typed in a context Γ and output a signature $\mathtt{S}'$ such that $\Gamma \vdash \mathtt{S} \precsim \mathtt{S}'$. It first performs a top-down left-to-right visit of $\mathtt{S}$ to build a graph of dependencies between nodes of $\mathtt{S}$ (e.g.,. all declaration fields adn subsignatures of $\mathtt{S}$). We then use nodes as a absolute reference to the declaration or signature they represent.

In a second pass floating-fields are visited in reverse-order of dependences, and removed when they were unaccessed before being subsumed by structural nodes, together with their dependencies, thus possibly eanbling further simplifications. The second pass performs a sort of garbage collection, but with more complex dependencies than in an simple GC algorithm.

Nodes are partially ordered by the subterm relation $\multimap$ within $\mathtt{S}$. The initial post-order traversal of $\mathtt{S}$ defines a total ordering $\prec$ on nodes that is actually compatible with (i.e., a super relation of) $\multimap$. During the visit, we build the additional edges.

4.9.1 Path equivalence

One difficulty to understand simplificaton is to understand and restrict equivalence. The purpose of simplification is to find anchoring points for essential virtual type of structural module fields. Namely, fields $(\mathtt{type}\, t = \mathtt{u})$ and $(\mathtt{module}\, X : \mathtt{S})$ such that $P.t$ or $(= P.X < \mathtt{S}')$ is later used when P refers to this particular field. A type can only be an anchor for a virtual field if it is equivalent to that field. This is a prerequisite. However, this is not sufficient. To make anchoring simpler, we restrict the equivalence of fields.

For instance, one restriction is to only replace an application $P_1(P_2)$ by another application $P_1'(P_2')$ when P_1' and P_2' are equivakent to P_1 and P_2, respectively. This is sound but incomplete.

Consider for instance,

```
module type S = sig ... end
module M : S = struct ... end
module X = struct
  module F₁ (Y : S) = struct module X = struct type t end end
  module F₂ (Y : S) = F₁(Y).X
end
```

```
module type S = sig ... end
module M : S
module X : sig
  module F₁ (Y : S) : sig module X : sig type t end end
  module F₂ (Y : S) : sig type t = F₁(Y).t end
end
```

Calling this environment Γ, we have $\Gamma \vdash X.F_1(M).X.t \approx X.F_2(M).t$. both types are actually equivalent to $(X.F_1)(M).t$ in Γ. Clearly, however, $\Gamma \vdash X.F_1 : X.F_2$ does not hold.

Our solution is to use a more restrictive (and syntactic) definition of *equivalence for anchorability* with the property:

$$\frac{\Gamma \vdash P_1(P_2).P \mathrel{<\!\!\approx} Q}{\begin{array}{c} Q = P_1'(P_2').P' \qquad \Gamma \vdash P_1 : (Y : \mathtt{S}_a) \to \mathtt{S} \qquad \Gamma \vdash P_1' : (Y : \mathtt{S}_a) \to \mathtt{S} \qquad \Gamma \vdash P_1 \mathrel{<\!\!\approx} P_1' \\ \Gamma \vdash P_2 \mathrel{<\!\!\approx} P_2' \qquad \Gamma, \mathbb{Z} : \mathtt{S} \vdash \mathbb{Z}.P \mathrel{<\!\!\approx} \mathbb{Z}.P' \end{array}}$$

(It is possible to exhibit a inductive definition that has this property.)

With such a restriction, it may be simpler to track anchorable positions, since as sonne as there is an application, we split anchorability into smaller problems.

One question is whether the above definition is satisfactory (sufficient): the example above would be rejected, although the two paths are obviously equivalent; hence, it may perhaps this not be good enough.

With this restriction, it is easier to see which definition could anchor another one. We may somewhat compute normal form for paths and compare them. Without such a restriction, anchorability of $P.X$ or $P.t$ when P contains an application is no so obvious. Still, in the case of the previous example, it is obvious that $X.F_2(M).t$ could be an anchoring point for $X.F_1(M).X.t$—if there is not other use of F_1.

4.9.2 Computing dependencies

We describe the simplification of a node zipped signature $\langle (A_i : \overline{\mathtt{D}}_i) I_1 \rangle\, \mathtt{S}$.

XDR [We temporarily consider a simpler form $\langle A_1 : \overline{\mathtt{D}_1} \rangle\, \mathtt{S}_1$—a multiziper will just make notations a bit heavier.]

The algorithm proceeds by visiting nodes in $\prec$-order, transporting a typing environment Γ during the visit as in the well-formnedness judgment $\Gamma \vdash \langle A : \overline{\mathtt{D}} \rangle\, \mathtt{S}$. However, we need to name fields that will be visited in both $\mathtt{D}$ and $\mathtt{S}$. For that purpose, we instead consider the context equal to $\Gamma \uplus A : \overline{\mathtt{D}} \uplus \mathbb{Z} : \mathtt{S}$.

We may then address *floadting* nodes $A.P$ and *structural* $\mathbb{Z}.P$ where P is of the form $\overline{X}$ or $\overline{X}.t$. We use *accessible* node to designate a node that is either floating or structural. We build a graph φ between accessible nodes.

The visit builds dependency *direct* and *reverse* edges. Reverse edges represent anchoring options that would redirect a floating field to a structural field, under the condition that the floating field could be removed. During the visit, we may also *lock* floating nodes, fields that we know cannot be anchored since they are used before they could be anchored.

Path typing $\Gamma \vdash P : \mathtt{S}$ is not deterministic as it allows, without enforcing, normalization. We assume that there is a normalization operation $\lfloor \Gamma \vdash P \rfloor$ and $\lfloor \Gamma \vdash P.t \rfloor$ that returns the normal form P' of path P or $\mathtt{u}$ of type $P.t$ in context Γ.

> Normalization proceeds by following type and module type definitions. Both processes are acyclic—as long as we recognize abstract type definition and module identity definitions as self aliases and thus terminate.

The normal type $\lfloor \Gamma \vdash P.t \rfloor$ is a type $\mathtt{u}$ whose paths have all been normalized. Therefore, remaining paths $P'.t'$ are all abstract type definitions. Two abstract type definitions are the same if and only if their normal forms are equal. Similarly, two paths have the same identity if and only if their normal forms are equal. **XDR [Needs more precision]**

4.9.3 The visit

XDR [The rests of this section is old material not up-to-date]

General Schema

Intuitively, and as a guideline, this amounts to simplifying the unzipped signature $\mathtt{sig}_A$ `D` `module` $\mathbb{Z}$: `S` `end` and projecting it back. However, we proceed directly on the zipped signature in two steps.

In a first step we collect information about the use of floating fields, and the possibility of re-anchoring some abstract types of the zipper in `S` rather so that they can be moved after `S` and dropped. This perform a left-to-right top-down visit of the floating declarations `D`, then of the signature .

When visiting `D`, we collect information in a map φ from nodes of `D` addressed by their paths from A to some state that is adjusted imperatively. When visiting `S` we continue adjusting address the nodes of `S` using $\mathbb{Z}$ to refer to the toplevel signature as if we had $\mathbb{Z}$: `S` and we continue adjusting the mapping φ.

The state $\varphi(P)$ (where P is of the form $A.P$) may be:

- $\mathsf{Used}\,L$ where L is the list of nodes in `D` that depends on $A.P$. The state $\mathsf{Used}\,\varnothing$, is the initial state of the node when it is first visited and made visible. This state can be set during and only during the visit of `D`.

- Frozen: means that P is not anchorable. It is set during the visit of `S` if the field is used in `S` before being anchorable.

- $\mathsf{Anchorable}\,(L)\,P'$: means that P is anchorable at path P', which must be of the form $\mathbb{Z}$, and depends on nodes in L. This is set during the visit of `S` if P' in an anchoring position and the field was not frozen.

- $\mathsf{Anchored}\,P$: this will only be set after the visit of `S`.

We write $\prec$ the left-to-right depth-first ordering of `D`. A node P' may only appear in $\varphi(P)$ if $P \prec P'$.

Finalizing

Finally, we use collected information to transform the $\langle A : \overline{\mathtt{D}}\rangle\,\mathtt{S}$ into $\langle A : \mathtt{D}_1 \cdot \mathbb{Z} \cdot \mathtt{D}_2\rangle\,\mathtt{S}'$. First, we should split `D` into $\mathtt{D}_1$ and $\mathtt{D}_2$. We visit nodes in the $\prec$-order to adjust the states: **XDR [Need to be adjusted]**

- $\mathsf{Anchorable}\,(L)\,P$: if a node of L is frozen, make it frozen, and recursively make its root (the node immediately under A frozen). Otherwise set it to $\mathsf{Anchored}\,P$

- $\mathsf{Used}\,L$: if a node of L is frozen, freeze P. Otherwise reset it to $\mathsf{Used}\,\varnothing$.

Then, frozen fields are moved to $\mathtt{D}_1$ and all other fields are moved to $\mathtt{D}_2$.

The final step is to rewrite `S` into $\mathtt{S}'$ (and in principle $\mathtt{D}_2$ into $\mathtt{D}_2'$ to take into account the redirections after making some nodes of `S` anchoring points). In fact, fields of $\mathtt{D}_2$ will be dropped and so need not be adjusted. **XDR [Normalizing $\mathtt{D}_2$ does not seem necessary]**

To do so, we collect the list of nodes at P marked $\mathsf{Anchored}\,P_0$. We build a redirection map $P \mapsto P'$ which applies the substitutions in a final visit of `S`.

The effect is intuitively simple but technically somewhat tricky as we need to not only substituted the definitions of anchoring nodes (easy), but also to redirect the older paths that pointing to the floating components to point back to their “new anchor” and we must do so using self references, as even though we used $\mathbb{Z}$ to refer to nodes of `S`, the signature `S` is not necessarily transparent.

The key is to reinstall self-references as far as possible. When at location P_0 we must replace an old path P_1 by a new path P_2, we should actually replace P_1 by $P_2 \setminus P_0$ defined to find the "shortest" redirection via the use of closest self-references in common scope.

XDR [We need to describe the substitution more precisely]

Finally, we have described the simplification of zipped signature $\left\langle A : \overline{\mathtt{D}} \right\rangle \mathtt{S}$. We may simplify arbitrary zippers $\langle \gamma \rangle\, \mathtt{S}$ by iterating the simplification of a simple zipper. Then, we may simplify an arbitrary signature $\mathtt{S}$ by recursively simplifying its zippers, inner ones first. The algorithm is not efficient as it will visit the signature several times. We believe the simplification of all zippers could be done altogether without iteration.

Chapter 5

Advanced features

5.1 Hello

5.2 World

Conclusion

Bibliography

Sandip K. Biswas. 1995. Higher-Order Functors with Transparent Signatures. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 154–163. https://doi.org/10.1145/199448.199478

Clément Blaudeau, Didier Rémy, and Gabriel Radanne. 2024. Fulfilling OCaml Modules with Transparency. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 101 (apr 2024), 29 pages. https://doi.org/10.1145/3649818

Karl Crary. 2020. A focused solution to the avoidance problem. *Journal of Functional Programming* 30 (2020), e24. https://doi.org/10.1017/S0956796820000222

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a recursive module? *SIGPLAN Not.* 34, 5 (may 1999), 50–63. https://doi.org/10.1145/301631.301641

Derek Dreyer. 2007a. Recursive type generativity. *Journal of Functional Programming* 17, 4-5 (2007), 433–471. https://doi.org/10.1017/S0956796807006429

Derek Dreyer. 2007b. A type system for recursive modules. *SIGPLAN Not.* 42, 9 (oct 2007), 289–302. https://doi.org/10.1145/1291220.1291196

Derek Dreyer, Karl Crary, and Robert Harper. 2003. A type system for higher-order modules. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, Alex Aiken and Greg Morrisett (Eds.). ACM, 236–249. https://doi.org/10.1145/604131.604151

Derek Dreyer, Robert Harper, and Karl Crary. 2005. *Understanding and evolving the ML module system*. Ph. D. Dissertation. USA. AAI3166274.

Jacques Garrigue and Didier Rémy. 2012. Tracing ambiguity in GADT type inference. (June 2012).

Robert Harper and Mark Lillibridge. 1994. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 123–137. https://doi.org/10.1145/174675.176927

Robert Harper, John C. Mitchell, and Eugenio Moggi. 1989. Higher-Order Modules and the Phase Distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '90)*. Association for Computing Machinery, New York, NY, USA, 341–354. https://doi.org/10.1145/96709.96744

Robert Harper and Christopher A. Stone. 2000. A type-theoretic interpretation of standard ML. In *Proof, Language, and Interaction*. https://api.semanticscholar.org/CorpusID:9208816

Hyeonseung Im, Keiko Nakata, Jacques Garrigue, and Sungwoo Park. 2011. A syntactic type system for recursive modules. *SIGPLAN Not.* 46, 10 (oct 2011), 993–1012. https://doi.org/10.1145/2076021.2048141

Pauli Jaakkola. 2020. *A Type System for First-Class Recursive ML Modules.* Master's thesis. https://trepo.tuni.fi/bitstream/handle/10024/123958/JaakkolaPauli.pdf

Xavier Leroy. 1994. Manifest Types, Modules, and Separate Compilation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 109–122. https://doi.org/10.1145/174675.176926

Xavier Leroy. 1995. Applicative functors and fully transparent higher-order modules. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95*. ACM Press, San Francisco, California, United States, 142–153. https://doi.org/10.1145/199448.199476

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

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, Kc Sivaramakrishnan, and Jérôme Vouillon. 2023. *The OCaml system release 5.1: Documentation and user's manual.* Intern report. Inria. https://inria.hal.science/hal-00930213

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2024. The OCaml system. https://ocaml.org/manual/index.html

Runhang Li and Jeremy Yallop. 2017. Extending OCaml's 'open'. In *Proceedings ML Family / OCaml Users and Developers workshops, ML/OCaml 2017, Oxford, UK, 7th September 2017 (EPTCS, Vol. 294)*, Sam Lindley and Gabriel Scherer (Eds.). 1–14. https://doi.org/10.4204/EPTCS.294.1

Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. *Proc. ACM Program. Lang.* 8, ICFP, Article 253 (aug 2024), 30 pages. https://doi.org/10.1145/3674642

David MacQueen, Robert Harper, and John Reppy. 2020. The history of Standard ML. *Proc. ACM Program. Lang.* 4, HOPL, Article 86 (jun 2020), 100 pages. https://doi.org/10.1145/3386336

Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA.

John C. Mitchell. 1988. Polymorphic type inference and containment. *Information and Computation* 76, 2 (1988), 211–249. https://doi.org/10.1016/0890-5401(88)90009-0

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

Benoît Montagu and Didier Rémy. 2009. Modeling Abstract Types in Modules with Open Existential Types. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL'09)*. Savannah, GA, USA, 354–365. https://doi.org/10.1145/1480881.1480926

Keiko Nakata and Jacques Garrigue. 2006. Recursive Modules for Programming. (2006), 13.

Benjamin C. Pierce. 2004. *Advanced Topics in Types and Programming Languages.* The MIT Press.

Andreas Rossberg. 1999. Undecidability of OCaml type checking. https://sympa.inria.fr/sympa/arc/caml-list/1999-07/msg00027.html.

Andreas Rossberg. 2006. The missing link: dynamic components for ML. *SIGPLAN Not.* 41, 9 (sep 2006), 99–110. https://doi.org/10.1145/1160074.1159816

Andreas Rossberg and Derek Dreyer. 2013. Mixin'Up the ML Module System. *ACM Trans. Program. Lang. Syst.* 35, 1 (April 2013), 2:1–2:84. https://doi.org/10.1145/2450136.2450137

Andreas Rossberg, Claudio Russo, and Derek Dreyer. 2014. F-ing modules. *Journal of Functional Programming* 24, 5 (Sept. 2014), 529–607. https://doi.org/10.1017/S0956796814000264

Claudio V. Russo. 2000. First-Class Structures for Standard ML. *Nord. J. Comput.* 7, 4 (2000), 348–374.

Claudio V. Russo. 2001. Recursive structures for standard ML. *SIGPLAN Not.* 36, 10 (oct 2001), 50–61. https://doi.org/10.1145/507669.507644

Claudio V. Russo. 2004. Types for Modules. *Electronic Notes in Theoretical Computer Science* 60 (2004), 3–421. https://doi.org/10.1016/S1571-0661(05)82621-0

Didier Rémy and Jérôme Vouillon. 1997. Objective ML: A simple object-oriented extension of ML. In *Proceedings of the 24th ACM Conference on Principles of Programming Languages*. Paris, France, 40–53.

Chung-Chieh Shan. 2004. Higher-order modules in System F^ω and Haskell. (01 2004).

Zhong Shao. 1999. Transparent Modules with Fully Syntactic Signatures. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999*. ACM, 220–232. https://doi.org/10.1145/317636.317801

Leo White. 2015. Girard Paradox implemented in OCaml using abstract signatures.