Avoiding signature avoidance in ML modules with zippers

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

We present ZipML, a new path-based type system for a fully fledged ML-module language that avoids the signature avoidance problem. This is achieved by introducing *floating fields*, which act as additional fields of a signature, invisible to the user but still accessible to the typechecker. In practice, they are handled as *zippers* on module signatures, and can be seen as a lightweight extension on existing signatures. Floating fields allow to delay the resolution of signature avoidance as long as possible or desired. Since they do not exist at runtime, they can be simplified along type equivalence, and dropped once they became unreachable. We give a simple equivalence criterion for the simplification of floating components without loss of type-sharing. We present a principled strategy that implements this criterion and performs much better than OCaml. Remaining floating fields, instead of polluting the code, partially disappear at functor applications and fully disappear at signature ascription, including toplevel interfaces. Residual unavoidable floating fields can be shown to the user as a last resort, either to be explicitly resolved by the user, or kept until link time. The correctness of the type system is proved by elaboration into M^{ω}, which has itself been proved sound by translation to F^{ω}. The language includes module type definitions that are kept and returned in inferred types, as long as possible. ZipML has been designed to be a specification of an improvement over OCaml with transparent ascription, a better solution to signature avoidance, while staying close to the source language and to its implementation.

ACM Reference Format:
Clément Blaudeau, Didier Rémy, and Gabriel Radanne. 2024. Avoiding signature avoidance in ML modules with zippers. 1, 1 (July 2024), 27 pages.

1 INTRODUCTION

Modularity is essential to the design, development, and maintenance of complex systems. For software systems, language-level mechanisms are crucial to manage namespaces, enforce interfaces, provide encapsulation, and promote code factorization and reuse. A wide variety of modularity techniques appear in different programming languages: from simple functions to libraries, compilation units, objects, type-classes, packages, etc. In languages of the ML family (OCaml, SML, Moscow ML, 1ML, etc.), modularity is provided by a *module system*, which forms a *separate* language layer built on top of the core language. ML modules are renowned for their expressiveness and flexibility, making them adaptable to numerous contexts, from large-scale OS-libraries [15] to complex parameterized interpreters [19]. At the most basic level, types and values are gathered in *modules*. Such modules can be *parameterized*, similarly to templates. *Signatures* govern the public interface of a module.

A key feature of ML modules is encapsulation, provided by *ascription*: a module can be forced to a certain signature. If the signature contains *abstract type fields* of the form **type** `t`, the actual definition of the type `t` will be lost, hiding away the implementation details. By using ascription, a developer can make sure that the data-structures will be accessed only through the functions defined inside the module, before the ascription. In essence, this allows enforcing arbitrarily complex invariants, ranging from simple validity, for instance in a date library, to complex runtime properties, as is essential in most data-structures. The library designers only have to ensure that public functions preserve such properties, thanks to the *language-wide guarantee* that users cannot tweak with the inner details. In practice, an abstract type field **type** `t` introduces a name for a

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

2024. XXXX-XXXX/2024/7-ART $15.00
https://doi.org/

new type, accessible for the rest of the scope. A recent overview of the other advanced features (generative and applicative functors, transparent ascription, etc.) can be found in [2].

Providing a theoretical background for the seemingly simple mechanism of abstract type fields turned out to be the crux of module systems. In their foundational paper of 1985, Mitchell and Plotkin [17] suggested to represent abstract types as *existential types*. This idea was extended by Russo [24] who explained the *extrusion* mechanism: existential types defined in a module are gathered and actually quantified at the top of modules to extend their scope. Applicative functors where identified as having higher-order existential types by Biswas [1], and the extrusion mechanism was extended to support *skolemization* [2, 23, 24]. In this setting, abstract type fields actually introduce existential type *variables*, possibly higher-order, quantified in front of the signature. This has culminated in the successful, so-called, *F-ing* line of works [2, 21–25] (among others) where most significant ML module features are specified and proven sound *by elaboration* in F^ω, the higher-order polymorphic lambda calculus. In those works the syntactic, user-writable signatures are elaborated in the more expressive language of F^ω types, where type-sharing between modules is expressed differently. The approach has been *successful* in the ability to prove the soundness and explain module features with the standard tools of F^ω.

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 [10, 11], which has not been extended to more complex constructs. The SML compiler has an official specification [16] that has also evolved over the years, but not towards an elaboration in F^ω (see [14] for more details). Furthermore, the *F-ing* 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* [2] 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 [2] type-system combined with the anchoring is not *fully syntactic* in the sense of [25]: 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 user to [4, 6, 7]. 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* line of works so far.

```
module type SORTABLE = sig type t val compare : t → t → int end
```

```
module SortedDS (Elt:SORTABLE) = struct
  module Set = struct
    type key = Elt.t
    type t = key list
    let insert key set = ...
  end
  module Dict = struct
    type key = Elt.t
    type 'a t = (key * 'a) list
    let insert (key, v) d = ...
    let keys d = List.map fst m
  end
end
```

```
module SortedDS (Elt:SORTABLE) : sig
  module Set = struct
    type key = Elt.t
    type t
    val insert: key → t → t
  end`
  module Dict = struct
    type key = Elt.t
    type 'a t
    val insert: key * 'a → 'a t → 'a t
    val keys: 'a t → Set.t
  end
end
```

Fig. 1. Abstraction in action

(2) Handle applicative functors and module identities. (3) Maintain proper sharing of types, notably via a rewriting rule called *strengthening* [10]. (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.

1.1 Modules, Abstraction and Sharing of types

Some seemingly well-formed module expressions do not admit a principal type in the source signature language. A concrete example is given in Figure 1.

The module `SortedDS` contains sorted data-structures. It is parameterized by a signature, `SORTABLE`, which describes the type of elements that can be sorted using a `compare` operation. Sortable types can then be used in data-structures that require an order: sets implemented by sorted lists in the sub-module `Set` or dictionaries implemented by sorted association lists in the sub-module `Dict`. The implementation is on the left-hand side, while the external interface is on the right hand-side. Crucially, it uses abstraction to hide the implementation as sorted lists. Indeed, exposing such detail would allow users to break the invariant that the lists are sorted, compromising the behavior of the data-structure. Another interesting point is that `Set` and `Dict` share types: the type of elements is used by both, and a dictionary can be reduced to the set of its keys. Here, the notion of *identity* of an abstract type becomes essential, as it precisely allows such sharing of types to occur.

Now, a user might instantiate such data-structures on a local type definition of *int-or-string* keys, using an inlined structure:

```
module IntOrStringDS =
  SortedDS(struct type t = String of string | Int of int let compare ... end)
```

In OCaml, this program turns out to be problematic: indeed, what should be the type of keys for `IntOrStringDS` ? The element type has no external name! This is precisely, the *avoidance problem*: infer the signature for `IntOrStringDS` while *avoiding* the unreachable type `t`. This problem arises from the combination of three mechanisms: Abstract types, created at interface boundaries, which are only compatible with themselves; sharing of abstract types, essential for module interactions, which produces inter-module dependencies; and finally, unreachable types or modules (either by a projection or functor application) which *break* such dependencies.

In practice, OCaml has a *resolution* algorithm that tries to rewrite the signature in a way that does not refer to hidden parts. This syntactic process can lead to loss of relevant type information, sometimes making the typechecker infer invalid signatures[1] and overall can be hard to predict for the user. Here, it would lead to losing the type sharing between the `Set` and `Dict` types, making it impossible to share keys.

1.1.1 Signature Avoidance. At a more abstract level, the avoidance problem boils down to the projection out of a dependent record: given a signature of the following form, how can we extract the signature `S` of the submodule `X` when it depends on the type `t` to be well-formed ?

```
module M : sig type t module X : S end
```

Sometimes there exists a principal signature `S'` that is equivalent to `S` while *avoiding* the type `t`, sometimes there exists none. Inspired by Harper and Stone [8], we take the stance that projection should make components invisible to the user but not necessarily to the typechecker: if need be, the typechecker should be able to keep the type `t` in scope. Instead of handling the whole signature, we would like to focus on `S` while still being able to refer to the rest if needed. This fits well with the purpose of *zippers* [9]. We introduce *zipper signatures* that allow us to keep the lost part of signatures, which we call *floating fields*, while focusing on a specific submodule. In this simple example, it would give:

```
⟨ type t ⟩ S
```

ZipML uses zipper signatures to keep relevant type information and therefore, delay the signature avoidance problem until a signature is forced by ascription.

Zipper also behaves gracefully in case of errors, allowing to present users with the exact fields that were lost during typechecking.

1.1.2 Applicativity, abstraction safety, and aliasing. Applicative functors where introduced by [11] and are one of the key features of OCaml. They initially relied on a simple *syntactic criterion*: 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 *)
```

The strength of this criterion is that it preserves *abstraction safety* (see [2, 23]). However, a key weakness is that giving a name to some sub-expression breaks the syntactic equality:

```
module FX = F(X)
let f': G(FX).t → G(F(X)).t = fun x → x    (* fails : the types are not equal *)
```

Here, the type of module `FX` should manifest its module-level equality with `F(X)`. For this purpose, ZipML reuses a syntactic construction, *transparent signatures* `(= P < S)`, proposed by Blaudeau et al. [2], which generalizes module aliases [5] and allows expressing module level-sharing.

1.1.3 Strengthening. In path based systems, type sharing is expressed with manifest types[10]: explicit equalities between type components. As a consequence, when copying a module, copying its signature directly would loss type sharing. Instead, all type fields should be rewritten to refer to the original module. This operation, called *strengthening*, is central to *path-based* systems such as OCaml. However, rewriting a whole signature can be costly[2] and hinder performance for very large libraries. To prevent useless rewriting while keeping type-sharing, ZipML adopts a *delayed* version

[1]See the following issues: OCaml #10491, OCaml #13173, OCaml #11441, OCaml #13230
[2]See the discussion at https://github.com/ocaml-flambda/flambda-backend/pull/1337

of strengthening: we reuse the transparent signatures `(= P < S)` of [2] to delay the strengthening of `S` by `P`, which is computed lazily when actually accessing or inspecting the signature.

1.1.4 Preventing inlining. Arbitrary substitutions in modules can make type-checking of large-scale libraries difficult. Arbitrary expansion of aliases is similarly problematic. Indeed, OCaml allows both arbitrary type aliases (`type t = int * float`) and arbitrary module type definitions (`module type T = sig .. end`). OCaml keeps such definitions internally, while both M^{ω} and *F-ing* handle them by *full inlining*. This is problematic for multiple reasons: it tends to blow up signatures, making typechecking of large libraries completely intractable; but also looses the intent of programmers by inferring signature where their aliases are not used anymore. ZipML keeps module type definitions internally and in inferred signatures, as any real-word implementation, instead of systematically inlining them. This makes ZipML closer to an actual implementation, such as OCaml, which could quickly and easily benefit from our solution to signature avoidance, as well as transparent ascriptions.

1.2 ZipML

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}.

Plan. In §2, we start with a more detailed overview of floating fields and signature zippers. In §3, we present the syntax and typing rules of ZipML. In §4, we show how signature avoidance can be delayed and resolved with an equivalence criterion on floating fields. In §5, we state formal properties of ZipML, including type soundness, for which we give the structure of a proof by elaboration into M^{ω}. Finally, we discuss missing features and future works in §6.

2 AN INTRODUCTION TO FLOATING FIELDS

The main novelty of ZipML is the introduction of floating fields, as a way to solve the signature avoidance problem. Floating fields provide additional expressiveness that allows to describe all inferred signatures. Hence, we first explain the signature avoidance problem in OCaml and how floating fields help solve it.

We use OCaml-like syntax [12] for examples. However, we use self-references for signatures[3] 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 $P.X$[4], 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:[5]

```
⟨ 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
```

[3]ZipML also uses self-references in structures, but we do not in examples to keep closer to OCaml syntax.
[4]This limitation encourages users to name intermediate structures, circumventing the avoidance problem.
[5]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.

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`.

```
⟨ 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.

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 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 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 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 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 $S_D\left[S_B\left[S_Z\right]\right]$, where the signature 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 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 S_Z while *avoiding* to mention the unreachable fields from S_D and S_B. It is usually solved by rewriting the signature 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 $\langle S_D\ [S_B]\rangle\ \mathsf{S}_Z$, or equivalently, $\langle D : S_D\rangle\ \langle B : S_B\rangle\ \mathsf{S}_Z$. Thus, projection never fails, avoiding (or delaying) the signature avoidance problem. The zipper notation gives an immediate access to S_Z but still allowing 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 S_2 equal to $\langle D : S_D\rangle\ (S_B\ [\mathsf{S}_Z])$, 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 = [·]` 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 $S_B\ [\mathsf{S}_Z]$ on Z, we get $\langle B : S_B\rangle\ \mathsf{S}_Z$. Popping the local environment S_D, we then return the signature $\langle S_D\rangle\ \langle S_B\rangle\ \mathsf{S}_Z$, which exactly corresponds to the signature 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 S_Z via their self-reference names, respectively D and B. Hence S_Z need not be changed—provided we have chosen disjoint self-references while zipping.

The next step to is transform 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 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 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 S_1 (repeated below), which is thus equivalent to S_3 and then to 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 S_3 could be simplified by removing its floating field, this was not the case of the signature S_2, since the floating field `D.X.t` is referenced in $S_B\,[\mathsf{S}_Z]$ 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 (`M : 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.

3 FORMAL PRESENTATION

3.1 Grammar

The syntax of ZipML is given in Figure 2. 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 $\overline{\mathsf{B}}$. 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** (M) and **Signatures** (S), and the usual module fields (expression, types, modules, module types) in **Bindings** (B) and **Declarations** (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 ($= P < \tilde{\mathsf{S}}$) for a transparent signature which has them the identity of P but with the interface of 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 $\mathtt{struct}_A\ \overline{\mathsf{B}}$ end and structural signatures $\mathtt{sig}_A\ \overline{\mathsf{D}}$ end to refer to the structure or the signature itself from fields $\overline{\mathsf{B}}$ or $\overline{\mathsf{D}}$: The annotation is the binding occurrence, their scope extends to $\overline{\mathsf{B}}$ or $\overline{\mathsf{D}}$, 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 $\mathtt{type}\ t = \mathsf{u}$ where 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.

Path and Prefix

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

Module Expression

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

Binding

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

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

$$\gamma ::= \varnothing \mid A : \overline{\mathsf{D}} \mid \gamma\, ;\gamma$$

Signature

$$\begin{array}{rll} \mathsf{S} ::= & \langle \gamma \rangle\, \mathsf{S} & \text{(Zipped type)} \\ \mid & \tilde{\mathsf{S}} & \text{(Unzipped type)} \\ \tilde{\mathsf{S}} ::= & (= P < \tilde{\mathsf{S}}) & \text{(Transparent signature)} \\ \mid & Q.T & \text{(Module type)} \\ \mid & () \rightarrow \mathsf{S} & \text{(Generative functor)} \\ \mid & (Y : \tilde{\mathsf{S}}) \rightarrow \mathsf{S} & \text{(Applicative functor)} \\ \mid & \mathtt{sig}_A\ \overline{\mathsf{D}}\ \mathtt{end} & \text{(Structural signature)} \end{array}$$

Declaration

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

Core language types and expression

$$\mathsf{e} ::= \cdots \mid Q.x \quad \text{(Qualified value)} \qquad \mathsf{u} ::= \cdots \mid Q.t \quad \text{(Qualified type)}$$

Fig. 2. Syntax of ZipML

We may omit the self-reference and just write $\mathtt{sig}\ \mathsf{D}\ \mathtt{end}$ for $\mathtt{sig}_A\ \mathsf{D}\ \mathtt{end}$ when A does not appear free in D. Conversely, when we write $\mathtt{sig}\ \mathsf{D}\ \mathtt{end}$, we should read $\mathtt{sig}_A\ \mathsf{D}\ \mathtt{end}$ where the anonymous self-reference A is chosen fresh for 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 typechecking. 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 $\Gamma, A.\overline{\mathsf{D}}$ for the sequence $\Gamma, \overline{A.\mathsf{D}}$. Finally, context, may also contain floating fields Γ, γ. By associativity, we identify $\Gamma, (A.\overline{\mathsf{D}}, A'.\overline{\mathsf{D}}')$ and $(\Gamma, A, \overline{\mathsf{D}}), A'.\overline{\mathsf{D}}'$.

Zippers. Finally, the novelty is the introduction of zipped signatures $\langle \gamma \rangle\, \mathsf{S}$ used to introduce zippers γ, which are sequences of floating fields $A : \bar{\mathsf{D}}$. In a zipped signature $\langle A : \mathsf{D} \rangle\, \mathsf{S}$, the self-reference A, which is used to access fields of D from S, can be renamed consistently. That is, we identify $\left\langle A : \bar{\mathsf{D}} \right\rangle \mathsf{S}$ and $\left\langle A' : \bar{\mathsf{D}}[A \mapsto A'] \right\rangle \mathsf{S}[A \mapsto A']$ provided A' is fresh for $\bar{\mathsf{D}}$ and S. Of course, a zipper taken alone cannot be renamed. The concatenation of zippers $\gamma_1 \mathbin{;} \gamma_2$ is only defined when the domains of γ_1 and γ_2 are disjoint where their domain is defined as follows:

$$\mathsf{dom}(\emptyset) = \emptyset \qquad \mathsf{dom}(A : \bar{\mathsf{D}}) = \{A\} \qquad \mathsf{dom}(\gamma_1 \mathbin{;} \gamma_2) = \mathsf{dom}(\gamma_1) \uplus \mathsf{dom}(\gamma_2)$$

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 S and $\langle \emptyset \rangle\, \mathsf{S}$ and $\langle \gamma_1 \rangle \langle \gamma_2 \rangle\, \mathsf{S}$ and $\langle \gamma_1 \mathbin{;} \gamma_2 \rangle\, \mathsf{S}$ whenever γ_1 and γ_2 have disjoint domains. Therefore a signature S can always be written as $\langle \gamma \rangle\, \tilde{\mathsf{S}}$ where $\tilde{\mathsf{S}}$ 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.[6] Finally, we define $\tilde{\mathsf{S}}$ 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.[7] The head **v**alue form S^{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 S^{n} is similar, but still contains the *identity* of a signature, if it has one, via a transparent ascription.

$$\mathsf{S}^{\mathrm{v}} \;::=\; \mathtt{sig}_A\, \bar{\mathsf{D}}\, \mathtt{end} \mid (Y : \mathsf{S}) \to \mathsf{S} \mid () \to \mathsf{S} \qquad\qquad \mathsf{S}^{\mathrm{n}} \;::=\; \mathsf{S}^{\mathrm{v}} \mid (= P < \mathsf{S}^{\mathrm{v}})$$

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 $(= P < \tilde{\mathsf{S}})$ that also allows transparent ascriptions to be placed inside zippers. The definition is the following:

$$\mathbb{S} \;::=\; \langle \mathbb{G} \rangle\, \mathbb{S} \mid \tilde{\mathbb{S}} \qquad \tilde{\mathbb{S}} \;::=\; (= P < \tilde{\mathsf{S}}) \qquad \mathbb{G} \;::=\; A : \bar{\mathbb{D}} \mid \mathbb{G} \mathbin{;} \mathbb{G} \mid \emptyset$$
$$\mathbb{D} \;::=\; \mathtt{module}\, X : \mathbb{S} \mid \mathtt{val}\, x : \mathsf{u} \mid \mathtt{module\ type}\ T = \mathsf{S} \mid \mathtt{type}\, t = \mathsf{u}$$

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 $\tilde{\mathsf{u}}$ means 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 $\mathsf{M}_1(\mathsf{M}_2)$ may be encoded as syntactic sugar for $(\mathtt{struct}_A\ \mathtt{module}\ X_1 = \mathsf{M}_1\ \mathtt{module}\ X_2 = \mathsf{M}_2\ \mathtt{module}\ X = A.X_1(A.X_2)\ \mathtt{end}).X$. Indeed, our projection $\mathsf{M}.X$ is unrestricted. By contrast, OCaml requires 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, M is currently restricted to be a path $\tilde{P}$ in ascription $(\mathsf{M} : \tilde{\mathsf{S}})$, but the general case can be encoded as $(\mathtt{struct}_A\ \mathtt{module}\ X_1 = \mathsf{M}\ \mathtt{module}\ X = (A.X_1 : \tilde{\mathsf{S}})\ \mathtt{end}).X$.

Note some grammatical ambiguity: $P.X$ may be a projection from a path P or from a module expression M which may itself be a path. This is not an issue as their typing will be the same.

[6] In the absence of floating fields, self references could only be at the origin of a path Q.
[7] For sake of readability, we do not always use the most precise syntactic categories and sometimes just write S when 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.

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 §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:

$$\begin{aligned} \mathtt{unzip}\ \left\langle A : \overline{\mathtt{D}} \cdot X \cdot \overline{\mathtt{D}}' \right\rangle \mathtt{S} &\triangleq \mathtt{sig}_A\ \overline{\mathtt{D}}, \mathtt{module}\ X : \mathtt{S}, \overline{\mathtt{D}}'\ \mathtt{end} \\ \mathtt{unzip}\ \langle \gamma_1 \,;\, \gamma_2 \rangle\, \mathtt{S} &\triangleq \mathtt{unzip}\ \langle \gamma_1 \rangle\, (\mathtt{unzip}\ \langle \gamma_2 \rangle\, \mathtt{S}) \\ \mathtt{unzip}\ \langle \emptyset \rangle\, \mathtt{S} &\triangleq \mathtt{S} \end{aligned}$$

Conversely, given a structural signature $\mathtt{S}$, $\mathtt{S} \cdot X$ is the zipper positioned at the submodule X:

$$\left(\mathtt{sig}_A\ \overline{\mathtt{D}}, \mathtt{module}\ X : \mathtt{S}, \overline{\mathtt{D}}'\ \mathtt{end}\right) \cdot X \ \triangleq\ \left\langle A : \overline{\mathtt{D}} \cdot X \cdot \overline{\mathtt{D}}' \right\rangle \mathtt{S} \qquad \mathtt{S} \cdot (X.\overline{X}) \ \triangleq\ (\mathtt{S} \cdot X) \cdot \overline{X} \qquad \mathtt{S} \cdot \emptyset \ \triangleq\ \mathtt{S}$$

We may also define the path of a zipper as the sequence of projections that returns the zipped signature from its unzipped one:

$$\mathtt{path}\ (A : \overline{\mathtt{D}} \cdot X \cdot \overline{\mathtt{D}}') = X \qquad\qquad \mathtt{path}\ (\gamma \,;\, \gamma') = (\mathtt{path}\ \gamma) \cdot (\mathtt{path}\ \gamma')$$

Then, we have $(\mathtt{unzip}\ (\langle \gamma \rangle\, \mathtt{S})) \cdot (\mathtt{path}\ \gamma)$ 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.

3.3 Strengthening and typing environment $\boxed{\mathtt{S} \,/\, P}\ \boxed{\mathtt{S} \,//\, P}$

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.[8]

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 3. 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.
- 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

[8]There is actually a proposal to add lazy strengthening in OCaml for efficiency purposes.

Delayed strengthening

$$\begin{aligned}
(= P' < \mathtt{S}) \mathbin{//} P &\triangleq (= P' < \mathtt{S}) \\
\langle \gamma \,;\, \gamma' \rangle\, \mathtt{S} \mathbin{//} P &\triangleq \big(\langle \gamma \rangle \big(\langle \gamma' \rangle\, \mathtt{S}\big)\big) \mathbin{//} P \\
\big\langle A : \overline{\mathtt{D}} \big\rangle\, \mathtt{S} \mathbin{//} P &\triangleq \big\langle A : \overline{\mathtt{D}} \mathbin{//} P \big\rangle\, (\mathtt{S}[A \mapsto P.A] \mathbin{//} P) \\
\mathtt{S} \mathbin{//} P &\triangleq (= P < \mathtt{S})
\end{aligned}$$

Shallow strengthening

$$\begin{aligned}
\mathtt{sig}_A\, \overline{\mathtt{D}}\ \mathtt{end} \,/\, P &\triangleq \mathtt{sig}_A\, \overline{\mathtt{D}[A \mapsto P] \mathbin{//} P}\ \mathtt{end} \\
(Y : \tilde{\mathtt{S}}_a) \to \mathtt{S} \,/\, P &\triangleq (Y : \tilde{\mathtt{S}}_a) \to (\mathtt{S} \mathbin{//} P(Y)) \\
() \to \mathtt{S} \,/\, P &\triangleq () \to \mathtt{S}
\end{aligned}$$

Declaration strengthening

$$\begin{aligned}
(\mathtt{val}\, x : \mathtt{u}) \mathbin{//} P &\triangleq \mathtt{val}\, x : \mathtt{u} & (\mathtt{module}\, X : \mathtt{S}) \mathbin{//} P &\triangleq \mathtt{module}\, X : (\mathtt{S} \mathbin{//} P.X) \\
(\mathtt{type}\, t = \mathtt{u}) \mathbin{//} P &\triangleq \mathtt{type}\, t = \mathtt{u} & (\mathtt{module\ type}\ T = \mathtt{S}) \mathbin{//} P &\triangleq \mathtt{module\ type}\ T = \mathtt{S}
\end{aligned}$$

Context strengthening

$$\begin{aligned}
\Gamma \uplus (Y : \mathtt{S}) &\triangleq \Gamma, Y : (\mathtt{S} \mathbin{//} Y) & \Gamma \uplus (\mathtt{module}\, A.X : \mathtt{S}) &\triangleq \Gamma, \mathtt{module}\, A.X : (\mathtt{S} \mathbin{//} A.X) \\
\Gamma \uplus (A : \overline{\mathtt{D}} \,;\, \gamma) &\triangleq (\Gamma \uplus A : (\overline{\mathtt{D}} \mathbin{//} A)), \gamma
\end{aligned}$$

Fig. 3. Strengthening (delayed by default) – $\mathtt{S} / P$ and $\mathtt{S} \mathbin{//} P$

zipper, which is neutral for zipping. Notice that strengthening commutes with unzipping. That is, unzip $(\langle \gamma \rangle\, \mathtt{S} \mathbin{//} P)$ is equal to $(\text{unzip}\, \langle \gamma \rangle\, \mathtt{S}) \mathbin{//} P$.

- 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 3 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.

3.4 Path typing $\boxed{\Gamma \vdash P : \mathbb{S}}$ $\boxed{\Gamma \vdash P \triangleright \mathbb{S}}$

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 4. 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 $\mathtt{sig}_A\, \overline{\mathbb{D}}$ 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.

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.

$$\textsc{Typ-P-Arg}\ \dfrac{Y : \mathbb{S} \in \Gamma}{\Gamma \vdash Y : \mathbb{S}} \qquad \textsc{Typ-P-Module}\ \dfrac{\texttt{module}\, A.X : \mathbb{S} \in \Gamma}{\Gamma \vdash A.X : \mathbb{S}} \qquad \textsc{Typ-P-Zip}\ \dfrac{\Gamma \vdash P : \langle \mathbb{G} \rangle\, \mathbb{S} \qquad \mathbb{G}(A) = \overline{\mathbb{D}}}{\Gamma \vdash P.A : \texttt{sig}\ \overline{\mathbb{D}}\ \texttt{end}} \qquad \textsc{Typ-P-Norm}\ \dfrac{\Gamma \vdash P : \mathbb{S} \qquad \Gamma \vdash \mathbb{S} \downarrow \mathbb{S}'}{\Gamma \vdash P : \mathbb{S}'}$$

$$\textsc{Typ-P-Proj}\ \dfrac{\Gamma \vdash P \triangleright \texttt{sig}\ \overline{\mathbb{D}}\ \texttt{end} \qquad \texttt{module}\, X : \mathbb{S} \in \overline{\mathbb{D}}}{\Gamma \vdash P.X : \mathbb{S}} \qquad \textsc{Typ-P-AppA}\ \dfrac{\Gamma \vdash P \triangleright (Y : \tilde{\mathsf{S}}_a) \to \mathbb{S} \qquad \Gamma \vdash P' : \mathbb{S}' \qquad \Gamma \vdash \mathbb{S}' \leqslant \tilde{\mathsf{S}}_a}{\Gamma \vdash P(P') : \mathbb{S}\big[Y \mapsto P'\big]}$$

Fig. 4. Path typing – $\Gamma \vdash P : \mathsf{S}$

$$\textsc{Res-P-Id}\ \dfrac{\Gamma \vdash P : \langle \mathbb{G} \rangle\, (= P' < \tilde{\mathsf{S}})}{\Gamma \vdash P \triangleright P'} \qquad \textsc{Res-P-Val}\ \dfrac{\Gamma \vdash P : \langle \mathbb{G} \rangle\, (= P' < \tilde{\mathsf{S}}) \qquad \tilde{\mathbb{S}} = \tilde{\mathsf{S}} \,/\, P'}{\Gamma \vdash P \triangleright \tilde{\mathbb{S}}}$$

Fig. 5. Path Resolution – $\Gamma \vdash P \triangleright P'$ and $\Gamma \vdash P \triangleright \tilde{\mathbb{S}}$

$$\textsc{Norm-S-Zip}\ \dfrac{\Gamma \uplus \gamma \vdash \mathsf{S} \downarrow \mathsf{S}'}{\Gamma \vdash \langle \gamma \rangle\, \mathsf{S} \downarrow \langle \gamma \rangle\, \mathsf{S}'} \qquad \textsc{Norm-S-Trans-Some}\ \dfrac{\Gamma \vdash \mathsf{S} \downarrow (= P' < \mathsf{S})}{\Gamma \vdash (= P < \mathsf{S}) \downarrow (= P' < \mathsf{S}')} \qquad \textsc{Norm-S-Trans-None}\ \dfrac{\Gamma \vdash P \triangleright P' \qquad \Gamma \vdash \mathsf{S} \downarrow \mathsf{S}'}{\Gamma \vdash (= P < \mathsf{S}) \downarrow (= P' < \mathsf{S}')} \qquad \textsc{Norm-Typ-Res}\ \dfrac{\Gamma \vdash P \triangleright P'}{\Gamma \vdash P.t \downarrow P'.t}$$

$$\textsc{Norm-S-LocalModType}\ \dfrac{\texttt{module type}\ A.T = \mathsf{S} \in \Gamma}{\Gamma \vdash A.T \downarrow \mathsf{S}} \qquad \textsc{Norm-S-PathModType}\ \dfrac{\Gamma \vdash P \triangleright \texttt{sig}\ \overline{\mathsf{D}}\ \texttt{end} \qquad \texttt{module type}\ T = \mathsf{S} \in \overline{\mathsf{D}}}{\Gamma \vdash P.T \downarrow \mathsf{S}} \qquad \textsc{Norm-Typ-Local}\ \dfrac{\texttt{type}\, A.t = \mathsf{u} \in \Gamma}{\Gamma \vdash A.t \downarrow \mathsf{u}} \qquad \textsc{Norm-Typ-Path}\ \dfrac{\Gamma \vdash P \triangleright \texttt{sig}\ \overline{\mathsf{D}}\ \texttt{end} \qquad \texttt{type}\, t = \mathsf{u} \in \overline{\mathsf{D}}}{\Gamma \vdash P.t \downarrow \mathsf{u}}$$

Fig. 6. Signature and type normalization – $\Gamma \vdash \mathsf{S} \downarrow \mathsf{S}'$

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 5, 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.

3.5 Normalization $\boxed{\Gamma \vdash \mathsf{S} \downarrow \mathsf{S}'}$

The judgment $\Gamma \vdash \mathsf{S} \downarrow S'$ *allows* the (head) normalization of a signature S into 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. Rules Norm-S-Trans-Some and Norm-S-Trans-None allows normalization under a transparent ascription. If the (partial) normal form of 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.

The judgment $\Gamma \vdash P.t \downarrow \mathsf{u}$ 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 $Q.t$.

3.6 Subtyping judgments $\boxed{\Gamma \vdash \mathsf{S} \leqslant \tilde{\mathsf{S}}}$

The subtyping judgment $\Gamma \vdash \mathsf{S} \leqslant \tilde{\mathsf{S}}$ is only defined when the target signature is an elaborated signature $\tilde{\mathsf{S}}$ and therefore is well-formed, does not contain zippers nor zipper accesses (nor chains of transparent ascriptions). The left-hand side signature S is also a well-formed signature resulting

$$\textsc{Sub-S-Norm}\quad \frac{\Gamma \vdash \mathsf{S}_1 \downarrow \mathsf{S}_1' \qquad \Gamma \vdash \mathsf{S}_2 \downarrow \mathsf{S}_2' \qquad \Gamma \vdash \mathsf{S}_1' \leqslant \mathsf{S}_2'}{\Gamma \vdash \mathsf{S}_1 \leqslant \mathsf{S}_2}$$

$$\textsc{Sub-S-Zipper}\quad \frac{\Gamma \uplus \gamma \vdash \mathsf{S}_1 \leqslant \mathsf{S}_2}{\Gamma \vdash \langle \gamma \rangle\, \mathsf{S}_1 \leqslant \mathsf{S}_2}$$

$$\textsc{Sub-S-TrAscr}\quad \frac{\Gamma \vdash \mathsf{S}_1 \,/\, P \leqslant \mathsf{S}_2 \,/\, P}{\Gamma \vdash (= P < \mathsf{S}_1) \leqslant (= P < \mathsf{S}_2)}$$

$$\textsc{Sub-S-LooseAlias}\quad \frac{\Gamma \vdash \mathsf{S}_1 \,/\, P \leqslant \mathsf{S}_2}{\Gamma \vdash (= P < \mathsf{S}_1) \leqslant \mathsf{S}_2}$$

$$\textsc{Sub-S-FctG}\quad \frac{\Gamma \vdash \mathsf{S}_1 \leqslant \mathsf{S}_2}{\Gamma \vdash () \to \mathsf{S}_1 \leqslant () \to \mathsf{S}_2}$$

$$\textsc{Sub-S-FctG}\quad \frac{\Gamma \vdash \tilde{\mathsf{S}}_{a_2} \leqslant \tilde{\mathsf{S}}_{a_1} \qquad \Gamma \uplus Y : \tilde{\mathsf{S}}_{a_2} \vdash \mathsf{S}_1 \leqslant \mathsf{S}_2}{\Gamma \vdash (Y : \tilde{\mathsf{S}}_{a_1}) \to \mathsf{S}_1 \leqslant (Y : \tilde{\mathsf{S}}_{a_2}) \to \mathsf{S}_2}$$

$$\textsc{Sub-S-Sig}\quad \frac{\overline{\mathsf{D}_0} \sqsubset_{\leqslant} \overline{\mathsf{D}_1} \qquad \Gamma \uplus A : \overline{\mathsf{D}_1} \vdash \overline{\mathsf{D}_0} \leqslant \overline{\mathsf{D}_2}}{\Gamma \vdash \mathtt{sig}_A\ \overline{\mathsf{D}_1}\ \mathtt{end} \leqslant \mathtt{sig}_A\ \overline{\mathsf{D}_2}\ \mathtt{end}}$$

$$\textsc{Sub-S-DynEq}\quad \frac{\Gamma \vdash \mathsf{S} \lesssim \mathsf{S}' \qquad \Gamma \vdash \mathsf{S}' \lesssim \mathsf{S}}{\Gamma \vdash \mathsf{S} \approx \mathsf{S}'}$$

$$\textsc{Sub-T-Norm}\quad \frac{\Gamma \vdash \mathsf{u}_1 \downarrow \mathsf{u} \qquad \Gamma \vdash \mathsf{u}_2 \downarrow \mathsf{u}}{\Gamma \vdash \mathsf{u}_1 \leqslant \mathsf{u}_2}$$

$$\textsc{Sub-D-Mod}\quad \frac{\Gamma \vdash \mathsf{S}_1 \leqslant \mathsf{S}_2}{\Gamma \vdash \mathtt{module}\, X : \mathsf{S}_1 \leqslant \mathtt{module}\, X : \mathsf{S}_2}$$

$$\textsc{Sub-D-Val}\quad \frac{\Gamma \vdash \mathsf{u}_1 \leqslant \mathsf{u}_2}{\Gamma \vdash \mathtt{val}\, x : \mathsf{u}_1 \leqslant \mathtt{val}\, x : \mathsf{u}_2}$$

$$\textsc{Sub-D-ModType}\quad \frac{\Gamma \vdash \mathsf{S}_1 \approx \mathsf{S}_2}{\Gamma \vdash \mathtt{module\ type}\ T = \mathsf{S}_1 \leqslant \mathtt{module\ type}\ T = \mathsf{S}_2}$$

$$\textsc{Sub-D-Type}\quad \frac{\Gamma \vdash \mathsf{u}_1 \approx \mathsf{u}_2}{\Gamma \vdash \mathtt{type}\, t = \mathsf{u}_1 \leqslant \mathtt{type}\, t = \mathsf{u}_2}$$

Fig. 7. Subtyping – $\Gamma \vdash \mathsf{S} \leqslant \tilde{\mathsf{S}}$

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 7. 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 S_1may 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 $\overline{\mathsf{D}_0}$ related to $\overline{\mathsf{D}_1}$ by $\sqsubset_{\leqslant}$, and subtyped pointwise with $\overline{\mathsf{D}_2}$, but in an environment extended with $A.\overline{\mathsf{D}_1}$.

The main subtyping relation is $\leqslant$ defined as $\leqslant_{\subseteq}$, i.e., using $\subseteq$ for $\sqsubset_{\leqslant}$. Namely, $\overline{\mathsf{D}_0}$ can be any subset of $\overline{\mathsf{D}_1}$ 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 $\overline{\mathsf{D}_0} \sqsubset_{\lesssim} \overline{\mathsf{D}_1} \ \triangleq\ \overline{\mathsf{D}_0} \subseteq \overline{\mathsf{D}_1} \wedge \mathsf{dyn}(\overline{\mathsf{D}_0}) = \mathsf{dyn}(\overline{\mathsf{D}_1})$ and $\mathsf{dyn}(\overline{\mathsf{D}})$ returns the subsequence

$$\textsc{Typ-S-ModType}\ \frac{\Gamma \vdash \tilde{Q}.T : \mathsf{S}}{\Gamma \vdash \tilde{Q}.T} \qquad \textsc{Typ-S-GenFct}\ \frac{\Gamma \vdash \mathsf{S}}{\Gamma \vdash () \rightarrow \mathsf{S}} \qquad \textsc{Typ-S-AppFct}\ \frac{\Gamma \vdash \tilde{\mathsf{S}}_a \qquad \Gamma \uplus (Y : \tilde{\mathsf{S}}_a) \vdash \mathsf{S}}{\Gamma \vdash (Y : \tilde{\mathsf{S}}_a) \rightarrow \mathsf{S}} \qquad \textsc{Typ-S-Ascr}\ \frac{\Gamma \vdash \mathsf{S} \qquad \Gamma \vdash \tilde{P} : \mathbb{S} \qquad \Gamma \vdash \mathbb{S} \leqslant \mathsf{S}}{\Gamma \vdash (= \tilde{P} < \mathsf{S})}$$

$$\textsc{Typ-S-Str}\ \frac{\Gamma \vdash_A \overline{\mathsf{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{sig}_A\ \overline{\mathsf{D}}\ \mathtt{end}} \qquad \textsc{Typ-D-Val}\ \frac{\Gamma \vdash \tilde{\mathsf{u}}}{\Gamma \vdash_A (\mathtt{val}\, x : \tilde{\mathsf{u}})} \qquad \textsc{Typ-D-Type}\ \frac{\Gamma \vdash \tilde{\mathsf{u}}}{\Gamma \vdash_A (\mathtt{type}\, t = \tilde{\mathsf{u}})} \qquad \textsc{Typ-D-TypeAbs}\ \Gamma \vdash_A (\mathtt{type}\, t = A.t)$$

$$\textsc{Typ-D-Mod}\ \frac{\Gamma \vdash \mathsf{S}}{\Gamma \vdash_A (\mathtt{module}\, X : \mathsf{S})} \qquad \textsc{Typ-D-ModType}\ \frac{\Gamma \vdash \mathsf{S}}{\Gamma \vdash_A (\mathtt{module\ type}\ T = \mathsf{S})} \qquad \textsc{Typ-D-Empty}\ \Gamma \vdash_A \varnothing \qquad \textsc{Typ-D-Seq}\ \frac{\Gamma \vdash_A \mathsf{D}_0 \qquad \Gamma \uplus A.\mathsf{D}'_0 \vdash_A \overline{\mathsf{D}}}{\Gamma \vdash_A (\mathsf{D}_0, \overline{\mathsf{D}})}$$

Fig. 8. Signature elaboration (all signatures are $\tilde{\mathsf{S}}$) – $\Gamma \vdash \tilde{\mathsf{S}}$

of $\overline{\mathsf{D}}$ 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. 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 $\Gamma \vdash \mathsf{u}_1 \approx \mathsf{u}_2$ which means code-free subtyping in both directions, i.e., $\Gamma \vdash \mathsf{u}_1 \leqslant \mathsf{u}_2$ and $\Gamma \vdash \mathsf{u}_2 \leqslant \mathsf{u}_1$.

3.7 Signature typing $\boxed{\Gamma \vdash \tilde{\mathsf{S}}}$

Source signatures provided by users are not necessarily well formed, and thus must be typed, using the judgment $\Gamma \vdash \tilde{\mathsf{S}}$, defined on Figure 8. The resulting source signature $\tilde{\mathsf{S}}$ 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 S and D for signatures and declarations, for the sake of readability. The signature $\tilde{\mathsf{S}}$ 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 S. Rule Typ-S-Str for structural signatures delays most of the work to the elaboration judgment $\Gamma \vdash_A \overline{\mathsf{D}}$ for declarations, which carries the self-reference variable A which should be chosen fresh for Γ, as it now appears free in declarations $\overline{\mathsf{D}}$. Rule Typ-D-Seq for sequence of declarations pushes $A.\mathsf{D}_0$ in the context while typing the remaining sequence $\overline{\mathsf{D}}$. 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 $\Gamma \vdash \mathsf{S}$ by an elaboration judgment $\Gamma \vdash \mathsf{S} : \mathsf{S}'$ that returns an elaborated signature S'. In fact, this would be necessary if we allowed declarations `open` S and `include` S that should always be elaborated. We have not included them, but the type system has been designed to allow them.

$$\textsc{Typ-M-Norm}\quad \frac{\Gamma \vdash^{\diamond} \mathsf{M} : \mathsf{S} \qquad \Gamma \vdash \mathsf{S} \downarrow \mathsf{S}'}{\Gamma \vdash^{\diamond} \mathsf{M} : \mathsf{S}'} \qquad \textsc{Typ-M-Mode}\quad \frac{\Gamma \vdash^{\triangledown} \mathsf{M} : \mathsf{S}}{\Gamma \vdash^{\blacktriangledown} \mathsf{M} : \mathsf{S}}$$

$$\textsc{Typ-M-Ascr}\quad \frac{\Gamma \vdash \tilde{\mathsf{S}} \qquad \Gamma \vdash \tilde{P} : \mathsf{S}'' \qquad \Gamma \vdash \mathsf{S}'' \preccurlyeq \tilde{\mathsf{S}}}{\Gamma \vdash^{\diamond} (\tilde{P} : \tilde{\mathsf{S}}) : \tilde{\mathsf{S}}} \qquad \textsc{Typ-M-Path}\quad \frac{\Gamma \vdash \tilde{P} : (= P' < \mathsf{S})}{\Gamma \vdash^{\diamond} \tilde{P} : (= \tilde{P} < \mathsf{S})}$$

$$\textsc{Typ-M-FctG}\quad \frac{\Gamma \vdash^{\blacktriangledown} \mathsf{M} : \mathsf{S}}{\Gamma \vdash^{\diamond} () \rightarrow \mathsf{M} : () \rightarrow \mathsf{S}} \qquad \textsc{Typ-M-AppG}\quad \frac{\Gamma \vdash P \triangleright () \rightarrow \mathsf{S}}{\Gamma \vdash^{\blacktriangledown} P() : \mathsf{S}}$$

$$\textsc{Typ-M-FctA}\quad \frac{\Gamma \vdash \tilde{\mathsf{S}}_a \qquad \Gamma \uplus Y : \tilde{\mathsf{S}}_a \vdash^{\triangledown} \mathsf{M} : \mathsf{S}}{\Gamma \vdash^{\diamond} (Y : \tilde{\mathsf{S}}_a) \rightarrow \mathsf{M} : (Y : \mathsf{S}'_a) \rightarrow \mathsf{S}}$$

$$\textsc{Typ-M-Str}\quad \frac{\Gamma \vdash_A \bar{\mathsf{B}} : \bar{\mathsf{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{struct}_A\ \bar{\mathsf{B}}\ \mathtt{end} : \mathtt{sig}_A\ \bar{\mathsf{D}}\ \mathtt{end}}$$

$$\textsc{Typ-M-Proj}\quad \frac{\Gamma \vdash^{\diamond} \mathsf{M} : \langle \gamma \rangle\, \mathsf{S} \qquad \Gamma \uplus \gamma \vdash \mathsf{S} \triangleright \mathtt{sig}_A\ \bar{\mathsf{D}}; \mathtt{module}\ X : \mathsf{S}'; \bar{\mathsf{D}}'\ \mathtt{end}}{\Gamma \vdash^{\diamond} \mathsf{M}.X : \big\langle \gamma\, ; A : \bar{\mathsf{D}} \big\rangle\, \mathsf{S}'}$$

$$\textsc{Typ-B-Seq}\quad \frac{\Gamma \vdash_A^{\diamond} \mathsf{B}_0 : \mathsf{D}_0 \qquad \Gamma \uplus A.\mathsf{D}_0 \vdash_A^{\diamond} \bar{\mathsf{B}} : \bar{\mathsf{D}}}{\Gamma \vdash_A^{\diamond} A, \mathsf{B}_0, \bar{\mathsf{B}} : \mathsf{D}_0, \bar{\mathsf{D}}} \qquad \textsc{Typ-B-Typ-Bind}\quad \frac{\Gamma \vdash \tilde{\mathsf{u}} : \mathsf{u}'}{\Gamma \vdash_A^{\diamond} (\mathtt{type}\, t = \tilde{\mathsf{u}}) : (\mathtt{type}\, t = \tilde{\mathsf{u}})}$$

$$\textsc{Typ-B-Empty}\quad \Gamma \vdash_A^{\diamond} \varnothing : \varnothing$$

$$\textsc{Typ-B-AbsType}\quad \Gamma \vdash_A^{\diamond} (\mathtt{type}\, t = A.t) : (\mathtt{type}\, t = A.t) \qquad \textsc{Typ-B-Let}\quad \frac{\Gamma \vdash^{\diamond} \mathsf{e} : \mathsf{u}}{\Gamma \vdash_A^{\diamond} (\mathtt{let}\, x = \mathsf{e}) : (\mathtt{val}\, x : \mathsf{u})}$$

$$\textsc{Typ-B-Mod}\quad \frac{\Gamma \vdash^{\diamond} \mathsf{M} : \mathsf{S}}{\Gamma \vdash_A^{\diamond} (\mathtt{module}\, X = \mathsf{M}) : (\mathtt{module}\, X : \mathsf{S})} \qquad \textsc{Typ-B-ModType}\quad \frac{\Gamma \vdash \tilde{\mathsf{S}} : \mathsf{S}'}{\Gamma \vdash_A^{\diamond} (\mathtt{module\ type}\ T = \tilde{\mathsf{S}}) : (\mathtt{module\ type}\ T = \mathsf{S}')}$$

Fig. 9. Typing rules

3.8 Module typing $\boxed{\Gamma \vdash^{\diamond} \mathsf{M} : \mathsf{S}}$

The typing judgment $\Gamma \vdash^{\diamond} \mathsf{M} : \mathsf{S}$ for module expressions is given on Figure 9. The $\diamond$ 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 $\diamond$ metavariable. Many rules use the same metavariable $\diamond$ 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 $(= P' < \mathsf{S})$. However, we return the "more recent" identity $(= \tilde{P} < \mathsf{S})$, 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 $(\tilde{P} : \tilde{\mathsf{S}})$ gives the source path $\tilde{P}$ the (elaborated) signature of the source signature $\tilde{\mathsf{S}}$ 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 $\tilde{\mathsf{S}}'$ of $\tilde{\mathsf{S}}$ which is not strengthened by $\tilde{P}$. However, we can also use this construct to implement transparent ascription $(\tilde{P} < \tilde{\mathsf{S}})$ as syntactic sugar for $(P : (= \tilde{P} < \tilde{\mathsf{S}}))$, which then returns the view 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 $A.\mathsf{D}_0$ into the context while typing the $\overline{\mathsf{D}}$, much as Typ-D-Seq for signatures.

Finally, Rule Typ-M-Proj for typing a projection $\mathsf{M}.X$ is the key rule that leverages zippers. Thanks to the use floating components, we can easily deal with the general case when M is not a path, without running into signature avoidance. Intuitively, it just returns the signature S' of the field X of the signature S of M zipped in the prefix of S before the field X. Still, we have to consider that the signature of M may itself be in a zipper γ, which is then composed with the prefix zipper, resulting in $\gamma ; A : \overline{\mathsf{D}}$. Notice also that we must push γ in the context while resolving the type of S since 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 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.

4.1 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.

Code-free equivalence $\approx$, defined in Rule Sub-S-DynEq, §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 $\tilde{\mathsf{S}}$. 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

$$\frac{\begin{array}{c}\Gamma' = \Gamma \uplus A : \mathsf{D}_1 ; \mathtt{module}\, X : \mathsf{S}_1 ; \overline{\mathsf{D}_1}' \\ \overline{\mathsf{D}_0} \cdot \overline{\mathsf{D}_0}' \sqsubset_{\leqslant} \overline{\mathsf{D}_1} \cdot \overline{\mathsf{D}_1}' \qquad \Gamma' \vdash \overline{\mathsf{D}_0} \leqslant \overline{\mathsf{D}_2} \qquad \Gamma' \vdash \mathsf{S}_1 \leqslant \mathsf{S}_2 \qquad \Gamma' \vdash \overline{\mathsf{D}_0}' \leqslant \overline{\mathsf{D}_2}'\end{array}}{\Gamma \vdash \left\langle A : \overline{\mathsf{D}_1} \cdot X \cdot \overline{\mathsf{D}_1}' \right\rangle \mathsf{S}_1 \leqslant \left\langle A : \overline{\mathsf{D}_2} \cdot X \cdot \overline{\mathsf{D}_2}' \right\rangle \mathsf{S}_2}$$

The rule has been designed so that it commutes with zipping, i.e., $\Gamma \vdash \langle \gamma \rangle\, \mathsf{S} \leqslant \langle \gamma' \rangle\, \mathsf{S}'$ holds if and only if $\Gamma \vdash \mathrm{unzip}\, \langle \gamma \rangle\, \mathsf{S} \leqslant \mathrm{unzip}\, \langle \gamma' \rangle\, \mathsf{S}'$. 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 $\overline{\mathsf{D}_0} ; \overline{\mathsf{D}_0}'$ and the $\sqsubset_{\leqslant}$ relation. By instantiating this rule with $\sqsubset_{\lesssim}$, 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 $\leq$ called zipper-subtyping where $\sqsubset_{\leq}$ is defined as:[9]

$$\overline{\mathsf{D}_0} \cdot \overline{\mathsf{D}_0}' \sqsubset_{\leq} \overline{\mathsf{D}_1} \cdot \overline{\mathsf{D}_1}' \ \triangleq\ \overline{\mathsf{D}_0} = \overline{\mathsf{D}_1} \wedge \overline{\mathsf{D}_0}' \subseteq \overline{\mathsf{D}_1}' \qquad\qquad \overline{\mathsf{D}_0} \sqsubset_{\leq} \overline{\mathsf{D}_1} \ \triangleq\ \overline{\mathsf{D}_0} = \overline{\mathsf{D}_1}$$

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 $(\leq ; \approx)$ that composes code-free equivalence with zipper-subtyping. This relation is still code-free and commutes with subtyping, that is, $(\lesssim ; \leq) \subseteq (\leq ; \lesssim)$. This indirectly implies that typechecking will never fail because of $\lesssim$-simplifications.

Since $\lesssim$ is a code-free subtyping relation, we may extend the typing judgment with the following floating (i.e., non syntactic) typing rule:

$$\textsc{Typ-M-Simplify}\quad \frac{\Gamma \vdash \mathsf{M} : \mathsf{S} \qquad \Gamma \vdash \mathsf{S} \lesssim \mathsf{S}'}{\Gamma \vdash \mathsf{M} : \mathsf{S}'}$$

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.

4.2 Simplification algorithm

In this section, we propose a simplification algorithm that transforms a signature along $\lesssim$, 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 [2], 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.

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.)

4.2.1 *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 $\langle A : \mathtt{type}\ t = A.t\rangle\, \mathsf{S}$ reified as $\langle A : \mathtt{type}\ t = A.t \cdot \mathbb{Z} \cdot \emptyset\rangle\, \mathsf{S}$ into $\langle A : \emptyset \cdot \mathbb{Z} \cdot\ \mathtt{type}\ t = \mathbb{Z}.P.t'\rangle\, \mathsf{S}'$, 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 $\mathtt{type}\, t' = A.t$ 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.

[9]Formally, we should now see $\sqsubset_{\leq}$ 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.

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;
- Anchorable $P_0.t_0$: if an anchoring point `type` $t_0 = A.t$ has been found at position $\mathbb{Z}.P_0.t_0$ in S.

We then define a visit$_P$ 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;[10]
- 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 visit$_{\mathbb{Z}}$ 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
- 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 Anchorable $\mathbb{Z}.X_1.X_2.t'$, occurrences of $A.t$ should be replaced by

- $A_2.t'$ 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 S.

For the rest of this section, we refer to this technique as *contextual path substitution.*

4.2.2 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:

$$\langle A : \texttt{module}\ X : \texttt{sig}_B\ (\texttt{type}\ t_1 = B.t_1\ \texttt{type}\ t_2 = B.t_2)\ \texttt{end}\rangle\ \texttt{sig}_C\ \ (\texttt{type}\ t_3 = A.X.t_1\ \texttt{val}\ x : A.X.t_2)\ \texttt{end}$$

We just extend the previous algorithm using a multi-point map φ. At the end of the visiting phase, it would give here:

$$A.X.t_1 \mapsto \text{Anchorable}\ \mathbb{Z}.t_3\ ;\ A.X.t_2 \mapsto \text{Used}$$

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 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.

[10]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.

4.2.3 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 $\texttt{module}\, X : \mathsf{S}$ are accessible module declarations of the form $\texttt{module}\, X' : (= X < \mathsf{S}')$ with the additional constraint that S' is a subtype of 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 $\texttt{module}\, X' : (= X' < \mathsf{S}')$ that is equal to itself, which we rewrite in $\texttt{module}\, X' : \mathsf{S}' \mathbin{/} X'$; then we continue with the substitution inside S. If X appears as a prefix or inside a functor application, it is also substituted.

Let us consider a small example:

$$\begin{array}{l}\langle A : \texttt{module}\ X : (\texttt{sig}_B\ \texttt{type}\ t_1 = B.t_1\ \texttt{type}\ t_2 = B.t_2\ \texttt{end})\rangle \\ \qquad \texttt{sig}_C\ \texttt{type}\ t_3 = A.X.t_1\ \texttt{module}\ X' : (= A.X < (\texttt{sig}_B\ \texttt{type}\ t_1 = B.t_1\ \texttt{type}\ t_2 = B.t_2\ \texttt{end}))\ \texttt{end}\end{array}$$

After the visit of the signature, we get the following map:

$$A.X.t_1 \mapsto \text{Anchorable}\, \mathbb{Z}.t_3\ ;\ A.X \mapsto \text{Anchorable}\, \mathbb{Z}.X'\ ;\ A.X.t_2 \mapsto \text{Anchorable}\, \mathbb{Z}.X'.t_2\ ;$$

After the substitution, we would get the signature:

$$\texttt{sig}_C\ \texttt{type}\ t_3 = C.t_3\ \texttt{module}\ X' : (= C.X' < (\texttt{sig}_B\ \texttt{type}\ t_1 = C.t_3\ \texttt{type}\ t_2 = C.X.t_2\ \texttt{end}))\ \texttt{end}$$

Actually, reinstalling self-references during the substitution, we finally get:

$$\texttt{sig}_C\ \texttt{type}\ t_3 = C.t_3\ \texttt{module}\ X' : (\texttt{sig}_B\ \texttt{type}\ t_1 = C.t_3\ \texttt{type}\ t_2 = B.t_2\ \texttt{end})\ \ \texttt{end}$$

4.2.4 Functor applications. Supporting functor application does not require modifying the map. As done in [2], 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.

4.2.5 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:

$$\langle A : \texttt{type}\ t = A.t\ \texttt{module}\ F : (Y : \texttt{sig}\ \texttt{type}\ u = A.t\ \texttt{end}) \to \ldots \rangle\, \mathsf{S}$$

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 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.

4.3 Restructuring zippers

In fact, while $\lesssim$ 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 $\gtrless\!\!\lesssim$ as $(\gtrsim ; \lesssim)$ 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.

$$\textsc{Typ-M-Generalize}\quad \frac{\Gamma \vdash \mathsf{M} : \mathsf{S} \qquad \Gamma \vdash \mathsf{S} \gtrsim \mathsf{S}'}{\Gamma \vdash \mathsf{M} : \mathsf{S}'} \qquad\qquad \textsc{Typ-M-Equiv}\quad \frac{\Gamma \vdash \mathsf{M} : \mathsf{S} \qquad \Gamma \vdash \mathsf{S} \gtrless\!\!\lesssim \mathsf{S}'}{\Gamma \vdash \mathsf{M} : \mathsf{S}'}$$

5 PROPERTIES

5.1 Soundness of ZipML by Elaboration in M$^\omega$

We now present the elaboration from ZipML to M$^\omega$ [2]. We expect the reader to be familiar with elaboration of ML modules into F$^\omega$, ideally M$^\omega$ [2] or *F-ing* [23]. This elaboration serves as a proof of soundness of the system. Elaborating in M$^\omega$ rather than directly in F$^\omega$ allows to benefit from the sound extrusion and skolemization of M$^\omega$. The goal is to show that every module expression that typechecks in ZipML also typechecks in M$^\omega$ (as they have the same source language):

$$\vdash^{\diamond} \mathsf{M} : \mathsf{S} \quad\Longrightarrow\quad \overset{\omega}{\vdash} \mathsf{M} : \exists^{\diamond}\overline{\alpha}.\mathcal{C} \qquad \text{(For some } \overline{\alpha}, \mathcal{C}\text{)}$$

We write $\overset{\omega}{\vdash}$ for judgments in M$^\omega$, 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 $\Delta \overset{\omega}{\vdash} \mathsf{S} : \mathcal{S}$ where Δ and $\mathcal{S}$ are M$^\omega$ typing environments and signatures. We may then define $[\![\Gamma]\!]$ by folding the elaboration of signatures over Γ and $[\![\Gamma \vdash \mathsf{S}]\!]$ for the signature $\mathcal{S}$ such that $[\![\Gamma]\!] \vdash \mathsf{S} : \mathcal{S}$. In standalone M$^\omega$, 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:

$$\Gamma \vdash^{\diamond} \mathsf{M} : \mathsf{S} \quad\Longrightarrow\quad [\![\Gamma]\!] \overset{\omega}{\vdash} \mathsf{M} : [\![\Gamma \vdash \mathsf{S}]\!]^{\diamond} \qquad \text{(Soundness statement)}$$

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$^\omega$.

5.1.1 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$^\omega$, with the introduction of zippers in signatures and zipper accesses in paths. Intuitively, zippers are removed at runtime, and could be removed in M$^\omega$. 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$^\omega$ signatures and environments with zippers. We add zippers to M$^\omega$ signatures with the following signature elaboration rule, along with appropriate path typing extension.

$$\textsc{Typ-S-Zip}\quad \frac{\Gamma \overset{\omega}{\vdash}_{A} \overline{\mathsf{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}} \qquad \Gamma, A : \overline{\mathcal{D}} \overset{\omega}{\vdash} \mathsf{S} : \lambda\overline{\beta}.\mathcal{C}}{\Gamma \overset{\omega}{\vdash} \left\langle A : \overline{\mathsf{D}} \right\rangle \mathsf{S} : \lambda\overline{\alpha}, \overline{\beta}. \left\langle A : \overline{\mathcal{D}} \right\rangle \mathcal{C}}$$

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.

5.1.2 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: $\Gamma; \texttt{module}\, A.X : \mathsf{S}$. 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:

$$\frac{[\![\Gamma]\!]; \overline{\alpha}; \texttt{module}\, A.X : \mathcal{C} \overset{\omega}{\vdash} \mathsf{S} : \mathcal{C}}{[\![\Gamma; \texttt{module}\, A.X : \mathsf{S}]\!] = [\![\Gamma]\!]; \overline{\alpha}; \texttt{module}\, A.X : \mathcal{C}}$$

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 §3.3. From $(\Gamma \uplus \texttt{module}\, A.X : \mathsf{S})$, we can "catch" the signature S before its strengthening and use it to find $\mathcal{C}$. Then, the elaboration of S is $\Gamma \overset{\omega}{\vdash} \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}$. 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 $\mathsf{S} \,//\, P$ is the same as elaboration of the signature S before strengthening is applied to some type variables:

Lemma 5.1 (Elaboration of strengthening). *Given a path P and a signature S, such that $[\![\Gamma]\!] \overset{\omega}{\vdash} P : \mathcal{C}'$ and $[\![\Gamma]\!] \overset{\omega}{\vdash} \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}$ and $[\![\Gamma]\!] \overset{\omega}{\vdash} \mathcal{C}' < \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]$, we have $[\![\Gamma]\!] \overset{\omega}{\vdash} \mathsf{S} \,//\, P : (\lambda\overline{\alpha}.\mathcal{C})\,\overline{\tau}$.*

5.1.3 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 5.2 (Elaboration of normalization). *If S normalizes to S' in ZipML ($\Gamma \vdash \mathsf{S} \downarrow \mathsf{S}'$), then both signatures have the same elaboration in M^ω. Namely, $[\![\Gamma]\!] \overset{\omega}{\vdash} \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}$ implies $[\![\Gamma]\!] \overset{\omega}{\vdash} \mathsf{S}' : \lambda\overline{\alpha}.\mathcal{C}$.*

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:

$$(\tau, \mathcal{C}) \triangleq \text{sig type } id = \tau; \text{module } Val : \mathcal{C} \text{ end}$$

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 5.3 (Elaboration of path-typing and resolution). *If $\Gamma \vdash P : \mathsf{S}$ then we have both $[\![\Gamma]\!] \overset{\omega}{\vdash} P : (\tau, \mathcal{C})$ and $[\![\Gamma]\!] \overset{\omega}{\vdash} \mathsf{S} : (\tau, \mathcal{C})$. Moreover,*

- *If $\Gamma \vdash P \triangleright \mathsf{S}$ then P and S have the same signature in M^ω: $[\![\Gamma]\!] \overset{\omega}{\vdash} P : (\tau, \mathcal{C})$ and $[\![\Gamma]\!] \overset{\omega}{\vdash} \mathsf{S} : (\tau, \mathcal{C})$.*
- *If $\Gamma \vdash P \triangleright P'$ then P and P' have the same identity in M^ω: $[\![\Gamma]\!] \overset{\omega}{\vdash} P : (\tau, \_)$ and $[\![\Gamma]\!] \overset{\omega}{\vdash} P' : (\tau, \_)$.*

Lemma 5.4 (Elaboration of subtyping). *The elaboration preserves subtyping relationship: If $[\![\Gamma]\!] \overset{\omega}{\vdash} \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}$ and $[\![\Gamma]\!] \overset{\omega}{\vdash} \mathsf{S}' : \lambda\overline{\alpha}'.\mathcal{C}'$ then $\Gamma \vdash \mathsf{S} \leqslant \mathsf{S}'$ implies $[\![\Gamma]\!] \overset{\omega}{\vdash} \lambda\overline{\alpha}.\mathcal{C} < \lambda\overline{\alpha}.\mathcal{C}'$.*

Theorem 5.5 (Soundness). *If $\Gamma \vdash^{\diamond} \mathsf{M} : \mathsf{S}$ then $\exists \mathcal{S}$ such that $[\![\Gamma]\!] \overset{\omega}{\vdash} \mathsf{S} : \mathcal{S}$ and $[\![\Gamma]\!] \overset{\omega}{\vdash} \mathsf{M} : \mathcal{S}^{\diamond}$.*

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 $\mathtt{M}$. 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.

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 §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.

5.2.1 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:

$$\begin{aligned}
&\mathtt{sig}\ \mathtt{module}\ X : (\mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end})\ \ \mathtt{end} \\
&\qquad \approx\ \langle B : \emptyset \cdot \mathbb{Z} \cdot\ \mathtt{type}\ t_0 = \mathbb{Z}.X.t \rangle\ \mathtt{sig}\ \mathtt{module}\ X : (\mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end})\ \ \mathtt{end} \\
&\qquad \approx\ \langle B : \mathtt{type}\ t_0 = B.t_0 \cdot \mathbb{Z} \cdot \emptyset \rangle\ \mathtt{sig}\ \mathtt{module}\ X : (\mathtt{sig}_A\ \mathtt{type}\ t = B.t_0\ \mathtt{end})\ \ \mathtt{end}
\end{aligned}$$

Similarly, floating fields can be used to encode extrusion of higher-order types, by introducing unary functors that take an empty argument as input:

$$\begin{aligned}
&(Y : \mathsf{S}) \rightarrow \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{val}\ x : A.t\ \mathtt{end} \\
&\qquad \approx\ \langle B : \emptyset \cdot \mathbb{Z} \cdot\ \mathtt{module}\ F_0 : (Y : \mathtt{sig}\ \ \mathtt{end}) \rightarrow \mathtt{sig}_A\ \mathtt{type}\ t = \mathbb{Z}(Y).t\ \mathtt{end} \rangle \\
&\qquad\qquad\quad (Y : \mathsf{S}) \rightarrow \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{val}\ x : A.t\ \mathtt{end} \\
&\qquad \approx\ \langle B : \mathtt{module}\ F_0 : (Y : \mathtt{sig}\ \ \mathtt{end}) \rightarrow \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end} \cdot \mathbb{Z} \cdot \emptyset \rangle \\
&\qquad\qquad\quad (Y : \mathsf{S}) \rightarrow \mathtt{sig}_A\ \mathtt{type}\ t = B.F_0(Y).t\ \mathtt{val}\ x : A.t\ \mathtt{end}
\end{aligned}$$

This shows that floating fields have at least the same expressiveness as existentially quantified signatures with skolemization, except that they introduce names.

5.2.2 Alpha-conversion of floating components. Using again the generalized equivalence of §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:

$$\begin{aligned}
\langle A : \mathtt{type}\ t = \mathsf{u} \rangle\ \mathsf{S} \quad &\approx \quad \langle A : \mathtt{type}\ t = \mathsf{u} \cdot \mathbb{Z} \cdot\ \mathtt{type}\ t' = A.t \rangle\ \mathsf{S} \\
&\approx \quad \langle A : \mathtt{type}\ t' = \mathsf{u} \cdot \mathbb{Z} \cdot\ \mathtt{type}\ t = A.t' \rangle\ \mathsf{S}[t \mapsto t'] \\
&\approx \quad \langle A : \mathtt{type}\ t' = \mathsf{u} \rangle\ \mathsf{S}[t \mapsto t']
\end{aligned}$$

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 $\langle A : \mathtt{type}\ t_\alpha = A.t_\alpha \rangle\ \mathsf{S}$ where the source signature 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.

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 $\Gamma \vdash \mathsf{M} : \mathsf{S}$ and $\Gamma \vdash \mathsf{M} : \mathsf{S}'$ when S and S' syntactically differ—even a lot! as one may contain a signature definition expanded in the other.

Still, we should then have $\Gamma \vdash \mathsf{S}' \approx \mathsf{S}''$. 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.

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 [13], which easily triggers signature avoidance, as we have shown in §2 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 §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 [20, 26]. 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* [3, 18]. 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 [22], 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.

REFERENCES

[1] 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

[2] 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

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

[4] 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

[5] Jacques Guarrigue and Leo White. 2014. Type-level module aliases: independent and equal *(ML Family/OCaml Users and Developers workshops)*. https://www.math.nagoya-u.ac.jp/~garrigue/papers/modalias.pdf

[6] 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

[7] 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

[8] 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

[9] GÉRARD HUET. 1997. The Zipper. *Journal of Functional Programming* 7, 5 (1997), 549–554. https://doi.org/10.1017/S0956796897002864

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

[15] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: library operating systems for the cloud. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, Vivek Sarkar and Rastislav Bodík (Eds.). ACM, 461–472. https://doi.org/10.1145/2451116.2451167

[16] Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA.

[17] John C. Mitchell and Gordon D. Plotkin. 1985. Abstract Types Have Existential Types. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '85)*. Association for Computing Machinery, New York, NY, USA, 37–51. https://doi.org/10.1145/318593.318606

[18] Keiko Nakata and Jacques Garrigue. 2006. Recursive Modules for Programming. (2006), 13.

[19] Norman Ramsey. 2005. ML Module Mania: A Type-Safe, Separately Compiled, Extensible Interpreter. In *Proceedings of the ACM-SIGPLAN Workshop on ML, ML 2005, Tallinn, Estonia, September 29, 2005 (Electronic Notes in Theoretical Computer Science, Vol. 148)*, Nick Benton and Xavier Leroy (Eds.). Elsevier, 181–209. https://doi.org/10.1016/J.ENTCS.2005.11.045

[20] Andreas Rossberg. 1999. Undecidability of OCaml type checking. https://sympa.inria.fr/sympa/arc/caml-list/1999-07/msg00027.html.

[21] Andreas Rossberg. 2018. 1ML - Core and modules united. *J. Funct. Program.* 28 (2018), e22. https://doi.org/10.1017/S0956796818000205

[22] 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

[23] 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

[24] 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

[25] 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

[26] Leo White. 2015. Girard Paradox implemented in OCaml using abstract signatures.