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 signatures, and can be seen as a lightweight extension of existing signatures. Floating fields allow to delay the resolution of instances of the signature avoidance problem 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 fields without loss of type-sharing. We present a principled strategy that implements this criterion and performs much better than OCaml. Remaining floating fields 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, improving the quality of error messages. Besides, ZipML implements early and lazy strengthening, as well as lazy inlining of definitions, preventing duplication of signatures inside the typechecker. The correctness of the type system is proved by elaboration into M^{ω}, which has itself been proved sound by translation to F^{ω}. ZipML has been designed to be an improvement over OCaml that could be retrofitted into the existing implementation.

ACM Reference Format:
Clément Blaudeau, Didier Rémy, and Gabriel Radanne. 2024. Avoiding signature avoidance in ML modules with zippers. 1, 1 (October 2024), 30 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.[1] 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 [20]. 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`, hence the implementation details, are hidden. By using ascription, a developer can make sure that the data-structures are accessed only through the functions defined inside the module, before the ascription. In essence, this allows enforcing complex invariants, ranging from simple validity, for instance in a date library, to involved 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 break the

[1] Except for 1ML, which in principle unifies the two layers, although some stratification will persist in practice.

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

2024. ACM XXXX-XXXX/2024/10-ART
https://doi.org/

abstractions. In practice, an abstract type field `type t` introduces a name for a 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 [18] suggested to represent abstract types as *existential types*. This idea was extended by Russo [27] 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 were identified as having higher-order existential types by Biswas [1], and the extrusion mechanism was extended to support *skolemization* [2, 24, 27]. In this setting, syntactic, user-writable signatures are elaborated in the more expressive language of types of F^ω, the higher-order polymorphic lambda calculus and type-sharing between modules is expressed differently: abstract type fields actually introduce existential type *variables*, possibly higher-order, quantified in front of the signature, so that two modules that share an abstract type simply refer to the same type expression. This has culminated in the successful, so-called, *F-ing* line of works [2, 22–24, 27, 28] among others where most significant ML module features are specified and proved sound *by elaboration* in F^ω, thus benefiting from the meta-theoretical properties of the target language. This is a real benefit compared with purely syntactic systems for ML modules, which use somewhat nonstandard typing rules, whose meta-theory has to be redone from start, but also has proved hard to formalized, often requiring complex syntactic techniques or semantic objects [4, 6, 7].

Yet, ML modules system remained a case where advanced theoretical works had a limited impact on real-world implementations: neither OCaml nor SML compilers are based on an elaboration into F^ω. Notably, OCaml relies on a *path-based* system, initially described by Leroy [10, 11], which specification has not been extended to more complex constructs. The compiler has evolved to support new features and be more efficient, but maintaining an internal representation of signatures that is *more-or-less* syntactic. The SML compiler has an official specification [16, 17] and a subset of the language have been mechanically formalized using singleton types. The language has also evolved over the years, but not towards an elaboration in F^ω (see [14] for more details). Notably, Moscow ML, an implementation of the SML standard extended with applicative functors, recursive modules, and first-class modules [25–27], is an exception, as it uses F^ω-like types internally. To put it somewhat provocatively: *explainability*, *usability*, and *soundness* can be misaligned goals. While the *F-ing* approach fulfills the last goal, usability of industrial-grade *F-ing* based compiler remained to be demonstrated.[2]

Moreover, the *F-ing* approach creates a significant gap between the user writable *source* signatures and their internal representations in F^ω, which can undermine explainability. Requiring users to think in terms of the internal language while still writing types in the surface language imposes a mental gymnastics, as the elaboration between the surface and internal languages is quite involved. Another option would be to conceal the internal representation with a reverse translation, called *anchoring* in [2]. Unfortunately, it seems difficult to truly hide the internal representation: (1) Anchoring can fail because types of F^ω are more expressive than the source language signatures. Consequently, some inferred signatures cannot be expressed in the source signature syntax—and the program must then be rejected. This issue, called *signature avoidance*, discussed in more details below, is a serious problem in all syntactic approaches that has not found yet a good solution. Those cases trigger a specific class of typechecking errors that might be tricky for users to understand, as it might require exposing the internal representation of types. (2) The printing of a signature

[2] In particular, the *lazy inlining* of definitions, which seems essential for efficiency, has not been formalized nor implemented in Moscow ML and could be problematic.

```
1  module type Comparable = sig type t val eq : t → t → bool end
2  module type Keys = sig type t type k val getKey : t → k val fast_eq : k → k → bool end
```

```
3  module Map (E: Comparable)
4      (K: Keys with type t := E.t) = struct
5    type map = (K.k * (E.t * int)) list
6    let insert x n (l : map) =
7      let k = K.getKey x in ...
8        if (K.fast_equal k k') then ...
9        if (E.equal x x') then ...
10   let get x m = ...
11   let get_from_key k m = ...
12   (* ... *)
13 end
```

```
14 module Map (E: Comparable)
15     (K: Keys with type t := E.t) : sig
16   type map
17   val insert
18         : E.t → int → map → map
19   val get : E.t → map
20   val get_from_key
21         : K.k → map → (E.t * int) list
22   (* ... *)
23 end
```

Fig. 1. Example of OCaml code which could trigger signature avoidance. The left-hand side is the code while the right-hand side is the interface (typically, with the code in a `.ml` file and the interface in a `.mli` file). We use consecutive line numbers only for reference in the text.

for error messages might cause an unrelated signature avoidance error, which might be confusing for the user. (3) Even when the typechecking succeeds, the M^{ω} type-system [2] combined with anchoring is not *fully syntactic* in the sense of [28]: if a module expression admits a signature, not all sub-expressions necessarily do so. This might be counter-intuitive for the user, as simply exposing a sub-module might make typechecking fail in non-trivial ways. Overall, we believe that *F-ing* based systems cannot fully conceal their internal representation, which undermines explainability and might affect usability.

In the remainder of this section, starting with examples in OCaml, we discuss our design for a new syntactic module system, called ZipML. We address explainability by using syntactic signatures as internal representation—hence the *syntactic* adjective—which should match the user's intuitive understanding of syntax. We address usability by proposing a somewhat *conservative* extension of a path-based system, which *should not* interfere with other features or significantly affect performance trade-offs. We have not implemented ZipML, so this claim is not backed by experiments yet. Finally, we maintain *soundness* by translation in M^{ω} [2], leveraging the *F-ing* line of works.

Namely, (1) we explain, delay, and resolve the *signature avoidance* problem by zipping out-of-scope components; (2) we handle applicative functors and module identities; (3) we ensure a proper sharing of types, via the so-called *strengthening* operation [10], but with a new lazy and early strategy; (4) we maintain intermediate types and module type definitions, only lazily inlining them, so as to print concise interfaces respecting the user's intent—an aspect previously left behind in the whole *F-ing* line of works.

1.1 Modules, Abstraction, and Type Sharing

1.1.1 Introductory example. Let us start with a concrete example of modular code given in Figure 1. We want to build a library that provides parametric maps to integer. To that aim, we define the module-type `Comparable` (line 1) that represents the minimal signature that a module must satisfy for our library to build maps on: it must contain a type definition `type t` (left abstract for polymorphism), and an equality function `val eq : t -> t -> bool`. Then, we define `Map` (line 3) as a *functor* that creates a structure given a module parameter `E` that satisfies the interface `Comparable`. However, for performance reasons, we want to limit the number of equality tests. To do so, we define a module-type `Keys` (line 2) for modules that provide a type `k` of keys, a key generator `getKey`, and a fast equality test `fast_eq` for keys. Map then takes a second module parameter `K : Keys` as input (line 4), used to shortcut equality tests. Note that the generator `getKey` does not have to be injective: `Map` uses the default equality on elements when their keys are equal (lines 8 and 9). Internally, `Map` represents maps as lists of triples containing a key, an element, and its associated integer (line 5). Among the functions provided by `Map`, we have `get_from_key` (line 12) that returns the list of stored elements that have the same key. In the interface of `Map`, we force the type `map` to be abstract (line 16). It serves two purposes: first, it hides implementation details, especially the fact that we used lists to represent maps; second, it prevents users from inserting objects along with the wrong key, which would violate internal invariants of the module. A user can then instantiate `Map` with any comparable module along with a type of keys. Here, we create a datatype for keys:

```
module T = struct type t = Node of string * t * t | Leaf let eq = ... end
module K = struct type k = Root of string | Length of int ... end
module M = Map(T)(K)
```

However, this reveals the implementation details of `K`. One might hide them by writing, directly:

```
module M = Map(T)(struct type k = Root of string | Length of int ... end)
The parameter cannot be eliminated in the result type.
```

Unfortunately, this fails to typecheck, as the interface of `Map` mentions the type `K.k` (line 21), while there is no name to refer to it outside of its enclosing structure. This is a case of the *signature avoidance* problem. Naming `K` solves this case. However, forcing users to name every module would impact usability and make some code patterns impractical.

1.1.2 Signature Avoidance. At a more abstract level, the signature avoidance problem can occur whenever type declarations become inaccessible while dependencies on those declarations remain. We illustrated it above with a functor call on a structure, but for the sake of readability and conciseness, we now use module-level let-binding like **let** `X = M` **in** `M'` and projection out of an arbitrary module like `M.X`[3]. Neither one is present in OCaml which only allows projection out of paths to prevent cases prone to signature avoidance. However, functor calls on structures and generalized open statements can still trigger signature avoidance, so the core problem remains the same: given a signature S that depends on a type t, can we extract S and keep it well-formed even when t has become inaccessible?

```
module M = (... : sig type t module X : S end).X
```

There might not exist a principal signature `S'` that is equivalent to `S` while *avoiding* the type `t`. The key issue here is to assume that hiding the field **type** `t` necessarily means *deleting* it. Indeed, one could argue that hiding a field makes it inaccessible *to the user* but not necessarily to the typechecker. This is the stance taken by Harper and Stone [8] and Dreyer et al. [4]: their approach relies an elaboration with reserved names:

```
module M : sig module HIDDEN : sig type t end  module VISIBLE : S end
```

We share their intuition, but we found the technical device to achieve it too rigid: we would like to *focus* on `S` while keeping the rest of the fields *on the side* if needed. A classical technique in functional programming, *zippers* [9], allows precisely to extend any tree-like type definition with the ability to focus on certain parts of the tree while keeping the rest on the side. As signatures are

[3] Projection out of an arbitrary module expression and let-binding can each be encoded as syntactic sugar using the other: **let** `X = M` **in** `M'` can be encoded as `(`**struct module** `X = M` **module** `Y = M').Y` while `M.X` can be encoded as **let** `Y = M` **in** `Y.X`. Both can encode functor call on an arbitrary module: `F(M)` is **let** `X = M` **in** `F(X)` or `(`**struct module** `Arg= M` **module** `Res = F(Arg)` **end**`).Res`.

record trees, we can change the meaning of hiding to be the *zipping* of a signature onto the visible field, keeping the rest, called *floating fields*, in the *zipper context*. Here, ZipML would return:

```
module M : ⟨ type t ⟩ S
```

ZipML uses zippers to preserve relevant type information and therefore delay the resolution of signature avoidance until a source signature is forced by ascription. Zippers also behave gracefully in case of errors, allowing to present users with the field names that were lost during typechecking.

1.1.3 Applicativity, abstraction safety, and aliasing. Applicative functors, introduced by Leroy [11], 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 (Y : sig end) = struct end
module G (Y : 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, 24]). However, it is also fragile, as naming a subexpression can break syntactic equality:

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

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.[4]

1.1.4 Strengthening. In path-based systems, type sharing is expressed with manifest types [10] using explicit equalities between type fields. As a consequence, to avoid a loss of type sharing when copying or aliasing a module, its signature must be rewritten, hence copied, to refer to the type fields of the original module. This operation, called *strengthening*, is central to *path-based* systems such as OCaml. However, rewriting a whole signature can be costly[5] and hinder performance for very large libraries. To prevent useless rewriting while keeping type sharing, ZipML implements three innovations: strengthening is made lazy and early. *Laziness* is achieved also by using transparent signatures `(= P < S)` to mark the strengthening of `S` by `P` in the syntax tree, but delay the actual duplication of `S`. Besides, strengthening is used when signatures enters the typing environment (*earliness*) rather than when retrieving them, which makes for a simpler typing rule for accesses.

1.1.5 Lazy expansion. Module languages allow for both type definitions (`type t = int * float`) and module type definitions (`module type T = sig .. end`). However, while OCaml retains such definitions internally, as any real-word implementation, both M^{ω} and *F-ing* handle them by *eager inlining*. Inlining of module-type definitions in signatures might considerably increase their size and make typechecking of large-scale libraries with intensive use of modules quite expensive. It also loses the programmer's intent by inferring large signatures whose names or aliases have been lost. ZipML keeps module-type definitions internally and in inferred signatures, 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.

[4] In fact, transparent signatures were already expressible in *F-ing* [24]—see [2] for details.
[5] See the discussion at https://github.com/ocaml-flambda/flambda-backend/pull/1337

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 zippers.

Floating fields follow the way the user would resolve instances of 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. Besides, floating fields can be simplified and dropped after they became unreferenced. This mechanism is an internalized, improved counterpart of the anchoring of [2].

Our contributions are:

- The introduction of a syntactic type system for a fully-fledged OCaml-like module language, including both generative and applicative functors, and extended with transparent signatures.
- A new concept of *floating fields*, implemented as *zippers*, that enables to internalize and avoid (or delay) signature avoidance resolution.
- 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 systematic, lazy and early 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 zippers. In §3, we present the syntax and typing rules of ZipML. In §4, we define an equivalence on signatures that allows for the *simplification* of floating fields. We present an algorithm that computes such simplification. 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 delay and resolve instances of the signature avoidance problem. Floating fields provide additional expressiveness that allows to describe all inferred signatures. We use OCaml-like syntax [12] for examples. However, we use self-references for signatures[6] 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. Self-references are just a convenient syntactic notation that does not bring additional expressiveness, nor any cyclic references, but avoids some forms of shadowing and simplifies the definition of strengthening.

[6] ZipML also uses self-references in structures, but we do not in examples to keep closer to OCaml syntax.

We show examples with a module-level let-binding. We also use **type** t to introduce abstract types directly in a structure (a feature present in OCaml), but more realistic examples would use algebraic data-types[7] or ascriptions. We start with a simple example:

```
let module M = struct type t module X = struct let l : t list = [] end end in M.X
  The value "l" has no valid type if "X" is hidden.
```

Instead, ZipML will return[8] the following *zipper* signature:

```
⟨ B: type t = B.t ⟩ sig(A) val l : B.t list end
```

The highlighted expression is the zipper-context. It contains a single *floating field* **type** t = B.t, bound to the self-reference B. Intuitively, the zipper signature is obtained as follows. Before projection, the signature of M is:

```
sig(B) type t = B.t module X : sig(A) val l : B.t list end end
```

When projecting on field X, we would like to return **sig**(A) **val** l : B.t list **end**, but the reference B.t would become dangling. The solution is to keep the type of t defined as a floating field B.t, which gives exactly the signature just above.

Another typical situation of signature avoidance is when a type is used in other type definitions:

```
(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-abstraction *)
```

Here, instead of failing, OCaml silently turns u and v into abstract types, not only losing their list structure but also forgetting that these are actually equal types. We describe this as being erroneously resolved by *over-abstraction*. While abstraction is *safe*, as it could be achieved through an ascription, it is *incomplete* and therefore should be performed only when explicitly required by the user. In ZipML, we would return the following signature:

```
⟨ B: type t = B.t ⟩ sig(A) type u = B.t list type v = B.t list end
```

This is a correct answer, as no type information has been lost.

We now consider a module M with some nested submodules X and Y. We then project on a deeply nested module M.X.Y.

```
let M = struct type t
  module X = struct type u = t list
    module Y = struct type v = t type w = u end end
end in M.X.Y
sig(C) type v type w end
```

In ZipML, we first return the following signature S_0 with floating fields:

```
⟨ A: type t = A.t end ⟩ ⟨ B: type u = A.t list ⟩                  (* S0 *)
  sig(C) type v = A.t type w = B.u end
```

which we can then simplify to its equivalent final form S_1:

```
sig(C) type v type w = C.v list end                              (* S1 *)
```

[7]From a module level point of view, a declaration of an ADT **type** t = A **of** int | B **of** bool can be thought of as an abstract type **type** t followed by value bindings **val** A:int -> t and **val** B: bool -> t.

[8]By default, input programs are colored in blue; errors in red; output signatures in green with floating fields in yellow. We may still temporarily use other colors to emphasize specific subexpressions.

We now explain both steps, projection and simplification, separately.

2.1 Chaining zippers

To understand how S_0 was generated, let us look at the signature of `M`, before the projection:

```
M : sig(A) type t = A.t
  module X : sig(B) type u = A.t list
    module Y : sig(C) type v = A.t type w = B.u end end
end
```

It is of the form $\mathsf{S}_A\left[\mathsf{S}_B\left[\mathsf{S}_C\right]\right]$, where the signature S_C is placed in the context $\mathsf{S}_A\left[\mathsf{S}_B\left[\cdot\right]\right]$ and S_A and S_B are the outer and inner contexts. Unfortunately, the signature S_C is ill-formed outside of those contexts. Let us detail the process. For the first projection `M.X`, we turn the outer signature into a zipper context, which gives $\left\langle \mathsf{S}_A \right\rangle \mathsf{S}_B[\,\mathsf{S}_C\,]$. For the second projection `(M.X).Y`, we need to project out of a zipper signature, which is not a structural signature. However, it actually composes well: any operation on a zipped signature correspond to pushing the zipper context in the environment, doing the operation and popping the zipper context back. For our projection, we therefore first push the zipper context in the typing environment, leaving us with $\mathsf{S}_B[\,\mathsf{S}_C\,]$. Projecting gives $\left\langle \mathsf{S}_B \right\rangle \mathsf{S}_C$, and popping the zipper context back again gives: $\left\langle \mathsf{S}_A \right\rangle (\left\langle \mathsf{S}_B \right\rangle \mathsf{S}_C)$. Finally, we merge the two zipper contexts into a single one and obtain $\left\langle \mathsf{S}_A ; \mathsf{S}_B \right\rangle \mathsf{S}_C$, which is exactly S_0. Conceptually, we may can sum up those steps as:

$$(\mathsf{S}_A[\mathsf{S}_B[\mathsf{S}_C]].X).Y \rightsquigarrow (\left\langle \mathsf{S}_A \right\rangle \mathsf{S}_B[\mathsf{S}_C]).Y \rightsquigarrow \left\langle \mathsf{S}_A \right\rangle (\mathsf{S}_B[\mathsf{S}_C].Y) \rightsquigarrow \left\langle \mathsf{S}_A \right\rangle (\left\langle \mathsf{S}_B \right\rangle \mathsf{S}_C) \rightsquigarrow \left\langle \mathsf{S}_A ; \mathsf{S}_B \right\rangle \mathsf{S}_C$$

2.2 Simplifying zippers

We now explain the simplification process from S_0 to S_1. Interestingly, each component of the zipper contexts $\langle A : \mathsf{S}_A \rangle$ and $\langle B : \mathsf{S}_B \rangle$ can be directly accessed from S_C via its self-reference name, respectively A and B. Hence, S_C need not be renamed—provided we have chosen disjoint self-references while zipping. We may first inline the definition of `B.u` in the field `w` of S_C, leading to the signature:

```
⟨ A: type t = A.t ⟩ ⟨ B: type u = A.t list ⟩
    sig(C) type v = A.t type w = A.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 `A.t`, the field `C.w` can be rewritten as `C.v list`. We obtain the signature S_3:

```
⟨ A: type t = A.t ⟩ sig(C) type v = A.t type w = A.t list end        (* S3 *)
```

The signature could also be seen as the projection of the unzipped signature (using some reserved field $\mathbb{Z}$ for the lost[9] projection path):

```
(sig(A) type t = A.t module ℤ : sig(C) type v = A.t type w = C.v list end end).ℤ
```

Currently `C.v` is an alias to the floating abstract type `A.t`, which comes first. However, since the module `X` will become a floating field, absent at runtime, it may be moved after field $\mathbb{Z}$, letting `C.v` become the defining occurrence for the abstract type and `A.t` be an alias to `C.v`. The key is that the two unzipped signatures, before their projection, are subtype of one another, hence equivalent.

[9] Our zippers are *partial*, since we dropped both the name of the field we projected on and the fields following the projection. We could have used *full* zippers to keep all the information necessary to recover the original signature, but this is actually never needed, as we may always bake another signature using a reserved field $\mathbb{Z}$ and ignore the following fields, as above.

```
(sig(A) ℤ : sig(C) type v type w = C.v end  type t = A.t end).ℤ
```

We may now project back the signature on field ℤ. It no longer depends on fields of the following submodule X, which can safely be dropped. This leads to 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                                    (* S₁ *)
```

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 types v and w.

In general, simplification may remove some, but not necessarily all, floating fields. This is acceptable, as we are able to pursue typechecking in presence of floating fields, which may 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 A.t is referenced in $S_B\,[S_C]$ as A.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, likely causing a failure later as a consequencenof over-abstraction.

Interestingly, when typechecking an ascription (M : S), the signature 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 Overview

An overview of the type system is given in Figure 2. The core of the system is composed of three judgments: **module typing** (§3.7), **signature typing** (§3.6) and **subtyping** (§3.5). Module typing $\Gamma \vdash \mathsf{M} : \mathsf{S}$ is the main judgment: given a module expression M, it infers its signature S. Since module expressions may contain signatures, module typing relies on signature typing (1), which is used to check them[10]. We use subtyping to check ascription in module expressions (2) and transparent signatures (3). Since paths appear in signatures, we extract **path typing** (§3.4.2) from module typing to avoid all typing judgments to be recursively defined. All judgments handle paths and therefore depend on path typing (dashed blue arrows). Path typing also depends on subtyping (4) since paths include functor applications to cope for applicative functors.[11] As ZipML maintains user-written definitions of core-language types and module types, we need a notion of normalization (dotted gray arrows). Finally, we use helper judgments to handle lazy and shallow **strengthening** (§3.3) and early-strengthening **environment extension**. After presenting the grammar and some technical choices in §3.2, we detail the judgments in reverse order of dependencies (from right to left).

3.2 Grammar

The syntax of ZipML is given in Figure 3. It reuses the syntax of OCaml, but with a few differences in notation, the addition of transparent ascriptions, and our key contribution: signatures with

[10] Signature typing is also used as a meta-theoretic well-formedness of signatures, which we maintain throughout the system

[11] Technically, this comes from the fact that we use path typing both as a *path-well-formedness check* and as a *path-lookup*. We could have two separate judgments: the former would depend on subtyping and be quite costly, while the latter, only used on known well-formed paths, would not recheck subtyping and would be faster.

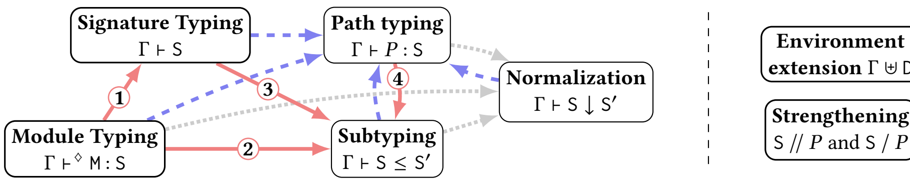

Fig. 2. Relationship between the main judgments of ZipML

Path and Prefix

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

Module Expression

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

Binding

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

Identifier

$$I ::= x \mid t \mid X \mid T$$

Typing environment

$$\Gamma ::= \varnothing \mid \Gamma, A.\mathsf{D} \mid \Gamma, Y : \mathsf{S}$$

Zipper context

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

Signature

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

Declaration

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

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. 3. Syntax of ZipML

floating fields, also called *zippers*. As meta-syntactic conventions, we use lowercase letters ($x, t, \ldots$) for elements of the core language, and uppercase letters ($X, T, \ldots$) for modules. We also use slanted letters ($I, A, Q, \ldots$) for identifiers and paths, and upright letters ($\mathsf{M}, \mathsf{e}, \mathsf{D}, \ldots$) for syntactic categories. Finally, we designate lists with an overbar: $\overline{\mathsf{B}}$ is a list of B's. We detail these syntactic categories below.

Main module constructs. The two main categories are **Module Expressions** (M) and **Signatures** (S). We put the content of structures and structural signatures, (expressions, types, modules, module types) in the separate categories of **Bindings** (B) and **Declarations** (D). As in OCaml we use a special unit argument to syntactically distinguish generative functors from applicative functors. Both structures and signatures are annotated with a *self-reference* A, explained below. Signatures contain transparent signatures $(= P < \dot{\mathsf{S}})$ to express that a module has the identity of P but the interface of S. It partially subsumes type-level module aliases of OCaml.

Self-references. A special class of identifiers A range over self-references, which are variables used in both module structures $\texttt{struct}_A\ \overline{\mathsf{B}}$ end and structural signatures $\texttt{sig}_A\ \overline{\mathsf{D}}$ end to refer to the structure or the signature itself from fields $\overline{\mathsf{B}}$ or $\overline{\mathsf{D}}$. They are **not** used to define **recursive structures**, but only to refer to previously defined fields in a telescope. The subcript annotation $_A$ on a structure or a signature is a binding occurrence whose scope extends to $\overline{\mathsf{B}}$ or $\overline{\mathsf{D}}$ and that can be freely renamed. Thanks to self-references, field names are no longer binders and behave rather as record fields. We write $\mathrm{dom}(\mathsf{D})$ and $\mathrm{dom}(\mathsf{B})$ the field name I of D. We disallow field shadowing in signatures, which is standard in module systems. By contrast with common usage, we also disallow field shadowing in structures: all field names in the same structure must be disjoint. However, fields names do not shadow other fields of the same name in a substructure (or subsignature) as these are accessed through their self-references which can be renamed. This restriction of shadowing is for the sake of simplicity and is just a matter of name resolution that has little interaction with typechecking.

Self-references are also used to represent abstract types: all type fields are of the form $\texttt{type}\ t = \mathsf{u}$ 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. This is merely a syntactic trick to simplify the treatment of telescopes and strengthening. We may omit the self-reference and just write sig D end for $\texttt{sig}_A$ D end when A does not appear free in D. Conversely, when we write sig D end, we should read $\texttt{sig}_A$ D end where the anonymous self-reference A is chosen fresh for D.

Identifiers and Paths. Paths P are the mechanism to access modules, statically. By static, we mean that we always know statically the identity of the module a path refers to. Paths may access the environment directly, either through a functor parameter Y or a field $A.I$, 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 $P(P)$. Floating fields are accessed with $P.A$. The letter Q designates a distant access via a path P or a local access via a self-reference A. Finally, we extend the core language with qualified values $Q.x$ and qualified 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 substituted during typechecking. By contrast, names are never used in a binding position.

Typing environments. Typing environments Γ bind module fields to declarations and functor parameters to signatures. Module fields are always prefixed by a self-reference in typing environments. As we disallow shadowing, $A.\mathsf{D}$ can only be added to Γ if Γ does is not already contain some $A.\mathsf{D}'$ where D and D' define the same field. For convenience, we may write $\Gamma, A.\overline{\mathsf{D}}$ for the sequence $\Gamma, \overline{A.\mathsf{D}}$, where fields of D have been added one by one.

Zippers. Finally, the novelty is the introduction of *zippers* $\langle\gamma\rangle\ \mathsf{S}$ where γ is a *zipper context*, i.e., a sequence of floating fields $A : \overline{\mathsf{D}}$. The self-reference A, now used as a label to access fields of $\overline{\mathsf{D}}$ from S, **cannot be** freely **renamed** any longer[12]: the introduction of zippers turns an α-convertible self-reference into a fixed label. The concatenation of zippers $\gamma_1 ; \gamma_2$ is only defined when the domains, i.e., the set of self-references, of γ_1 and γ_2 are disjoint.

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 $\langle\emptyset\rangle\ \mathsf{S}$ with S and $\langle\gamma_1\rangle\ \langle\gamma_2\rangle\ \mathsf{S}$ with $\langle\gamma_1 ; \gamma_2\rangle\ \mathsf{S}$ whenever γ_1 and γ_2 have disjoint domains.

[12] In section §4, we will allow for consistent renaming under certain conditions.

Therefore, a signature S can always be written as $\langle\gamma\rangle\, \dot{\mathsf{S}}$ where $\dot{\mathsf{S}}$ is a plain signature and γ concatenates all consecutive zipper contexts or is $\emptyset$ if there were none. The introduction of zippers requires a new form of path $P.A$, invalid in source programs, to access floating fields.[13]

Zipper-free signatures. We define plain signatures $\dot{\mathsf{S}}$ as those without an initial zipper (but where subterms may contain zippers) and source signatures $\ddot{\mathsf{S}}$ as those that do not contain any zipper at all, which are used in source types appearing in source expressions, i.e., in a transparent ascription or the domain of a functor. Similarly, we let $\ddot{P}$ stand for *source* paths P that do not contain any (direct or recursive) access to some zipper context (of the form $P'.A$). Consistently, $\ddot{Q}$ means $\ddot{P}$ or A and *source* types $\ddot{\mathsf{u}}$ means types u where all paths occurring in u are source paths.

Invariants. We also define several syntactic subcategories of signatures to capture some invariants.[14] The head value form S^{v} of a signature gives the actual shape of a signature, which is either a structural signature or a functor. The head normal form S^{n} is similar, but still contains the *identity* of a signature, if it has one, via a transparent ascription.

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

Notice that head normal forms (and value forms) are superficial and a signature appearing under a value form may itself be any signature and therefore contain inner zippers. A *Transparent* signature $\mathbb{S}$ is a generalization of the syntactic form $(= P < \dot{\mathsf{S}})$ that also allows zippers provided their signatures eventually start with transparent signatures. The definition is the following:

$$\mathbb{S} \;::=\; \langle\gamma\rangle\, \mathbb{S} \mid \dot{\mathbb{S}} \qquad\qquad \dot{\mathbb{S}} \;::=\; (= P < \dot{\mathsf{S}}) \qquad\qquad \gamma \;::=\; A : \overline{\mathbb{D}} \mid \gamma\,;\gamma \mid \emptyset$$
$$\mathbb{D} \;::=\; \mathtt{module}\, X : \mathbb{S} \mid \mathtt{val}\, x : \mathsf{u} \mid \mathtt{module\ type}\ T = \ddot{\mathsf{S}} \mid \mathtt{type}\, t = \mathsf{u}$$

Syntactic choices and syntactic sugar. Our grammar has some superficial syntactic restrictions that simplify 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 restricts 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, we restricted ascriptions to source paths, $(\ddot{P} : \ddot{\mathsf{S}})$, but the general case can be encoded as $(\mathtt{struct}_A\ \mathtt{module}\ X_1 = \mathsf{M}\ \mathtt{module}\ X = (A.X_1 : \ddot{\mathsf{S}})\ \mathtt{end}).X$. Finally, 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.

3.3 Strengthening

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. ZipML reuses transparent ascription $(= P < \mathsf{S})$ to express strengthening of the signature S by the path P, which effectively gives an identity, i.e., the path P, to an existing signature S—under a subtyping condition between the signature of P and 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.[15]

[13]In the absence of floating fields, self references could only be at the origin of a path Q.
[14]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.
[15]There is actually a proposal to add lazy strengthening in OCaml for efficiency purposes, see https://github.com/ocaml-flambda/flambda-backend/pull/1337

Delayed strengthening (signatures)

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

Shallow strengthening

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

Delayed strengthening (declarations)

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

Environment strengthening

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

Fig. 4. Strengthening (delayed by default) – $\mathtt{S} \,/\, P$ and $\mathtt{S} \mathbin{//} P$ and $\mathtt{D} \mathbin{//} Q$

Given a signature $\mathtt{S}$ and a path P, we consider two forms of strengthening: delayed strengthening $\mathtt{S} \mathbin{//} P$ and shallow strengthening $\mathtt{S} \,/\, P$, defined in Figure 4. Shallow strengthening is only defined on signatures in head normal form and is called during normalization to push strengthening just one level down. It then delegates the work to delayed strengthening $\mathtt{S} \mathbin{//} P$, which will insert a transparent ascription in a signature, if there is not one already, and push it under zippers if any. In particular, the shallow strengthening of a structural signature is a structural signature that does not use its self-reference any longer. Since the very purpose of strengthening is to make signatures transparent, both forms 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. We decompose compound zippers as two successive zippers. For a simple zipper, we strengthen both the zipper context and the underlying signature in which we replaced the self-reference A by the strengthened path. We ignore the empty zipper, which is neutral for zipping.
- Otherwise, delayed strengthening just inserts a transparent ascription, which is actually the materialization of the delaying.

We also defined delayed strengthening on declarations $\mathtt{D} \mathbin{//} Q$, which is called by strengthening on zipper contexts, shallow strengthening, and strengthening of the typing context: delayed strengthening is pushed inside module declarations and dropped on other fields. (Module type definitions never have an identity, so they cannot be strengthened.) Definition strengthening may be called on a zipper self-variable A, which will them immediately expand to a recursive call $S \mathbin{//} A.X$, hence on a path $A.X$. We also define a binary operator $\uplus$ that strengthens bindings as they enter the typing environment. We use it to maintain the invariant that all signatures entering the typing environment are transparent signatures $\mathbb{S}$, so that they can be duplicated without loss of sharing.

3.4 Path typing and normalization

Since paths include projections and applications, which require recursive lookups and substitutions, their types are not immediate to deduce. Moreover, signatures in the typing environment may themselves be module type definitions that must be inlined to be analyzed. We use path resolution, path typing, and normalization for this purpose.

$$\text{RES-P-ID}\ \frac{\Gamma \vdash P : \langle\gamma\rangle\,(= P' < \dot{\mathsf{S}})}{\Gamma \vdash P \triangleright P'} \qquad \text{RES-P-VAL}\ \frac{\Gamma \vdash P : \langle\gamma\rangle\,(= P' < \dot{\mathsf{S}}) \qquad \dot{\mathsf{S}} \,/\, P' = \mathsf{S}'}{\Gamma \vdash P \triangleright \mathsf{S}'}$$

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

$$\text{TYP-P-ARG}\ \frac{Y : \mathbb{S} \in \Gamma}{\Gamma \vdash Y : \mathbb{S}} \qquad \text{TYP-P-MODULE}\ \frac{A.\mathtt{module}\,X : \mathbb{S} \in \Gamma}{\Gamma \vdash A.X : \mathbb{S}} \qquad \text{TYP-P-ZIP}\ \frac{\Gamma \vdash P : \langle\gamma\rangle\,\mathbb{S} \qquad \gamma(A) = \overline{\mathbb{D}}}{\Gamma \vdash P.A : \mathtt{sig}\ \overline{\mathbb{D}}\ \mathtt{end}} \qquad \text{TYP-P-NORM}\ \frac{\Gamma \vdash P : \mathbb{S} \qquad \Gamma \vdash \mathbb{S} \downarrow \mathbb{S}'}{\Gamma \vdash P : \mathbb{S}'}$$

$$\text{TYP-P-PROJ}\ \frac{\Gamma \vdash P \triangleright \mathtt{sig}\ \overline{\mathbb{D}}\ \mathtt{end} \qquad \mathtt{module}\,X : \mathbb{S} \in \overline{\mathbb{D}}}{\Gamma \vdash P.X : \mathbb{S}} \qquad \text{TYP-P-APPA}\ \frac{\Gamma \vdash P \triangleright (Y : \ddot{\mathbb{S}}_0) \to \mathbb{S} \qquad \Gamma \vdash P' : \mathbb{S}' \qquad \Gamma \vdash \mathbb{S}' \le \ddot{\mathbb{S}}_0}{\Gamma \vdash P(P') : \mathbb{S}\big[Y \leftarrow P'\big]}$$

Fig. 6. Path typing – $\Gamma \vdash P : \mathbb{S}$

Path typing and path resolution. Intuitively, typing a path $\Gamma \vdash P : \mathbb{S}$ returns *all* the environment information about the module at path P, including a potential zipper, an aliasing information with another path (or itself), and finally the signature. To factor out a common pattern-match on the result of path typing, we also introduce *path resolution* to extract *the content* of the module $\Gamma \vdash P \triangleright \mathsf{S}$, by dropping the zipper and aliasing information and forcing a shallow strengthening, or the identity of the module $\Gamma \vdash P \triangleright P'$, by dropping the zipper and signature.

3.4.1 $\boxed{\Gamma \vdash P \triangleright P'}$ *and* $\boxed{\Gamma \vdash P \triangleright \mathsf{S}}$ *– Path resolution.* The two judgments are defined in Figure 5 by a single rule each. Rule RES-P-ID simply pattern-matches on the result of signature typing to return the aliasing information. Rule RES-P-VAL also pattern-matches on the result of signature typing both to extract $\dot{\mathsf{S}}$ but also to force it to be in head-normal (so that $\dot{\mathsf{S}} \,/\, P'$ is well-defined).

3.4.2 $\boxed{\Gamma \vdash P : \mathbb{S}}$ *– Path typing.* The judgment is defined in Figure 6. Rules TYP-P-ARG and TYP-P-MODULE are straightforward lookups. Rule TYP-P-ZIP accesses a zipper context through its self-reference. Notice that the signature $\mathtt{sig}_A\ \overline{\mathbb{D}}\ \mathtt{end}$ of $P.A$ need not be put back inside the zipper context γ, since the signature $\langle\gamma\rangle\,\mathbb{S}$, and hence the declarations $\overline{\mathbb{D}}$, are transparent and no longer depend on the zipper context γ, but only on the environment Γ. Rule TYP-P-NORM means that path typing is defined up to signature normalization, which allows the inlining of module type definitions. This makes path typing non-deterministic, purposedly. The two remaining rules use path resolution to pattern-match on the content of a signature. In Rule TYP-P-PROJ, we omitted the self-reference of the signature of P since we know P 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.[16]. We then return the codomain signature after substitution of the argument P for the parameter Y.

The dependency between path typing and subtyping (the premise $\Gamma \vdash \mathbb{S}' \le \ddot{\mathbb{S}}_0$ in TYP-P-APPA) is somewhat artificial. It comes from the fact that a single judgment is used for both *path lookups* and *path checking*. We could have (and an efficient implementation *would*) used two separate judgments, where path-lookup, only used valid paths, would not recheck subtyping.

3.4.3 $\boxed{\Gamma \vdash \mathsf{S} \downarrow \mathsf{S}'}$ *– Normalization.* 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 zipper. We never normalize the zipper context itself, as we will first access the

[16]The two first premises ensure that signatures $\mathbb{S}'$ and S are well-typed as required when using the subtyping judgment.

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

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

Fig. 7. Signature and type normalization – $\Gamma \vdash \mathsf{S} \downarrow \mathsf{S}'$

$$\textsc{Sub-S-Sig}\ \frac{\overline{\mathsf{D}}_0 \sqsubset \overline{\mathsf{D}}_1 \qquad \Gamma \uplus A : \overline{\mathsf{D}}_1 \vdash \overline{\mathsf{D}}_0 \,/\!/\, A \preccurlyeq \overline{\mathsf{D}}_2}{\Gamma \vdash \texttt{sig}_A\ \overline{\mathsf{D}}_1\ \texttt{end} \preccurlyeq \texttt{sig}_A\ \overline{\mathsf{D}}_2\ \texttt{end}} \qquad \textsc{Sub-S-Norm}\ \frac{\Gamma \vdash \mathsf{S}_1 \downarrow \mathsf{S}_1' \qquad \Gamma \vdash \mathsf{S}_2 \downarrow \mathsf{S}_2' \qquad \Gamma \vdash \mathsf{S}_1' \preccurlyeq \mathsf{S}_2'}{\Gamma \vdash \mathsf{S}_1 \preccurlyeq \mathsf{S}_2}$$

$$\textsc{Sub-S-ZipperL}\ \frac{\Gamma \uplus \gamma \vdash \mathsf{S}_1 \preccurlyeq \mathsf{S}_2}{\Gamma \vdash \langle\gamma\rangle\, \mathsf{S}_1 \preccurlyeq \mathsf{S}_2} \qquad \textsc{Sub-S-TrAscr}\ \frac{\Gamma \vdash \mathsf{S}_1 \,/\, P \preccurlyeq \mathsf{S}_2 \,/\, P}{\Gamma \vdash (= P < \mathsf{S}_1) \preccurlyeq (= P < \mathsf{S}_2)} \qquad \textsc{Sub-S-LooseAlias}\ \frac{\Gamma \vdash \mathsf{S}_1 \,/\, P \preccurlyeq \mathsf{S}_2}{\Gamma \vdash (= P < \mathsf{S}_1) \preccurlyeq \mathsf{S}_2} \qquad \textsc{Sub-S-FctG}\ \frac{\Gamma \vdash \mathsf{S}_1 \preccurlyeq \mathsf{S}_2}{\Gamma \vdash () \to \mathsf{S}_1 \preccurlyeq () \to \mathsf{S}_2}$$

$$\textsc{Sub-S-FctA}\ \frac{\Gamma \vdash \ddot{\mathsf{S}}_2 \preccurlyeq \ddot{\mathsf{S}}_1 \qquad \Gamma \uplus Y : \ddot{\mathsf{S}}_2 \vdash \mathsf{S}_1' \preccurlyeq \mathsf{S}_2'}{\Gamma \vdash (Y : \ddot{\mathsf{S}}_1) \to \mathsf{S}_1' \preccurlyeq (Y : \ddot{\mathsf{S}}_2) \to \mathsf{S}_2'} \qquad \textsc{Sub-T-Norm}\ \frac{\Gamma \vdash \mathsf{u}_1 \downarrow \mathsf{u} \qquad \Gamma \vdash \mathsf{u}_2 \downarrow \mathsf{u}}{\Gamma \vdash \mathsf{u}_1 \preccurlyeq \mathsf{u}_2} \qquad \textsc{Sub-D-Val}\ \frac{\Gamma \vdash \mathsf{u}_1 \preccurlyeq \mathsf{u}_2}{\Gamma \vdash \texttt{val}\ x : \mathsf{u}_1 \preccurlyeq \texttt{val}\ x : \mathsf{u}_2}$$

$$\textsc{Sub-D-Mod}\ \frac{\Gamma \vdash \mathsf{S}_1 \preccurlyeq \mathsf{S}_2}{\Gamma \vdash \texttt{module}\ X : \mathsf{S}_1 \preccurlyeq \texttt{module}\ X : \mathsf{S}_2} \qquad \textsc{Sub-D-ModType}\ \frac{\Gamma \vdash \ddot{\mathsf{S}}_1 \approx \ddot{\mathsf{S}}_2}{\Gamma \vdash \texttt{module type}\ T = \ddot{\mathsf{S}}_1 \preccurlyeq \texttt{module type}\ T = \ddot{\mathsf{S}}_2} \qquad \textsc{Sub-D-Type}\ \frac{\Gamma \vdash \mathsf{u}_1 \approx \mathsf{u}_2}{\Gamma \vdash \texttt{type}\ t = \mathsf{u}_1 \preccurlyeq \texttt{type}\ t = \mathsf{u}_2}$$

Fig. 8. Code-free subtyping $\Gamma \vdash \mathsf{S}_1 \lesssim \mathsf{S}_2$ and coercion subtyping $\Gamma \vdash \mathsf{S}_1 \leq \mathsf{S}_2$

zipper context and normalize the result afterwards. Rules Norm-S-Trans-Some and Norm-S-Trans-None allows normalization under a transparent ascription. If the head 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. We also define $\Gamma \vdash P.t \downarrow \mathsf{u}$, the 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 $Q.t$.

3.5 Subtyping judgments $\boxed{\Gamma \vdash \mathsf{S}_1 \preccurlyeq \mathsf{S}_2}$

We define two subtyping judgments, *code-free* subtyping $\Gamma \vdash \mathsf{S} \lesssim \dot{\mathsf{S}}$ and *coercion* subtyping $\Gamma \vdash \mathsf{S} \leq \dot{\mathsf{S}}$. The latter can only be used at type ascription and functor applications, as it requires changing the representation of the underlying values.

To factor both definitions, we defined a set of subtyping rules $\mathcal{R}^{\sqsubset, \preccurlyeq}_{\preccurlyeq}$ parameterized by three relations in Figure 8. The key rule, which is also the main difference between $\leq$ and $\lesssim$ is Sub-S-Sig. It uses the binary relation $\sqsubset$ to tell which components can be dropped or reordered when subtyping between two structural signatures $\texttt{sig}_A\ \overline{\mathsf{D}}_1\ \texttt{end}$ and $\texttt{sig}_A\ \overline{\mathsf{D}}_2\ \texttt{end}$. Namely, we must find a sequence $\overline{\mathsf{D}}_0$ related to $\overline{\mathsf{D}}_1$ by $\sqsubset$ that is, after strengthening by A (to keep all sharing), in pointwise $\preccurlyeq$-subtyping relation with $\overline{\mathsf{D}}_2$ in an environment extended with $A.\overline{\mathsf{D}}_1$. We use two versions of $\sqsubset$: for coercion subtyping, we take $\subseteq$, i.e., $\overline{\mathsf{D}}_0$ can be any subset of $\overline{\mathsf{D}}_1$ where fields may appear in a different order; for code-free subtyping, we use the relation $\subsetapprox$ that is the subrelation of $\subseteq$ that preserves dynamic fields (modules and values) and their order. Formally, $\overline{\mathsf{D}}_0 \subsetapprox \overline{\mathsf{D}}_1$ means both $\overline{\mathsf{D}}_0 \subseteq \overline{\mathsf{D}}_1$ and $\mathsf{dyn}(\overline{\mathsf{D}}_0) = \mathsf{dyn}(\overline{\mathsf{D}}_1)$ where $\mathsf{dyn}(\overline{\mathsf{D}})$ returns the subsequence of $\overline{\mathsf{D}}$ composed of dynamic fields only.

The subtyping relations are defined in two steps: we first defined code-free subtyping $\lesssim$ as the smallest relation that satisfies the rules $\mathcal{R}_{\lesssim}^{\underset{\sim}{\sqsubseteq}, \lesssim}$. Once $\lesssim$ is defined and fixed, we then define coercion subtyping $\leq$ as the smallest relation that satisfies the rules $\mathcal{R}_{\leq}^{\subseteq, \lesssim}$.

Both subtyping judgments are of the form $\Gamma \vdash \mathsf{S}_1 \preccurlyeq \mathsf{S}_2$ and only defined when S_2 is a zipper-free signature $\ddot{\mathsf{S}}_2$ (we relax this in §4). The judgment should (and will) only be used when both signatures are well-typed. Typically, the right-hand side signature is a source signature while the left-hand side signature S results from the typing of either a source signature or a module expression—and may contain zippers. The same invariants extend to auxiliary subtyping judgments for declarations and paths. Type equivalence $\approx$ is thus only defined between zipper-free signatures.

When reading the subtyping rules, it may help to think of coercion subtyping first, i.e., reading of $\preccurlyeq$ as $\leq$ and $\precsim$ as $\lesssim$. The subtyping rules for signatures can be read by case analysis on the left-hand-side signature, except for Rule Sub-S-Norm, which 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. Since a zipper may only occur on the left-hand side, Rule Sub-S-ZipperL just pushes the zipper context in the typing environment and pursues with subtyping. For other cases, we require the left-hand side to be in head normal form, which can be achieved by Rule Sub-S-Norm. 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 from the left-hand side, which amounts to a loss of 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 (rules Sub-S-FctG and Sub-S-FctA). Finally, Rule Sub-S-DynEq is just used to define $\approx$ on signatures as the kernel of $\preccurlyeq$.

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 for the normalization of core-language types. 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 code-free subtyping in both directions, i.e., type equivalence, which is well-defined since module type definitions are source signatures.

Subtyping and well-typedness. Subtyping is only meant to be well-behaved on well-typed signatures: it is the caller's responsibility to ensure that both signatures are well-formed.

Subtyping and inlining. The premises of rules Sub-D-Modtype and Sub-D-Type, require code-free equivalence between the definitions. This is because names are not always inlined, and therefore T and t may appear in both positive and negative positions later in the signature. If definitions were fully inlined, subtyping would never see the names of module-type definitions but only their original inlined expansion and covariance would suffice.

Optimization. Judgments $\Gamma \vdash \mathsf{S}_1 \ / \ P \leq \mathsf{S}_2$ and $\Gamma \vdash \mathsf{S}_1 \ / \ P \leq \mathsf{S}_2 \ / \ P$ are in fact equivalent. Intuitively, a derivation of the former may abstract some types appearing in $\mathsf{S}_1 \ / \ P$, but never has to, i.e., the same derivation could be reproduced without any abstraction. Therefore, Rule Sub-S-LooseAlias

$$\textsc{Typ-S-ModType}\ \frac{\Gamma \vdash Q.T : \mathsf{S}}{\Gamma \vdash Q.T} \qquad \textsc{Typ-S-GenFct}\ \frac{\Gamma \vdash \mathsf{S}}{\Gamma \vdash () \to \mathsf{S}} \qquad \textsc{Typ-S-AppFct}\ \frac{\Gamma \vdash \ddot{\mathsf{S}} \qquad \Gamma \uplus (Y : \ddot{\mathsf{S}}) \vdash \mathsf{S}'}{\Gamma \vdash (Y : \ddot{\mathsf{S}}) \to \mathsf{S}'}$$

$$\textsc{Typ-S-Ascr}\ \frac{\Gamma \vdash \mathsf{S} \qquad \Gamma \vdash P : \mathsf{S}' \qquad \Gamma \vdash \mathsf{S}' \preccurlyeq \mathsf{S}}{\Gamma \vdash (= P < \mathsf{S})}$$

$$\textsc{Typ-S-Str}\ \frac{\Gamma \vdash_A \overline{\mathsf{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{sig}_A\ \overline{\mathsf{D}}\ \mathtt{end}} \qquad \textsc{Typ-S-Zipper}\ \frac{\Gamma \vdash_A \overline{\mathsf{D}} \qquad \Gamma \uplus A : \overline{\mathsf{D}} \vdash \mathsf{S}}{\Gamma \vdash \langle A : \overline{\mathsf{D}} \rangle\ \mathsf{S}}$$

$$\textsc{Typ-D-Val}\ \frac{\Gamma \vdash \mathsf{u}}{\Gamma \vdash_A (\mathtt{val}\ x : \mathsf{u})} \qquad \textsc{Typ-D-Type}\ \frac{\Gamma \vdash \mathsf{u}}{\Gamma \vdash_A (\mathtt{type}\ t = \mathsf{u})} \qquad \textsc{Typ-D-Nil}\ \Gamma \vdash_A \varnothing$$

$$\textsc{Typ-D-TypeAbs}\ \Gamma \vdash_A (\mathtt{type}\ t = A.t) \qquad \textsc{Typ-D-Mod}\ \frac{\Gamma \vdash \mathsf{S}}{\Gamma \vdash_A (\mathtt{module}\ X : \mathsf{S})} \qquad \textsc{Typ-D-ModType}\ \frac{\Gamma \vdash \mathsf{S}}{\Gamma \vdash_A (\mathtt{module\ type}\ T = \mathsf{S})}$$

$$\textsc{Typ-D-Seq}\ \frac{\Gamma \vdash_A \mathsf{D}_0 \qquad \Gamma \uplus A.\mathsf{D}_0 \vdash_A \overline{\mathsf{D}}}{\Gamma \vdash_A (\mathsf{D}_0, \overline{\mathsf{D}})}$$

Fig. 9. Signature typing – $\Gamma \vdash \mathsf{S}$

could be replaced by rule Sub-S-LooseAlias-Opt:

$$\textsc{Sub-S-LooseAlias-Opt}\ \frac{\Gamma \vdash \mathsf{S}_1 \,/\, P \leq \mathsf{S}_2 \,/\, P}{\Gamma \vdash (= P < \mathsf{S}_1) \leq \mathsf{S}_2} \qquad \textsc{Sub-S-Sig-Opt}\ \frac{\overline{\mathsf{D}}_0 \subseteq \overline{\mathsf{D}}_1 \qquad \Gamma \vdash \overline{\mathsf{D}}_0 \leq \overline{\mathsf{D}}_2}{\Gamma \vdash \mathtt{sig}\ \overline{\mathsf{D}}_1\ \mathtt{end} \leq \mathtt{sig}\ \overline{\mathsf{D}}_2\ \mathtt{end}}$$

We may then add Rule Sub-S-Optim, which is an instance of Sub-S-Sig that can be used when neither side uses its self-reference avoiding pushing useless information in the typing environment.

3.6 Signature typing $\boxed{\Gamma \vdash \mathsf{S}}$

User-provided source signatures $\ddot{\mathsf{S}}$ are not necessarily well-formed and are checked using the judgment $\Gamma \vdash \mathsf{S}$, defined on Figure 9. We still defined and use the judgment on inferred signatures, which may contain zippers.

Rule Typ-S-ModType uses path typing to check the well-formedness of paths. Rule Typ-S-Ascr must also check that the signature S of path P is a subtype of the source signature S.[17] 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 that 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 replacing the typing judgment $\Gamma \vdash \mathsf{S}$ by an elaboration judgment[18].

3.7 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 in Figure 10. 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 stands 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.

When considering a source path $\ddot{P}$ as a module expression (Rule Typ-M-Path) we use path typing, which returns a transparent signature $\mathbb{S}$. However, we return $(= \ddot{P} < \mathsf{S})$, i.e., S with its most recent identity $\ddot{P}$, as this is probably the one the user would like to see and the older identities can always

[17] The two first premises ensure that signatures S and S' are well-typed, as required when using the subtyping judgment.
[18] This would also 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.

$$\textsc{Typ-M-Norm}\ \frac{\Gamma \vdash^{\diamond} \mathtt{M} : \mathtt{S} \qquad \Gamma \vdash \mathtt{S} \downarrow \mathtt{S}'}{\Gamma \vdash^{\diamond} \mathtt{M} : \mathtt{S}'} \qquad \textsc{Typ-M-Mode}\ \frac{\Gamma \vdash^{\triangledown} \mathtt{M} : \mathtt{S}}{\Gamma \vdash^{\blacktriangledown} \mathtt{M} : \mathtt{S}} \qquad \textsc{Typ-M-Path}\ \frac{\Gamma \vdash \ddot{P} : \langle \gamma \rangle\, (= P' < \mathtt{S})}{\Gamma \vdash^{\diamond} \ddot{P} : (= \ddot{P} < \mathtt{S})} \qquad \textsc{Typ-M-Ascr}\ \frac{\Gamma \vdash \ddot{\mathtt{S}} \qquad \Gamma \vdash \ddot{P} : \mathbb{S} \qquad \Gamma \vdash \mathbb{S} \leq \ddot{\mathtt{S}}}{\Gamma \vdash^{\diamond} (\ddot{P} : \ddot{\mathtt{S}}) : \ddot{\mathtt{S}}}$$

$$\textsc{Typ-M-FctA}\ \frac{\Gamma \vdash \ddot{\mathtt{S}} \qquad \Gamma \uplus Y : \ddot{\mathtt{S}} \vdash^{\triangledown} \mathtt{M} : \mathtt{S}'}{\Gamma \vdash^{\diamond} (Y : \ddot{\mathtt{S}}) \to \mathtt{M} : (Y : \ddot{\mathtt{S}}) \to \mathtt{S}'} \qquad \textsc{Typ-M-FctG}\ \frac{\Gamma \vdash^{\blacktriangledown} \mathtt{M} : \mathtt{S}}{\Gamma \vdash^{\diamond} () \to \mathtt{M} : () \to \mathtt{S}} \qquad \textsc{Typ-M-AppG}\ \frac{\Gamma \vdash P \triangleright () \to \mathtt{S}}{\Gamma \vdash^{\blacktriangledown} P() : \mathtt{S}} \qquad \textsc{Typ-M-Str}\ \frac{\Gamma \vdash_A \overline{\mathtt{B}} : \overline{\mathtt{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{struct}_A\ \overline{\mathtt{B}}\ \mathtt{end} : \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathtt{end}}$$

$$\textsc{Typ-M-ProjT}\ \frac{\Gamma \vdash^{\diamond} \mathtt{M} : \langle \gamma \rangle\, (= P < \mathtt{sig}_A\ \overline{\mathtt{D}}, \mathtt{module}\ X : \mathtt{S}', \overline{\mathtt{D}}'\ \mathtt{end})}{\Gamma \vdash^{\diamond} \mathtt{M}.X : \langle \gamma \rangle\, (= P.X < \mathtt{S}'[A \leftarrow P])} \qquad \textsc{Typ-M-ProjA}\ \frac{\Gamma \vdash^{\diamond} \mathtt{M} : \langle \gamma \rangle\, \mathtt{sig}_A\ \overline{\mathtt{D}}, \mathtt{module}\ X : \mathtt{S}', \overline{\mathtt{D}}'\ \mathtt{end}}{\Gamma \vdash^{\diamond} \mathtt{M}.X : \left\langle \gamma\, ; A : \overline{\mathtt{D}} \right\rangle \mathtt{S}'}$$

$$\textsc{Typ-B-Seq}\ \frac{\Gamma \vdash_A^{\diamond} \mathtt{B}_0 : \mathtt{D}_0 \qquad \Gamma \uplus A.\mathtt{D}_0 \vdash_A^{\diamond} \overline{\mathtt{B}} : \overline{\mathtt{D}}}{\Gamma \vdash_A^{\diamond} \mathtt{B}_0, \overline{\mathtt{B}} : \mathtt{D}_0, \overline{\mathtt{D}}} \qquad \textsc{Typ-B-Type-Bind}\ \frac{\Gamma \vdash \ddot{\mathtt{u}}}{\Gamma \vdash_A^{\diamond} (\mathtt{type}\ t = \ddot{\mathtt{u}}) : (\mathtt{type}\ t = \ddot{\mathtt{u}})} \qquad \textsc{Typ-B-Empty}\ \Gamma \vdash_A^{\diamond} \varnothing : \varnothing$$

$$\textsc{Typ-B-AbsType}\ \Gamma \vdash_A^{\diamond} (\mathtt{type}\ t = A.t) : (\mathtt{type}\ t = A.t) \qquad \textsc{Typ-B-Let}\ \frac{\Gamma \vdash^{\diamond} \mathtt{e} : \mathtt{u}}{\Gamma \vdash_A^{\diamond} (\mathtt{let}\ x = \mathtt{e}) : (\mathtt{val}\ x : \mathtt{u})} \qquad \textsc{Typ-B-Mod}\ \frac{\Gamma \vdash^{\diamond} \mathtt{M} : \mathtt{S}}{\Gamma \vdash_A^{\diamond} (\mathtt{module}\ X = \mathtt{M}) : (\mathtt{module}\ X : \mathtt{S})} \qquad \textsc{Typ-B-ModType}\ \frac{\Gamma \vdash \dot{\mathtt{S}}}{\Gamma \vdash_A^{\diamond} (\mathtt{module\ type}\ T = \ddot{\mathtt{S}}) : (\mathtt{module\ type}\ T = \ddot{\mathtt{S}})}$$

Fig. 10. Typing rules

be recovered by normalization. Besides, we drop the zipper-context γ, which would be useless as the information is already stored in Γ.

A signature ascription $(\ddot{P} : \ddot{\mathtt{S}})$ has the signature $\ddot{\mathtt{S}}$ provided it is indeed a supertype[19] of the signature $\mathbb{S}$ of $\ddot{P}$ (Rule Typ-M-Ascr). This ascription is opaque[20] since it returns the signature $\ddot{\mathtt{S}}$ which is not strengthened by $\ddot{P}$. Since subtyping is not code free, it is not present a floating subtyping rule but only allowed here and at functor applications.

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 works as expected. In particular, Rule Typ-B-Seq pushes the declaration $A.\mathtt{D}_0$ into the context while typing the $\overline{\mathtt{D}}$, much as Typ-D-Seq for signatures.

Finally, we have two rules for projection. Rule Typ-M-ProjT projects on a module that is a known alias of P. Therefore, we do not need to introduce a new zipper, as we can *strengthen* the resulting signature to mention P instead of A, removing dependencies with $\overline{\mathtt{D}}$ which are then dropped. By contrast, Rule Typ-M-ProjA is the key rule that leverages zippers. Intuitively, it just returns the signature $\mathtt{S}'$ of the field X of the signature $\mathtt{S}$ of $\mathtt{M}$ zipped around the initial fields $\overline{\mathtt{D}}$ of $\mathtt{S}$ appearing before the field X. Still, we have to consider that the signature of $\mathtt{M}$ may itself be in a zipper context γ, which is then composed with the zipper context formed of the initial fields $\overline{\mathtt{D}}$, resulting in $\gamma\, ; A : \overline{\mathtt{D}}$. As we pattern-match on the signature, this might require normalization. Importantly, the name A becomes fixed during projection, and is no longer freely α-convertible afterwards.

[19]The two first premises ensure that signatures $\mathbb{S}$ and $\ddot{\mathtt{S}}$ are both well-typed, as required when using subtyping.
[20]However, we can also use this construct to implement transparent ascription $(\ddot{P} < \ddot{\mathtt{S}})$ as syntactic sugar for $(\ddot{P} : (= \ddot{P} < \ddot{\mathtt{S}}))$, which then returns the view $\ddot{\mathtt{S}}$ but with the identity of $\ddot{P}$.

4 Resolving signature avoidance by zipper simplification

So far, zippers allow us to *delay* signature avoidance, but are sometimes polluting the inferred signatures. Indeed, zipper may be removed by ascription, but they are not simplified otherwise, even when they became useless. Yet, floating fields are not present at runtime and should only maintain the minimal amount of type information needed for a signature.

In this section, we address this issue by extending the definition of code-free subtyping $\lesssim$ and, indirectly, of type equivalence $\approx$ to enable simplification of zippers. In particular, as we are interested in *removing* floating fields whenever possible, we define *signature simplification* as an oriented subset of equivalence, i.e., a rewriting judgment $\Gamma \vdash \mathsf{S}_1 \rightsquigarrow \mathsf{S}_2$ that implies $\Gamma \vdash \mathsf{S}_1 \approx \mathsf{S}_2$. This simplification is our actual *solution* to the signature avoidance problem. Simplifying along the equivalence ensures that we never loose equalities between visible types, hence we never do simplification by over-abstraction, or simplificatinos that could prevent further typing, by contrast with OCaml. We may thus inject simplification into module typing with Rule Typ-M-Simp. While the primary goal of zipper simplification is to eliminate (or reduce) floating fields in inferred types, it also helps print simpler error messages. Early simplification may also speed up typechecking.

Typ-M-Simp
$$\frac{\Gamma \vdash \mathsf{M} : \mathsf{S} \qquad \Gamma \vdash \mathsf{S} \rightsquigarrow \mathsf{S}'}{\Gamma \vdash \mathsf{M} : \mathsf{S}'}$$

Overview. We start with the extension of subtyping in §4.1, where we allow subtyping with zipper signatures on the right-hand side. The extended code-free equivalence becomes quite expressive and allows for a complete reorganization of floating fields and zippers. In §4.2, we present a set of four simplification rules defined on a single floating field, which suffice for our purpose. In §4.3, we give an algorithm that computes an iteration of these simplification rules without revisiting the signature multiple times.

4.1 Subtyping with zippers

We extend code-free subtyping to support zippers on both sides: $\Gamma \vdash \left\langle A : \overline{\mathsf{D}}_1 \right\rangle \mathsf{S}_1 \lesssim \left\langle A : \overline{\mathsf{D}}_2 \right\rangle \mathsf{S}_2$.

4.1.1 Adding fields by subtyping. Intuitively, subtyping between zippers should commute with projection: it should be seen as if it happened before an hypothetical projection that created the zipper. That is, we can see $\left\langle A : \overline{\mathsf{D}}_1 \right\rangle \mathsf{S}_1$ and $\left\langle A : \overline{\mathsf{D}}_2 \right\rangle \mathsf{S}_2$ as the result of a projection of S'_1 and S'_2 on a module field, say $\mathbb{Z}$. To avoid loosing generality, we can assume that S'_1 is *any* well-formed signature whose projection gives $\left\langle A : \overline{\mathsf{D}}_1 \right\rangle \mathsf{S}_1$, while S'_2 is the *least* signature (with respect to the subtyping order) whose projection gives $\left\langle A : \overline{\mathsf{D}}_2 \right\rangle \mathsf{S}_2$. Subtyping before projection would give:

$$\frac{\Gamma \vdash \mathtt{sig}_A\ \overline{\mathsf{D}}_1, \mathtt{module}\ \mathbb{Z} : \mathsf{S}_1, \overline{\mathsf{D}}'_1\ \mathtt{end}\ \lesssim\ \mathtt{sig}_A\ \overline{\mathsf{D}}_2, \mathtt{module}\ \mathbb{Z} : \mathsf{S}_2\ \mathtt{end}}{\Gamma \vdash \left\langle A : \overline{\mathsf{D}}_1 \right\rangle \mathsf{S}_1 \lesssim \left\langle A : \overline{\mathsf{D}}_1 \right\rangle \mathsf{S}_2}$$

While sound, this would be weaker than necessary, as it would require code-free subtyping on floating fields, hence preserving value and module floating fields, while these are actually not present at run-time. On floating fields, coercion subtyping is actually code-free.

Therefore, we extend subtyping with the following stronger rule:

Sub-S-Zipper
$$\frac{\Gamma' = \Gamma \uplus A : \overline{\mathsf{D}}_1, \mathtt{module}\,\mathbb{Z} : \mathsf{S}_1, \overline{\mathsf{D}}'_1 \qquad \overline{\mathsf{D}}_0 \subseteq \overline{\mathsf{D}}_1, \overline{\mathsf{D}}'_1 \qquad \Gamma' \vdash \overline{\mathsf{D}}_0\ /\!/\ A \leq \overline{\mathsf{D}}_2 \qquad \Gamma' \vdash \mathsf{S}_1 \lesssim \mathsf{S}_2}{\Gamma \vdash \left\langle A : \overline{\mathsf{D}}_1 \right\rangle \mathsf{S}_1 \lesssim \left\langle A : \overline{\mathsf{D}}_2 \right\rangle \mathsf{S}_2}$$

To use it, one must find a set of additional floating fields $\overline{\mathsf{D}}'_1$ that may refer to both the floating fields $\overline{\mathsf{D}}_1$ and the signature S_1 via $\mathbb{Z}$. Then, a subset $\overline{\mathsf{D}}_0$ of $\overline{\mathsf{D}}_1, \overline{\mathsf{D}}'_1$ is subtyped against $\overline{\mathsf{D}}_2$, and S_1 is subtyping against S_2. This stronger rule (when allowed) make subtyping undecidable, as it may require to instantiate the abstract types of D_2 without any information from D_1. This is not a

$$\textsc{Zipper-Unused}\quad \frac{\left\langle A : \overline{\mathsf{D}}, \overline{\mathsf{D}}' \right\rangle \mathsf{S}}{\left\langle A : \overline{\mathsf{D}} \right\rangle \mathsf{S}} \qquad \textsc{Zipper-Intro}\quad \frac{\mathtt{sig}_A\ \overline{\mathsf{D}}\ \mathtt{end}}{\left\langle A : \overline{\mathsf{D}} \right\rangle \mathtt{sig}\ \overline{\mathsf{D}}\ /\!/\ A\ \mathtt{end}}$$

$$\textsc{Zipper-Intro-Module}\quad \frac{\mathtt{sig}_A\ \overline{\mathsf{D}}, \mathtt{module}\ X : \mathsf{S}, \overline{\mathsf{D}}'\ \mathtt{end}}{\langle A_0 : \mathtt{module}\ X : \mathsf{S}\rangle\ \mathtt{sig}_A\ \overline{\mathsf{D}}, \mathtt{module}\ X : (= A_0.X < \mathsf{S}), \overline{\mathsf{D}}'\ \mathtt{end}}$$

$$\textsc{Zipper-Intro-Type}\quad \frac{\mathtt{sig}_A\ \overline{\mathsf{D}}, \mathtt{type}\ t = A.t, \overline{\mathsf{D}}'\ \mathtt{end}}{\langle A_0 : \mathtt{type}\ t = A_0.t\rangle\ \mathtt{sig}_A\ \overline{\mathsf{D}}, \mathtt{type}\ t = A_0.t, \overline{\mathsf{D}}'\ \mathtt{end}} \qquad \textsc{Zipper-Extrude}\quad \frac{\mathtt{sig}_A\ \overline{\mathsf{D}}, \mathtt{module}\ X : \left\langle A_0 : \overline{\mathsf{D}}_0 \right\rangle \mathsf{S}, \overline{\mathsf{D}}'\ \mathtt{end}}{\left\langle A_0 : \overline{\mathsf{D}}_0 \right\rangle \mathtt{sig}_A\ \overline{\mathsf{D}}, \mathtt{module}\ X : \mathsf{S}, \overline{\mathsf{D}}'\ \mathtt{end}}$$

Fig. 11. Noticeable Equivalences

problem, as this extended version of subtyping is only used at the meta-theoretical level to prove the correctness of the simplification algorithm of §4.3.

4.1.2 Well-formedness when subtyping telescopes. However, Rule Sub-S-Zipper is not as *monotonic* as one might expect. Specifically, we do **not** have a weakening result for well-formedness:

$$\Gamma \uplus X : \left\langle A : \overline{\mathsf{D}}_2 \right\rangle \mathsf{S}_2 \vdash \mathsf{S} \ \wedge\ \Gamma \vdash \left\langle A : \overline{\mathsf{D}}_1 \right\rangle \mathsf{S}_1 \lesssim \left\langle A : \overline{\mathsf{D}}_2 \right\rangle \mathsf{S}_2 \ \not\Longrightarrow\ \Gamma \uplus X : \left\langle A : \overline{\mathsf{D}}_1 \right\rangle \mathsf{S}_1 \vdash \mathsf{S}$$

This comes from the fact that we consider the *domain* of the floating fields to be more or less *irrelevant* in Rule Sub-S-Zipper: we allow $\overline{\mathsf{D}}_1$ to have *fewer* fields than $\overline{\mathsf{D}}_2$ (the inverse of the usual rule), but we require the domain of S_1 to be larger than the one of S_2 (as usual). Yet, S might refer to floating fields in $\overline{\mathsf{D}}_2$ and therefore become ill-formed in the environment that only contains $\overline{\mathsf{D}}_1$ and not $\overline{\mathsf{D}}_2$. This is problematic: in the rule Sub-S-Sig for subtyping between structural signatures, we rely on this property to maintain well-formedness when subtyping declaration by declaration. Therefore, we change this rule to explicitly require an additional well-formedness condition:

$$\textsc{Sub-S-SigCheck}\quad \frac{\overline{\mathsf{D}}_0 \subseteq \overline{\mathsf{D}}_1 \qquad \Gamma \uplus A : \overline{\mathsf{D}}_1 \vdash \overline{\mathsf{D}}_2 \qquad \Gamma \uplus A : \overline{\mathsf{D}}_1 \vdash \overline{\mathsf{D}}_0\ /\!/\ A \leq \overline{\mathsf{D}}_2}{\Gamma \vdash \mathtt{sig}_A\ \overline{\mathsf{D}}_1\ \mathtt{end} \leq \mathtt{sig}_A\ \overline{\mathsf{D}}_2\ \mathtt{end}}$$

The well-formedness condition prevents the deletion of floating fields when those fields still appear in the rest of the right-hand side declarations $\overline{\mathsf{D}}_2$. Before we added Rule Sub-S-Zipper, this condition was always implied by the other premises using weakening, as the domain of D_1 included the domain of D_2. With Sub-S-Zipper, it must now be checked. The warning is that *subtyping with zippers and telescopes* is not as compositional as one might expect: subtyping (or equivalence) between sub-parts of signatures does not necessarily implies subtyping between those signatures.

4.1.3 A weaker rule. The extra well-formedness condition is only required when powerful Rule Sig-S-Zipper is used in a derivation of the premises. However, we can often use a *weaker* rule, where the domain of floating fields on the left-hand side is (as usual) just a subset of the right-hand side:

$$\textsc{Sub-S-Zipper-Weak}\quad \frac{\overline{\mathsf{D}}_0 \subseteq \overline{\mathsf{D}}_1 \qquad \Gamma \uplus A : \overline{\mathsf{D}}_1 \vdash \overline{\mathsf{D}}_0\ /\!/\ A \leq \overline{\mathsf{D}}_2 \qquad \Gamma \uplus A : \overline{\mathsf{D}}_1 \vdash \mathsf{S}_1 \leq \mathsf{S}_2}{\Gamma \vdash \left\langle A : \overline{\mathsf{D}}_1 \right\rangle \mathsf{S}_1 \leq \left\langle A : \overline{\mathsf{D}}_2 \right\rangle \mathsf{S}_2}$$

4.1.4 Equivalences. The additional expressiveness lies mainly in new equivalences between signatures with zippers. In fact, the increase is considerable, and the equivalence allows for floating fields—and therefore signatures—to be considerably reorganized. We illustrate this with some noticeable equivalences in Figure 11. We present them as rules where the signatures S_1 on top and S_2 on bottom are equivalent, leaving the context Γ and well-typedness of both sides implicit, as well as some additional conditions. Hence, each rule should be read as, if $\Gamma \vdash \mathsf{S}$ and $\Gamma \vdash \mathsf{S}'$ (and some additional freshness conditions that we detailed below for each rule) then $\Gamma \vdash \mathsf{S} \approx \mathsf{S}'$. The reader should keep in mind that those equivalences might not compose without extra well-formedness conditions. That is, $\Gamma \vdash \mathsf{S} \approx \mathsf{S}'$ and $\Gamma \vdash C[\mathsf{S}]$ does not imply $\Gamma \vdash C[\mathsf{S}']$ for all contexts C.

$$\dfrac{I = \mathsf{dom}(\mathsf{D}) \qquad A.I \notin \mathsf{fv}(\mathsf{S})}{\Gamma \vdash \langle A : \mathsf{D}\rangle\, \mathsf{S} \rightsquigarrow \mathsf{S}}\ \textsc{Simp-Drop} \qquad \dfrac{I = \mathsf{dom}(\mathsf{D}) \qquad \mathsf{anchor}\,(\Gamma, A.I, \mathsf{S}) = \mathbb{Z}.P.I'}{\Gamma \vdash \langle A : \mathsf{D}\rangle\, \mathsf{S} \rightsquigarrow \mathsf{S}[A.I \Leftarrow \mathbb{Z}.P.I']}\ \textsc{Simp-Move}$$

$$\dfrac{\Gamma \vdash \langle A_0 : \texttt{type}\ t = A_0.t\rangle \left\langle A : \texttt{module}\ X : \texttt{sig}_B\ \overline{\mathsf{D}}_0, \texttt{type}\ t = A_0.t, \overline{\mathsf{D}}'_0\ \texttt{end}\right\rangle \mathsf{S} \rightsquigarrow \dot{\mathsf{S}}}{\Gamma \vdash \left\langle A : \texttt{module}\ X : \texttt{sig}_B\ \overline{\mathsf{D}}_0, \texttt{type}\ t = A.t, \overline{\mathsf{D}}'_0\ \texttt{end}\right\rangle \mathsf{S} \rightsquigarrow \dot{\mathsf{S}}}\ \textsc{Simp-Split-Type}$$

$$\dfrac{\Gamma \vdash \mathsf{S}_1 \quad \Gamma \vdash \langle A_0 : \texttt{module}\ X_1 : \mathsf{S}_1\rangle \left\langle A : \texttt{module}\ X : \texttt{sig}_B\ \overline{\mathsf{D}}_0, \texttt{module}\ X_1 : (= A_0.X_1 < \mathsf{S}_1), \overline{\mathsf{D}}'_0\ \texttt{end}\right\rangle \mathsf{S} \rightsquigarrow \dot{\mathsf{S}}}{\Gamma \vdash \left\langle A : \texttt{module}\ X : (\texttt{sig}_B\ \overline{\mathsf{D}}_0, \texttt{module}\ X_1 : \mathsf{S}_1, \overline{\mathsf{D}}'_0\ \texttt{end})\right\rangle \mathsf{S} \rightsquigarrow \dot{\mathsf{S}}}\ \textsc{Simp-Split-Mod}$$

$$\dfrac{\mathsf{dom}(\mathsf{D}) = I \quad \Gamma \vdash \left\langle A : \overline{\mathsf{D}}_1\right\rangle ((\texttt{sig}_B\ \mathsf{D}, \texttt{module}\ \mathbb{Z} : \mathsf{S}\ \texttt{end})[A.I \leftarrow B.I]) \rightsquigarrow \left\langle A : \overline{\mathsf{D}}'_1\right\rangle \texttt{sig}_B\ \mathsf{D}', \texttt{module}\ \mathbb{Z} : \mathsf{S}'\ \texttt{end}}{\Gamma \vdash \left\langle A : \overline{\mathsf{D}}_1, \mathsf{D}\right\rangle \mathsf{S} \rightsquigarrow \left\langle A : \overline{\mathsf{D}}'_1, \mathsf{D}'\right\rangle \mathsf{S}'[B.I \leftarrow A.I]}\ \textsc{Simp-Skip}$$

Fig. 12. Simplification rules

As equivalences, the rules can be used in both directions, even if each direction reads differently. Rule Zipper-Unused can be used to drop—or conversely introduce—unused floating components. Well-typedness of both sides implies that fields in $\mathsf{dom}(\overline{\mathsf{D}}')$ are unused in S. Rule Zipper-Intro move all signature fields into a zipper context and let the body be a redirection to the zipper. The two next rules do this selectively: Rule Zipper-Intro-Type extracts an abstract type field t of a structural signature in a zipper-context and redirects that field in the signature body to the one in the zipper context. Zipper-Intro-Module is similar but for a module field. Well scoping implicitly requires that $A_0 \notin \mathsf{fv}(\overline{\mathsf{D}}, \overline{\mathsf{D}}')$ for both rules. Finally, Rule Zipper-Extrude extrude a floating field from its enclosing signature. Well-scoping requires that and $A \notin \mathsf{fv}(\overline{\mathsf{D}}_0)$ and $A_0 \notin \mathsf{fv}(\mathsf{S}')$ or $A_0 \notin \mathsf{fv}(\overline{\mathsf{D}}')$.

Renaming. A zipper $\langle A : \mathsf{D}\rangle\, \mathsf{S}$ uses a self-reference A to access the zipper context D. Initially, D is accessed from S but after normalization D may also be accessed by fields of a signature following the one that introduced the zipper. When $\Gamma \vdash \langle A : \mathsf{D}\rangle\, \mathsf{S}$ and A does not appear free in Γ, and A' is fresh for both Γ and S, we actually have $\Gamma \vdash \langle A : \mathsf{D}\rangle\, \mathsf{S} \approx \langle A' : \mathsf{D}[A \leftarrow A']\rangle\, \mathsf{S}[A \leftarrow A']$. Hence, zipper self-references can actually be renamed, but consistently.

4.2 Simplification $\boxed{\Gamma \vdash \mathsf{S} \rightsquigarrow \mathsf{S}'}$

In this section, we present a small set of *simplification* rules that transforms a signature along $\approx$, which we can use to reduce the size of the zipper. When we can remove the zipper altogether, this coincides with solving signature avoidance. When some floating fields remain, this is just delaying signature avoidance. At a high level, the simplification follows three elementary rules (*drop*, *move*, *split*), as it tries to simplify a single floating component, and one rule to re-organize zippers (*skip*). In this section, we consider a zipper signature $\langle A : \mathsf{D}\rangle\, \mathsf{S}$ where I is the identifier of D. The rules are given in §4.2 and discussed below.

Restrictions. Simplification covers a restricted subset of signature equivalence, namely: (1) Given a signature $\langle \gamma\rangle\, \mathsf{S}$ it can only *remove* floating fields from γ, but will not introduce new fields (even if this could result in a smaller zipper context). (2) it cannot move fields in the zipped signature S—while toplevel-equivalence allows it, but it can rewrite the content of the fields, rewiring the type and module equalities[21]. (3) it uses a first-order criterion for applicative functors: it only rewrites functor

[21] Substituting a module alias for another might rewrite the content of a signature and technically *move* fields, still in a code-free manner.

applications when aliases (for either a floating functor or a floating module argument) are available. This is similar to the anchoring restriction of [2].

4.2.1 Dropping a floating field. First, if a floating field is useless, i.e., does not appear in the free variables of the signature, we may just remove it, as done by Rule Simp-Drop. We may also use normalization to make floating module type fields and floating concrete type fields useless (as their name is replaced by their definition). Floating core-language value fields are always useless.

4.2.2 Moving a floating field. There are cases where the floating field D is not useless ($A.I$ appears in the signature even after normalization), but can still be removed. Indeed, there might be a declaration in S that can take up the same role as D, without loss of type information.

Anchoring points. We call such declaration an *anchoring point* for $A.I$ inside S, which we write anchor $(\Gamma, A.I, \mathsf{S})$ if it exists. It must validate three conditions: (1) when I is a type field, then D must be of the form $\mathtt{type}\, t' = A.t$; when I is a module field, then D must be of the form $\mathtt{module}\, X' : (= A.X < \mathsf{S})$ where S is equivalent to the signature of $A.X$ (not just a supertype). (2) it must be in a strictly positive position, inside neither a functor nor a module type. (3) it must come before any other occurrence of $A.I$ (which are called *usage points*). Formally, we have for a type field anchor $(\Gamma, A.t, \mathsf{S}) = \mathbb{Z}.X_1.(\dots).X_n.u$ if and only if there exists n signatures $\mathsf{S}_1, \dots, \mathsf{S}_n$ such that:

$$
\begin{array}{ll}
\mathsf{S} = \mathtt{sig}_{A_0}\ \overline{\mathsf{D}_0}\ \ \mathtt{module}\ X_1 : \mathsf{S}_1\ \overline{\mathsf{D}_0'}\ \mathtt{end} & \wedge\ A.t \notin \mathsf{fv}(\overline{\mathsf{D}_0}) \\
\mathsf{S}_1 = \mathtt{sig}_{A_1}\ \overline{\mathsf{D}_1}\ \ \mathtt{module}\ X_2 : \mathsf{S}_2\ \overline{\mathsf{D}_1'}\ \mathtt{end} & \wedge\ A.t \notin \mathsf{fv}(\overline{\mathsf{D}_1}) \\
\vdots & \\
\mathsf{S}_n = \mathtt{sig}_{A_n}\ \overline{\mathsf{D}_n}\ \ \mathtt{type}\ u = A.t\ \ \overline{\mathsf{D}_n'}\ \mathtt{end} & \wedge\ A.t \notin \mathsf{fv}(\overline{\mathsf{D}_n})
\end{array}
$$

$$
\begin{array}{l}
\langle A : \mathtt{type}\ t = A.t \rangle \\
\mathtt{sig}_{A_0} \\
\quad \overline{\mathsf{D}_0} \\
\quad \mathtt{module}\ X_1 : \mathtt{sig}_{A_1} \\
\qquad \overline{\mathsf{D}_1} \\
\qquad\quad \ddots \\
\qquad\qquad \overline{\mathsf{D}_{n-1}} \\
\qquad\qquad \mathtt{module}\ X_n : \mathtt{sig}_{A_n} \\
\qquad\qquad\quad \overline{\mathsf{D}_n} \\
\qquad\qquad\quad \mathtt{type}\ u = A.t \\
\qquad\qquad\quad \overline{\mathsf{D}_n'} \\
\qquad\qquad \mathtt{end} \\
\qquad\qquad \overline{\mathsf{D}_{n-1}'} \\
\qquad\quad \iddots \\
\qquad \overline{\mathsf{D}_1'} \\
\quad \mathtt{end} \\
\quad \overline{\mathsf{D}_0'} \\
\mathtt{end}
\end{array}
$$

That is, there is a cascade of depth n of nested signatures where $A.t$ is not mentioned until the field $\mathtt{type}\, u = A.t$. It is illustrated on the right for a type field.

Contextual path substitution. If there exists an anchoring point for $A.t$ at $\mathbb{Z}.X_1.(\dots).X_n.u$, we may remove the floating field by (intuitively) replacing occurrences of $A.I$ by $\mathbb{Z}.X_1.(\dots).X_n.u$. However, the path to access u is not the same everywhere inside the signature. Therefore, we need a special form of substitution, called *contextual path substitution*, written $\mathsf{S}[A.t \Leftarrow \mathbb{Z}.X_1.(\dots).X_n.u]$ that proceeds as follows:

- we replace $\mathtt{type}\, u = A.t$ by $\mathtt{type}\, u = A_n.u$ (deep in the signature)
- in $\overline{\mathsf{D}_0}, \dots, \overline{\mathsf{D}_n}$, there is nothing to substitute as $A.t$ does not appear free.
- then u is accessible by $A_n.u$, so we may substitute $A.t$ by $A_n.u$ in the declaration $\overline{\mathsf{D}_{n-1}'}$, by $A_{n-1}.X_n.u$ in the declaration $\overline{\mathsf{D}_{n-1}'}$, and, finally, by $A_0.X_1.(\dots).X_n.u$ in the declarations $\overline{\mathsf{D}_0'}$.

$$
\begin{array}{l}
\mathtt{sig}_{A_0} \\
\quad \overline{\mathsf{D}_0} \\
\quad \mathtt{module}\ X_1 : \mathtt{sig}_{A_1} \\
\qquad \overline{\mathsf{D}_1} \\
\qquad\quad \ddots \\
\qquad\qquad \overline{\mathsf{D}_{n-1}} \\
\qquad\qquad \mathtt{module}\ X_n : \mathtt{sig}_{A_n} \\
\qquad\qquad\quad \overline{\mathsf{D}_n} \\
\qquad\qquad\quad \mathtt{type}\ u = A_n.u \\
\qquad\qquad\quad \overline{\mathsf{D}_n'}[A.t \leftarrow A_n.u] \\
\qquad\qquad \mathtt{end} \\
\qquad\qquad \overline{\mathsf{D}_{n-1}'}[A.t \leftarrow A_{n-1}.X_n.u] \\
\qquad\quad \iddots \\
\qquad \overline{\mathsf{D}_1'}[A.t \leftarrow A_1.X_2.(\dots).X_n.u] \\
\quad \mathtt{end} \\
\quad \overline{\mathsf{D}_0'}[A.t \leftarrow A_0.X_1.X_2.(\dots).X_n.u] \\
\mathtt{end}
\end{array}
$$

Basically, when visiting S, contextual path substitution replaces $A.I$ by $A_1.X_2.(\dots).X_n.u$ stripped of it common prefix with the path of the current point. It is illustrated on the right for a type field.

$$
\begin{aligned}
& \langle A_1 : \mathtt{module}\ X : \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end}\rangle \\
& \quad \mathtt{sig}_B\ \mathtt{type}\ u = A_1.X.t\ \ \mathtt{module}\ X' : (= A_1.X < \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end})\ \mathtt{end} \\
\approx\ & \langle A_0 : \mathtt{type}\ t = A_0.t\rangle\ \langle A_1 : \mathtt{module}\ X : \mathtt{sig}_A\ \mathtt{type}\ t = A_0.t\ \mathtt{end}\rangle && (1) \\
& \quad \mathtt{sig}_B\ \mathtt{type}\ u = A_1.X.t\ \ \mathtt{module}\ X' : (= A_1.X < \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end})\ \mathtt{end} \\
\approx\ & \langle A_0 : \mathtt{type}\ t = A_0.t\rangle\ \langle A_1 : \mathtt{module}\ X : \mathtt{sig}_A\ \mathtt{type}\ t = A_0.t\ \mathtt{end}\rangle && (2) \\
& \quad \mathtt{sig}_B\ \mathtt{type}\ u = A_0.t\ \ \mathtt{module}\ X' : (= A_1.X < \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end})\ \mathtt{end} \\
\approx\ & \langle A_0 : \mathtt{type}\ t = A_0.t\rangle\ \mathtt{sig}_B\ \mathtt{type}\ u = A_0.t\ \ \mathtt{module}\ X' : \mathtt{sig}_C\ \mathtt{type}\ t = A_0.t\ \mathtt{end}\ \mathtt{end} && (3) \\
\approx\ & \mathtt{sig}_B\ \ \mathtt{type}\ u = B.u\ \ \mathtt{module}\ X' : \mathtt{sig}_C\ \ \mathtt{type}\ t = B.u\ \mathtt{end}\ \mathtt{end} && (4)
\end{aligned}
$$

Fig. 13. Illustration of simplification by splitting a type field (1). Here, we cannot drop the module field, neither can we move it (a usage point appears before an anchoring point). Yet, by splitting the type field we can simplify the zipper, but moving away the inner type (2) and module (3) fields, and the outer type field (4).

Floating module fields. For a floating module field $\langle A : \mathtt{module}\ X : \mathsf{S}\rangle$, the definition of the anchoring point is the same, except that it must be a module declaration with a transparent signature:

$$\mathsf{S}_n = \mathtt{sig}_{A_n}\ \overline{\mathsf{D}_n}\ \ \mathtt{module}\ X' : (= A.X < \mathsf{S}')\ \ \overline{\mathsf{D}_n}'\ \ \mathtt{end}\ \wedge\ A.X \notin \mathsf{fv}(\overline{\mathsf{D}_n})$$

Besides, there is an additional equivalence condition to be satisfied:

$$\Gamma \uplus A_0 : \overline{\mathsf{D}}_0 \uplus A_1 : \overline{\mathsf{D}}_1 \uplus \ldots \uplus A_n : \overline{\mathsf{D}_n} \vdash \mathsf{S} \approx (\mathsf{S}' \ / \ A.X)$$

4.2.3 Splitting a floating module field. The splitting rules, Simp-Split-Type for type fields and Simp-Split-Mod for module fields, are meant to be used in conjunction with the rules for moving and dropping fields. They simply allow to *temporarily* introduce new floating fields if and only if those additional fields help the module field get simplified and removed in the end. An example is given in Figure 13. Both rules pattern match on the *absence* of a zipper on the right-hand side signature, therefore allowing only splitting when it helps simplification. Importantly, the rules apply only if the module X as a structural signature, not a functor signature. This is where the *first-order* restriction can be seen, as we do not try to *split* individual fields of functors. Simplification of functors can only use Rule Simp-Move-Mod. Implemented naively, splitting would require exponential backtracking and would be impractical.

4.2.4 Skipping a field. The three simplification rules pattern-match on the right-most floating field. If the right-most field does not fall into one of the three cases above, we can skip it and leave it as a floating component. Rule Simp-Skip just amounts to consider the last field D as part of the visible signature, simplify the rest, and put D back into the zipper. Implemented naively, skipping would require two substitutions and the allocation of a signature for each skipped field.

4.3 Simplification algorithm

In this subsection we draft an algorithm that simplifies a whole zipper using the simplification rules presented above. It works in three steps. First, *scanning* browses the zipper and the signature to collect information about the usage points of each floating field. Then, constraint resolution computes which floating field are going to be *dropped*, *moved*, *split*, or *skipped*. Finally, a *simplification* pass revisits the signature to apply the corresponding transformations.

In the rest of this section we consider the simplification of a zipper signature $\langle A : \mathsf{D}_1, \ldots\ \mathsf{D}_n\rangle\ \mathsf{S}$, assigned to variable $\mathbb{Z}$. We denote by I_k the identifier of the floating field D_k.

4.3.1 Scanning. We start by a depth-traversal of the signature, updating a mutable map ϕ that matches each identifier I_k with a list of *usage points* stored in order of appearance. Each usage point is one of three kinds:

- **`zipper`** $(A.I_\ell)$ indicates that $A.I_k$ is used in the floating field $A.\mathsf{D}_\ell$.

- **anchor** $(A.I_k.\overline{X}.I,\ \mathbb{Z}.\overline{X'}.I')$ is used when the declaration at $\mathbb{Z}.\overline{X'}.I'$ could serve as an anchoring point for the subfield $A.I_k.\overline{X}.I$ if no occurrence of $A.I_k$ appears before $\mathbb{Z}.\overline{X'}.I'$.
- **usage** is used otherwise. For module fields where $A.I_k$ is used as a prefix, we store the whole path, as **usage** $(A.I_k.\overline{X}.I)$; otherwise, **usage** has no argument.

We then define a function $\text{visit}_P(\cdot)$ that visits the zipper and the signature recursively, in order of appearance, while updating the map (hence adding new elements to the tail of the list). P is an optional argument that is only be passed when visiting the zipper body S. Initially, ϕ maps each I_k to an empty list. When visiting the zipper context, only **zipper** $(\cdot)$ usage points are used.

When visiting the signature S, the optional path argument P indicates the path to the root $\mathbb{Z}$ if still accessible, or none otherwise. The initial call is $\text{visit}_{\mathbb{Z}}\ \mathsf{S}$. A type declaration is first check as a possible anchoring point when the path is nonempty. If not an anchoring point or if the path is empty, then it is a usage point. The path is set to none for recursive calls when entering, (1) a functor or a module-type, (2) a submodule with a transparent signature, as paths reaching inside the submodule are normalized away, and (3) a submodule with a module-type signature (as normalization should be used first).

4.3.2 Constraint resolution. For constraint resolution, we use the information collected in the first pass to compute the set of simplifications that can be applied to the signature. For that purpose, we introduce a single assignment map Ω from paths of the form $A.I_k.\overline{X}.I'$ to actions, initially undefined everywhere, and which may be set once one value among **drop**, **move**, **split**, and **skip**. Constraint resolution updates Ω and returns a list of substitutions Θ to be applied to S.

We visit the floating fields I_k in reverse order, i.e., for k ranging from n to 1. We consider the list $\phi(I_k)$ of usage points collected in the first phase. We first remove from $\phi(I_k)$ all usage points **zipper** $(A.I_\ell)$ for which $\Omega(I_\ell)$ is **drop**, **move**, or **split**, since then I_ℓ will be removed during the simplification. We then scan the list $\phi(I_k)$ of remaining usage points in order:

- if $\phi(I_k)$ is empty, we set $\Omega(I_k)$ to **drop**.
- if the head of $\phi(I_k)$ is **anchor** $(P,\ P')$ we set $\Omega(I_k)$ to **move** and Θ to $[P \Leftarrow P'] \circ \Theta$.
- Otherwise, the head of $\phi(I_k)$ is a usage point. If the field I_k is a module declaration, we try to split that field. That is, we consider a local assignment map ω for suffixes of $A.X$ and we go through the list $\phi(I_k)$ of usage points, with the following cases:
  - **usage** $(A.I_k.\overline{X}.I)$: if $A.I_k.\overline{X}.I$ or any prefix of the form $A.I_k.\overline{X'}$ is already set to **move** in ω, we remove the current usage point from $\phi(I_k)$, and continue with the rest $\phi(I_k)$; otherwise, we set $\Omega(I_k)$ to **skip**, and proceed with the successor of I_k (discarding ω).
  - **anchor** $(P,\ P')$: if no prefix of P is already set to **move**, we set $\omega(A.I_k.\overline{X}.I)$ to **move** and Θ to $[P \Leftarrow P'] \circ \Theta$.

  If all usage points of the modules have been dealt with, we set $\Omega(I_k)$ to **split** and discard ω.

4.3.3 Simplification. Finally, we return the zipper $\left\langle A : \mathsf{D}_k{}^{k\in 1..n \wedge \Omega(A.I_k)=\mathbf{skip}} \right\rangle (\mathsf{S}\Theta)$ whose context just retained the floating fields set to **skip**, applying the path contextual substitution Θ to S.

5 Properties

The main property, type soundness, is proved by elaboration of ZipML into M^ω [2], which is sound, while preserving the underlying untyped semantics. This is done in §5.1. In §5.3, we present some arguments supporting a completeness conjecture, i.e., that any M^ω program can be elaborated into an equivalent ZipML program. However, since ZipML as a module-level notion of sharing, the comparison should be made with version M^ω_{id} of M^ω instrumented with module-level sharing, obtained by the composition of the source-to-source transformation that introduces identity tags at the level of modules prior to typing in M^ω, as described in [2, §3.6], or equivalently, using derived rules operating directly on source terms without their translation. We choose the later below, but just keep writing M^ω for M^ω_{id}. By contrast with [2], we use pairs to $(\!|\tau, \mathcal{C}|\!)$ to represent tagged signatures $\mathsf{sig}\ \ \mathsf{type}\ id = \tau\ \ \mathsf{module}\ Val : \mathcal{C}\ \ \mathsf{end}$ and keep them concise and distinct from regular signatures. We still keep letter $\mathcal{C}$ to range over either regular or tagged signatures.

5.1 Soundness of ZipML by Elaboration in M^ω

We expect the reader to be familiar with elaboration of ML modules into F^ω, ideally M^ω [2] (or *F-ing* [24]). This elaboration serves as a proof of soundness of ZipML. Elaborating in M^ω rather than directly in F^ω allows to benefit from the sound extrusion and skolemization of M^ω. By lack of space, we must refer the reader to [2] for a precise definition of the typing judgments of M^ω.

As both ZipML and M^ω have the same source language and the same untyped semantics, the goal is to show the following theorem, so that ZipML inherits type soundness from M^ω:

Theorem 5.1 (Soundness). *Every module expression that typechecks in ZipML (without the simplification rule) also typechecks in M^ω. That is,* $\vdash^\diamond \mathsf{M} : \mathsf{S}$ *implies* $\Vdash \mathsf{M} : \exists^\diamond \overline{\alpha}.\mathcal{C}$ *for some* $\overline{\alpha}$ *and* $\mathcal{C}$.

We have only proved soundness in the absence of simplifications, i.e., using the subtyping relation of §3 without rule Sub-S-Zipper. The soundness of simplifications is left for future work. It would either require updating the elaboration of subtyping (Lemma 5.6), or done purely in ZipML by showing that early simplifications can always be postponed, hence performed at the very end of typechecking.

We write $\Vdash$ for judgments in M^ω, as opposed to $\vdash$ for judgments in ZipML, and we use a light red background. 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 typing environments Γ into M^ω ones, written $\Vdash \Gamma \rightsquigarrow \Gamma_\omega$. The induction hypothesis for the soundness statement becomes:

$$\Gamma \vdash^\diamond \mathsf{M} : \mathsf{S}\ \wedge\ \Vdash \Gamma \rightsquigarrow \Gamma_\omega \qquad \Longrightarrow \qquad \exists \overline{\alpha}, \mathcal{C}.\ \ \Gamma_\omega \Vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}\ \wedge\ \Gamma_\omega \Vdash \mathsf{M} : \exists^\diamond \overline{\alpha}.\mathcal{C}$$

Yet, we need to extend M^ω elaboration of signatures to support zippers for this statement to make sense. The rest of this section is composed as follows:

- we first explain the treatment of zippers;
- we discuss the elaboration of abstract types in the environment and extend the induction hypothesis;
- we state the elaboration of strengthening, normalization, subtyping, and resolution;
- we state the elaboration of path typing, signature typing, and module typing;

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^ω, with the introduction of zippers signatures $\langle \gamma \rangle\, \mathsf{S}$ and zipper-context accesses in paths $P.A$. Intuitively, zippers are removed at runtime, and could be removed in M^ω. However, to establish a one-to-one correspondence on inferred signatures, we need to keep zippers in both inferred signatures and the environment. Therefore, we define M^ω_{zip}, an extension of M^ω with zippers. We add zippers to M^ω signatures with the following signature elaboration rule, along with appropriate path typing extension.

MZip-Typ-S-Zip
$$\frac{\Gamma \Vdash_A \overline{\mathsf{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}} \qquad \Gamma, A : \overline{\mathcal{D}} \Vdash \mathsf{S} : \lambda\overline{\beta}.\mathcal{C}}{\Gamma \Vdash \left\langle A : \overline{\mathsf{D}} \right\rangle \mathsf{S} : \lambda\overline{\alpha}, \overline{\beta}.\left\langle A : \overline{\mathcal{D}} \right\rangle \mathcal{C}}$$

MZip-Typ-P-Zip
$$\frac{\Gamma \Vdash P : \left\langle A : \overline{\mathcal{D}} \right\rangle \mathcal{C}}{\Gamma \Vdash P.A : \mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}}$$

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. In a typing derivation of

$\mathsf{M}^{\omega}_{zip}$ that does not use MZip-Typ-S-Zip nor MZip-Typ-P-Zip, we can erase all zippers and obtain a normal derivation of M^{ω}.

5.1.2 Elaboration of environments and strengthening. When elaborating a typing environment Γ of ZipML into a typing environment Γ_ω of M^{ω}, a technical point arises from the fact that declarations and signatures in Γ are always strengthened, and are therefore self-referring. By contrasts, abstract types variables are inserted in Γ_ω before declarations and signatures. We define a (non-algorithmic) rule M-Typ-E-Decl that captures the *self-referring* representation of abstract types in ZipML. It is best understood by its key property, shown on the right:

$$\textsc{M-Typ-E-Decl}\quad \frac{\Vdash \Gamma \rightsquigarrow \Gamma_\omega \qquad \Gamma, \overline{\alpha}, A : \mathcal{D} \Vdash_A \mathsf{D} : \mathcal{D}}{\Vdash (\Gamma, A : \mathsf{D}) \rightsquigarrow (\Gamma_\omega, \overline{\alpha}, A : \mathcal{D})} \qquad\qquad \frac{\Vdash \Gamma \rightsquigarrow \Gamma_\omega \qquad \Gamma_\omega \Vdash_A \mathsf{D} : \lambda\overline{\alpha}.\mathcal{D}}{\Vdash (\Gamma \uplus A : \mathsf{D}) \rightsquigarrow (\Gamma_\omega, \overline{\alpha}, A : \mathcal{D})}$$

The rule is not algorithmic as it requires *guessing* the set of abstract type variables $\overline{\alpha}$ and the result of the elaboration $\mathcal{D}$. However, as shown by the derived rule on the right, $\overline{\alpha}$ and $\mathcal{D}$ can be obtained by elaborating the unstrengthened declaration D_0, when it is added to the context. This is sufficient to conduct our proof, as declarations are only added to the context with $\uplus$. We have similar rules and properties for adding functor parameters and zippers to the environment.

We have the following property for strengthening:

Lemma 5.2 (Elaboration of strengthening). *Given a path P and a signature S, such that $[\![\Gamma]\!] \Vdash P : \mathcal{C}'$ and $[\![\Gamma]\!] \Vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}$ and $[\![\Gamma]\!] \Vdash \mathcal{C}' \leq \mathcal{C}[\overline{\alpha} \leftarrow \overline{\tau}]$, we have $[\![\Gamma]\!] \Vdash \mathsf{S} \,/\!/\, P : (\lambda\overline{\alpha}.\mathcal{C})\,\overline{\tau}$.*

5.1.3 Elaboration of judgments. As normalization, path resolution, path typing, and subtyping are mutually recursive, we state four combined properties that are proved 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 5.3 (Elaboration of normalization). *If S normalizes to S', i.e., $\Gamma \vdash \mathsf{S} \downarrow \mathsf{S}'$, and if we have $\Vdash \Gamma \rightsquigarrow \Gamma_\omega$, then both signatures have the same elaboration in M^{ω}. That is, $\Gamma_\omega \Vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}$ implies $\Gamma_\omega \Vdash \mathsf{S}' : \lambda\overline{\alpha}.\mathcal{C}$.*

Lemma 5.4 (Elaboration of path-typing). *If $\Gamma \vdash P : \mathsf{S}$ and $\Vdash \Gamma \rightsquigarrow \Gamma_\omega$ hold, then the typing of P and of its signature S coincide in M^{ω}, i.e., we have both $\Gamma_\omega \Vdash P : (\!|\tau, \mathcal{C}|\!)$ and $\Gamma_\omega \Vdash \mathsf{S} : (\!|\tau, \mathcal{C}|\!)$.*

Lemma 5.5 (Elaboration of path resolution). *Resolving a path P allows to fetch either the identity or the content of the signature. Assuming $\Vdash \Gamma \rightsquigarrow \Gamma_\omega$, we have:*

- *If $\Gamma \vdash P \triangleright \mathsf{S}$ then we have $\Gamma_\omega \Vdash P : (\!|\_, \mathcal{C}|\!)$ and $\Gamma_\omega \Vdash \mathsf{S} : \lambda\alpha.(\!|\_, \mathcal{C}|\!)$;*
- *If $\Gamma \vdash P \triangleright P'$ then we have $\Gamma_\omega \Vdash P : (\!|\tau, \mathcal{C}|\!)$ and $\Gamma_\omega \Vdash P' : (\!|\tau, \mathcal{C}'|\!)$ and $\Gamma_\omega \Vdash \mathcal{C}' \leq \mathcal{C}$.*

Lemma 5.6 (Elaboration of subtyping). *The elaboration preserves the subtyping relationship: If $\Gamma_\omega \Vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}$ and $\Gamma_\omega \Vdash \mathsf{S}' : \lambda\overline{\alpha}'.\mathcal{C}'$ then $\Gamma \vdash \mathsf{S} \leq \mathsf{S}'$ implies $\Gamma_\omega \Vdash \lambda\overline{\alpha}.\mathcal{C} \leq \lambda\overline{\alpha}'.\mathcal{C}'$.*

Lemma 5.7 (Elaboration of signature typing). *Well-typed signatures of ZipML can be elaborated in M^{ω}. If $\Gamma \vdash \mathsf{S}$ and $\Vdash \Gamma \rightsquigarrow \Gamma_\omega$ then $\exists \overline{\alpha}, \mathcal{C}.\ \Gamma_\omega \Vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}$.*

Soundness as stated by Theorem 5.1 relies on the lemmas Lemmas 5.3 to 5.6], which are proved by mutual induction over the typing derivations of their premises.

5.2 On abstraction safety

The language M^{ω}_{id} has been designed to ensure abstraction safety. Although just conjectured, the abstraction safety of M^{ω}, should transferred to ZipML by type soundness of the elaboration. Indeed, assume that two programs $\mathtt{M}_1$ and $\mathtt{M}_2$ have compatible signatures S_1 and S_2 in ZipML, that is $\Gamma \vdash \mathsf{M}_i : (= P < \mathsf{S}_i)$ for i in $\{1, 2\}$ with the same identity P. By the elaboration of path resolution lemma, we have $\Gamma_\omega \Vdash \mathsf{M}_i : (\!|\tau, C_i|\!)$ (**1**), for a same identity type τ. Therefore, we can apply the abstraction safety property of M^ω.

5.3 Completeness of ZipML with respect to M^ω

Conversely, one may wonder whether the type system of ZipML is powerful enough to simulate M^ω. We argue that it is indeed the case. In this section, we present some key properties supporting this claim. Namely, we remark that: (1) floating fields can be used to encode existentially quantified variables; (2) code free equivalence on zippers can simulate the M^ω extrusion and skolemization of variables; and (3) universally and lambda bound variables need not be encoded.

5.3.1 Encoding existentially quantified types with floating fields. Our claim is that *floating fields can encode all existentially quantified variables of M^ω*. We first consider first-order type variables, then module identities, and, finally, higher-order types.

First-order type variables. Let us consider a M^ω-signature with a single quantified type variable of the base kind $\star$, which is of the form $\exists \alpha. C$. Inside C, the variable α is accessible everywhere. This is quite similar to a zipper signature $\langle A : \texttt{type}\ t = A.t \rangle\ \mathsf{S}$, where $A.t$ is a type accessible everywhere inside S. Therefore, we could translate the M^ω-signature into a zipper signature by introducing a floating field with a fresh name for every quantified variable. We may define a reverse elaboration judgment $\Gamma_\omega \Vdash \exists \overline{\alpha}. C \hookleftarrow\!\!\!\hookrightarrow \langle \gamma \rangle\ \mathsf{S}$, where we extend the environment to attach the name of a floating field to every (existentially) quantified abstract type. We would have rules of the form:

$$\textsc{Rev-S-Star}\quad \frac{\Gamma_\omega, \alpha \hookleftarrow\!\!\!\hookrightarrow A.t \Vdash C \hookleftarrow\!\!\!\hookrightarrow \mathsf{S} \qquad A \text{ fresh}}{\Gamma_\omega \Vdash \exists \alpha. C \hookleftarrow\!\!\!\hookrightarrow \langle A : \texttt{type}\ t = A.t \rangle\ \mathsf{S}} \qquad\qquad \textsc{Rev-T-AbsType}\quad \frac{\alpha \hookleftarrow\!\!\!\hookrightarrow A.t \in \Gamma_\omega}{\Gamma_\omega \Vdash \alpha \hookleftarrow\!\!\!\hookrightarrow A.t}$$

α-conversion. A difference between quantified variables and fields of zippers is that the latter are not α-convertible. Technically, α-convertibility is part of type-equivalence and can be applied anywhere in a M^ω derivation. However, in ZipML, self-references are α-convertible, which gives us the following (top-level) equivalence: $\langle A : \texttt{type}\ t = \mathsf{u} \rangle\ \mathsf{S} \approx \langle B : \texttt{type}\ t = \mathsf{u} \rangle\ \mathsf{S}[A \leftarrow B]$. For all practical purposes, floating fields are as α-convertible as existential types.

Module identities. Modules identities of M^ω can be encoded as floating module fields. However, there is a difficulty: what is the attached signature of an identity floating field? The simplest answer is to extend ZipML with a *bottom signature* $\bot$ that is a subtype of all signatures. Doing so, we would get rules of the following form:

$$\textsc{Rev-S-Mod}\quad \frac{\Gamma, \alpha_{id} \hookleftarrow\!\!\!\hookrightarrow A.X \Vdash C \hookleftarrow\!\!\!\hookrightarrow \mathsf{S} \qquad A \text{ fresh}}{\Gamma \Vdash \exists \alpha_{id}. C \hookleftarrow\!\!\!\hookrightarrow \langle A : \texttt{module}\ X : \bot \rangle\ \mathsf{S}} \qquad\qquad \textsc{Rev-S-Transparent}\quad \frac{\alpha_{id} \hookleftarrow\!\!\!\hookrightarrow A.X \in \Gamma_\omega \qquad \Gamma \Vdash C \hookleftarrow\!\!\!\hookrightarrow \mathsf{S}}{\Gamma \Vdash (\!|\alpha_{id}, C|\!) \hookleftarrow\!\!\!\hookrightarrow (= A.X < \mathsf{S})}$$

However, as hinted in by [2, Theorem 3.1], module identities of M^ω are always attached to signatures that have a common ancestor in the subtyping order. Therefore, rather than extending ZipML with a bottom signature $\bot$, we could instrument the typing rules of M^ω to obtain the common ancestor signature associated with each module identity and use it in the corresponding floating field.

Higher-order types. Higher-order types of M^ω can be encoded as floating functors producing a single type field. Again, there is a subtlety: what is the signature of the domain of such a functor? The simplest answer is to extend ZipML with a *top signature* $\top$ that is a supertype of all signatures.

Using it, we would get rules of the form:

Rev-S-HigherOrder
$$\frac{\Gamma, \varphi \hookleftarrow A.F(\cdot).t \Vdash C \hookleftarrow \mathsf{S}}{\Gamma \Vdash \exists^{\blacktriangledown} \varphi . C \hookleftarrow \langle A : \mathtt{module}\ F : (Y : \top) \to \mathtt{sig}_B\ \mathtt{type}\ t = B.t\ \mathtt{end}\rangle\, \mathsf{S}}$$

Rev-T-Transparent
$$\frac{\varphi \hookleftarrow A.F(\cdot).t \in \Gamma_\omega \qquad \alpha_{id} \hookleftarrow B.X}{\Gamma \Vdash \varphi(\alpha_{id}) \hookleftarrow A.F(B.X).t}$$

Higher-order module identities would work similarly. Following the same reasoning as for module identities, we conjecture that, rather than extending ZipML with a top signature $\top$, we could instrument the typing rules of M^ω to obtain the domain of the functor that originally introduced φ, which is a supertype of all use-cases.

Universally and lambda quantified types. Our argument applies to existentially quantified types. But the universal quantification and lambda quantification of M^ω are always used for signatures that come from elaboration of the source (namely, functor parameter and module type definitions), and can therefore also be represented in ZipML, without using floating field to encode type variables.

5.3.2 *Extrusion.* Floating fields can also be extruded, similarly to existential types. For instance, if we consider type components inside submodules, we can introduce floating fields and use equivalence to emulate extrusion. For instance, we have the following equivalences:

$$\begin{aligned}
& \mathtt{sig}_A\ \mathtt{module}\, X_1 : \mathtt{sig}_B\ \mathtt{type}\ t = B.t\ \mathtt{end}\ \ \mathtt{module}\, X_2 : \mathtt{sig}_C\ \mathtt{type}\ t = C.t\ \mathtt{end}\ \mathtt{end} \\
\approx\ & \mathtt{sig}_A\ \mathtt{module}\, X_1 : \langle A_1 : \mathtt{type}\ t = A_1.t\rangle\, \mathtt{sig}\ \mathtt{type}\ t = A_1.t\ \mathtt{end} \\
& \qquad \mathtt{module}\, X_2 : \langle A_2 : \mathtt{type}\ t = A_2.t\rangle\, \mathtt{sig}\ \mathtt{type}\ t = A_2.t\ \mathtt{end}\ \mathtt{end} \\
\approx\ & \langle A_1 : \mathtt{type}\ t = A_1.t\, ; A_2 : \mathtt{type}\ t = A_2.t\rangle \\
& \quad \mathtt{sig}_A\ \mathtt{module}\, X_1 : \mathtt{sig}\ \mathtt{type}\ t = A_1.t\ \mathtt{end}\ \mathtt{module}\, X_2 : \mathtt{sig}\ \mathtt{type}\ t = A_2.t\ \mathtt{end}\ \mathtt{end}
\end{aligned}$$

This also applies to *skolemization*, as we also have the following equivalences:

$$\begin{aligned}
& (Y : \mathsf{S}_a) \to \mathtt{sig}_A\ \mathtt{type}\ t = A.t\ \mathtt{end} \\
\approx\ & (Y : \mathsf{S}_a) \to \langle B : \mathtt{type}\ t = B.t\rangle\, \mathtt{sig}\ \mathtt{type}\ t = B.t\ \mathtt{end} \\
\approx\ & \langle B : \mathtt{module}\ F : (Y_0 : \mathsf{S}_a) \to \mathtt{sig}_D\ \mathtt{type}\ t = D.t\ \mathtt{end}\rangle\, (Y : \mathsf{S}_a) \to \mathtt{sig}\ \mathtt{type}\ t = B.F(Y).t\ \mathtt{end}
\end{aligned}$$

Here, for the domain of the floating functor, we could also the top signature $\top$ instead of the signature S_a of the functor we extruded from, as we did.

Proof sketch. Overall, equivalence over floating fields allows us to mimic the extrusion and skolemization mechanisms of M^ω in ZipML. Informally, it should allow us to maintain a compositional correspondence between typings in M^ω and typings in ZipML. At each step, we would be able to combine the ZipML signatures obtained by induction hypothesis and use equivalence to make them correspond to M^ω.

5.4 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 depending on the amount of normalization that has been performed. Hence, we may have $\Gamma \vdash \mathtt{M} : \mathsf{S}$ and $\Gamma \vdash \mathtt{M} : \mathsf{S}'$ when S and S' syntactically differ—even a lot! as one may contain a signature definition that has been 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 anonymous signature that 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 delay and resolve instances of signature avoidance. ZipML also models transparent ascription, delayed strengthening, applicative and generative functors and parsimonious inlining of signatures. Several features of OCaml are still missing in ZipML.

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 §3.6.

OCaml allows *abstract signatures* which amounts to quantifying over signatures in functors. This feature, while rarely used in practice, unfortunately makes the system undecidable [21, 29]. 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, 19]. 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 [23], which could fit well with floating fields.

We leave these explorations to future work. 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 lose 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 Robert Harper. 1990. *The Definition of Standard ML (revised)*. MIT Press, Cambridge, MA, USA.

[17] Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA. https://doi.org/10.7551/mitpress/2319.003.0001

[18] 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

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

[20] 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

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

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

[23] 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

[24] 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

[25] Claudio V. Russo. 2000. First-Class Structures for Standard ML. In *Programming Languages and Systems*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Gert Smolka (Eds.). Vol. 1782. Springer Berlin Heidelberg, Berlin, Heidelberg, 336–350. https://doi.org/10.1007/3-540-46425-5_22 Series Title: Lecture Notes in Computer Science.

[26] Claudio V. Russo. 2001. Recursive structures for standard ML. *SIGPLAN Not.* 36, 10 (oct 2001), 50–61. https://doi.org/10.1145/507669.507644

[27] 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

[28] 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

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