

Mixin’ Up the ML Module System

Derek Dreyer Andreas Rossberg

Max Planck Institute for Software Systems

{dreyer,rossberg}@mpi-sws.mpg.de

Abstract

ML modules support hierarchical namespace management, as well as both abstract and transparent type components, but they do not support recursive linking of separately compiled modules. Mixin modules support recursive linking of separately compiled modules, but they are not hierarchically composable and typically do not contain type components, abstract or otherwise. We synthesize the complementary advantages of these two mechanisms in a new, unified module system design called MixML. A MixML module is like an ML structure in which some of the components are specified but not defined. In other words, it unifies the ML structure and signature languages into one. MixML seamlessly integrates hierarchical composition, translucent data abstraction, and mixin-style recursive linking. The design of MixML is minimalist, emphasizing how superficially distinct features of the ML module system can be modeled as stylized uses of the same underlying constructs, with mixin composition playing a central role. Of particular note is the treatment of ML’s “sealing” construct. Sealing and mixin composition are viewed as two sides of the same coin—opaque linking vs. transparent linking—with remarkably similar typing rules.

1. Introduction

ML modules and *mixin modules* are two well-known, yet very different, mechanisms for modular programming that have largely complementary advantages and disadvantages. In this paper, we show how to synthesize features of both approaches in the design of a new, unified module system called MixML. To begin, we review what the two mechanisms are, and what their strengths and weaknesses are.

1.1 ML Modules

Proposed originally by MacQueen [23] in the mid-1980s and developed further by Harper, Leroy, and many others [16, 20, 12], the ML module system offers powerful support for robust program construction and data abstraction. In ML, *structures* provide namespace management, *signatures* describe module interfaces, *functors* enable the definition of generic modules, and *opaque signature ascription* (aka *sealing*) lets one hide the implementation details of a module behind an interface.

One important feature of ML modules is that they are *hierarchical*. Structures may contain other structures as components, and thus be used to build hierarchical namespaces. Another criti-

cally important feature is that modules may contain both *dynamic* components, defined by core-ML terms, and *static* components, defined by core-ML types. The packaging together of types and terms, along with the opaque sealing construct, allows modules to express *abstract data types*. Furthermore, signatures are *translucent* [16], meaning that they can specify type components of modules either abstractly or transparently. This translucency gives the programmer fine-grained control over the propagation of type information.

However, one major limitation of ML modules (at least traditionally) is that they cannot be defined recursively, thus inhibiting the decomposition of mutually recursive functions and data types into modular components. Consequently, in the last decade, there have been several proposals for extending ML with recursive modules [7, 32, 22, 26, 8]. While the existing proposals address a variety of interesting issues, such as the interaction of recursion and data abstraction [7, 8], none of them provides adequate support for something we view as a central design goal: *separate compilation*. Half the motivation for recursive modules is the ability to break big modules into smaller ones that are independent of one another and can be compiled separately. Yet, except in restricted cases, this functionality is not accounted for by any of the existing proposals.

We believe that the reason existing proposals have failed to support general separate compilation of mutually recursive modules is that ML’s traditional means of supporting separate compilation and (non-recursive) linking—*functors* and *functor application*, respectively—do not scale well to the recursive case. The body of a functor (which defines its *exports*) may depend on its argument (which specifies its *imports*), but not vice versa. In the context of recursive modules, however, the import specifications of a separately-compiled module may need to refer recursively to abstract type components provided in its exports. It is not obvious how to generalize the functor mechanism in a simple way so that the argument may depend on the result.

1.2 Mixin Modules

Although the concept of *mixins* originated in work on Common LISP from the mid-1980s [25], Bracha and Cook [4] were the first to propose mixins as an actual language construct (in their case, as an extension to Modula-3 [6]). Since then, mixins have appeared in a variety of different languages, under a variety of different names, meaning a variety of different (albeit related) things.

In the context of Bracha and Cook’s pioneering work, as well as most subsequent object-oriented instances of mixins, a mixin is an *abstract subclass*, *i.e.*, a subclass that is parameterized over an abstract specification of its superclass and can be instantiated to extend multiple different superclasses.

The other common meaning of mixins, which is less specific to object-oriented programming and is the one we are primarily interested in for the purposes of this paper, is also due to Bracha, in particular his work with Lindstrom on the Jigsaw language [5]. Jigsaw’s central construct is actually not called a mixin, but rather a *module*. Jigsaw modules may contain both *defined* components

(i.e., exports) and *declared* components (i.e., imports). The language provides a suite of operators for adapting and combining modules. Of particular note is the *merge* operator, which takes as input two modules, M_1 and M_2 , and returns a module that

1. has an import corresponding to any component that was an import of either M_1 or M_2 but not an export of the other, and
2. has an export corresponding to any component that was an export of either M_1 or M_2 but not both.

The typing rule for the merge operator checks that any components with the same name in M_1 and M_2 have compatible types, and that the exports of M_1 and M_2 are disjoint.

While the merge operator does not permit M_1 and M_2 to have overlapping exports, Bracha provides a separate *override* operator that does, choosing the export from M_2 over the export from M_1 in case of an overlap. In some later versions of mixins, a variant of the override operator, not the merge operator, is adopted as the default notion of mixin composition. Moreover, support for overriding is often considered a central feature of mixins. Be that as it may, for the remainder of this paper we will use the term *mixin composition* to mean Bracha’s merge operator. Following mixin-based languages like Flatt *et al.*’s *units* [15, 29], our MixML language does not attempt to support any form of overriding.

The work on Jigsaw has inspired a significant amount of research into *mixin module systems*. Over the course of several papers [3, 2, 1], Ancona and Zucca have explored in depth the semantic properties and algebraic laws of mixin operators, and developed a foundational mixin module calculus called CMS, which refactors some of the Jigsaw primitives. While CMS is a pure call-by-name language, it has been extended with support for call-by-value evaluation [18] and monadic effects [1].

Compared with ML modules, a key advantage of CMS-style mixin modules is that mixin composition *is* recursive linking, and seems to offer a natural solution to the problems with separate compilation of recursive modules in ML. One major limitation, however, is that CMS modules contain only term components, not type components, which means that they cannot express ML-style abstract data types, let alone translucent signatures.

Another limitation of mixin modules, which to our knowledge has not been observed previously, is that they are not hierarchically composable. To understand what we mean, consider the following example. Suppose M_1 and M_2 are ML structures, each of signature

```
sig val x : int; val y : int end
```

One can compose them hierarchically to form a new structure containing both:

```
module M = struct
  module A1 = M1; module A2 = M2
end
```

If M_1 and M_2 were CMS mixin modules, each with x as an import and y as an export, we might wish to hierarchically compose them in the same way, with the result being a new *mixin* module M , with imports $A1.x$ and $A2.x$, and exports $A1.y$ and $A2.y$. Yet this is not how mixins behave. The CMS equivalent of the above code would result in a *record* M with two mixin components, $A1$ and $A2$. The kind of hierarchical composition we are interested in is simply not possible with CMS mixins, as they employ a flat namespace for their imports and exports (pathnames like $A1.x$ are disallowed).

In all fairness to mixins, hierarchical composition is not a feature that ML functors enjoy directly either. (Were one to define $A1$ and $A2$ using functors, M would be a *structure* with two functor components, not a functor itself.) However, as we will demonstrate in Section 2, hierarchical composition of mixin modules is quite a useful feature, in fact a cornerstone of our entire approach.

1.3 MixML

In this paper, we present the design of a novel module system called MixML. A MixML module is like an ML structure in which some of the components are specified but not defined. In other words, it is like a cross between an ML structure and an ML signature. By combining elements of both ML modules and mixin modules, MixML modules seamlessly integrate

- the hierarchical composability of ML structures,
- the translucent data abstraction of ML signatures and sealing,
- and the recursive composability of mixin modules.

This offers clear advantages over both ML modules (which cannot be recursively linked) and mixin modules (which cannot be hierarchically composed and typically do not support type abstraction).

The design of MixML is deliberately minimalist, emphasizing how superficially distinct features of the ML module system can be modeled as stylized uses of the same underlying constructs, with mixin-style recursive composition playing a central role. Several examples of this are:

- ML structures and signatures are modeled in MixML by the same syntactic category—modules. An ML structure is just a MixML module with no imports, whereas an ML signature is a MixML module with no *dynamic* exports (it may have static exports corresponding to its transparent type specifications).
- There is essentially one variable binding construct in MixML—the recursive linking construct. Attaching variable binding to recursive linking allows us to uniformly model all the forms of variable binding that arise in both ordinary ML modules and recursive modules.
- To model ML’s sealing construct, MixML introduces an alternative form of linking: *opaque linking*, as opposed to the usual *transparent linking*. By viewing sealing as a special kind of linking, the great similarity between the static semantics of sealing and linking is brought into relief, revealing these features to be two sides of the same coin.

As a proof of concept, we have implemented a prototype interpreter for MixML, which we intend to make publicly available in the near future.

1.4 Overview

The rest of the paper is structured as follows. In Section 2, we present the syntax of MixML and lead the reader on a tour of the language by example. In Section 3, we present the formal details of the MixML type system, in particular the static semantics. (Due to space limitations, we only sketch the salient features of the dynamic semantics and soundness proof. Full details appear in a companion technical appendix, available by request through the PC chair.) Much of the key technical machinery is an adaptation and generalization of Dreyer’s recent work on a type system for recursive modules in the presence of data abstraction [8]. In Section 4, we discuss related and future work.

2. A Tour of MixML

The language of MixML modules unifies the structure and signature languages of ML into one, so that ML’s original notions of structure and signature are just special cases of a single mechanism. The syntax of MixML is displayed in Figure 1.

In a MixML module, some components may be defined (the exports), and some may have a kind or type specification but are not defined (the imports). The import components of a module can be viewed as requirements that will be fulfilled in the future when the module is linked with other modules. Thus, the MixML type

Kinds	$K \in \mathbb{N}$
Type Var's	$\alpha, \beta \in TyVars \times \mathbb{N}$
Module Var's	$X, Y \in ModVars$
Labels	$\ell \in Labs$
Label Paths	$\ell s ::= \epsilon \mid \ell.\ell s \in Paths$
Type Constr's	$tyc ::= Tyc(mod) \mid$ $\alpha \mid \lambda(\bar{\alpha}).tyc \mid tyc(\overline{tyc}) \mid \dots$
Terms	$exp ::= Val(mod) \mid \dots$
Modules	$mod ::= X \mid \{ \} \mid [:tyc] \mid [exp] \mid$ $[:K] \mid [tyc] \mid$ $\{ \ell = mod \} \mid mod.\ell \mid$ $(X = mod_1) \text{ with } mod_2 \mid$ $(X = mod_1) \text{ seals } mod_2 \mid$ $[mod] \mid \text{new } mod \mid [:usig] \mid$ $\{ : \ell \approx tyc \} \mid \{ \ell \approx tyc \}$
Unit Signatures	$usig ::= mod \text{ import } \bar{\ell}s \mid mod \text{ export } \bar{\ell}s$
	$mod_1 \text{ with } mod_2 \stackrel{\text{def}}{=} (X = mod_1) \text{ with } mod_2$
	$mod_1 \text{ seals } mod_2 \stackrel{\text{def}}{=} (X = mod_1) \text{ seals } mod_2$
	where $X \notin FV(mod_2)$

Figure 1. MixML Syntax

system insists that no module operator be permitted to remove the imports of a module from scope (*e.g.*, by the use of data abstraction), as one should not be allowed to forget about a requirement. In contrast, exports may always be hidden.

Types and Terms Following Leroy [21], we define our module language in a way that is largely agnostic with respect to the details of the core language. Of the term language we expect only that it contains a *term projection* construct $Val(mod)$, which takes an atomic term module mod (*i.e.*, a module containing a single term component) and projects out the term. Similarly, we assume that the type language contains a *type projection* construct $Tyc(mod)$, which takes an atomic type module mod (*i.e.*, a module containing a single type component) and projects out the type. Atomic modules are discussed in more detail below.

We also assume that the type language contains ML-style type constructors, which take a list of type arguments (the notation $\bar{\alpha}$ denotes a list of 0 or more α 's, separated by commas) and return a single type as a result. The kind of a type constructor is its arity, *i.e.*, the number of type arguments it has. Types classifying terms have kind 0.

Atomic Modules Atomic modules are modules containing a single (type or term) component, and that component may be either specified (*i.e.*, an import) or defined (*i.e.*, an export). Whereas, in ML, definitions only occur in modules, and specifications only occur in signatures, in MixML both definitions and specifications are module constructs.

The module $[:tyc]$ represents a term specification of type tyc . The module $[exp]$ represents a term component defined to be the result of evaluating exp . The module $[:K]$ represents an abstract type specification of kind K . The module $[tyc]$ represents a transparent definition of a type component equal to tyc . Note that, in ML, there is a distinction between transparent type *definitions*, which appear in modules, and transparent type *specifications*, which appear in signatures. In MixML, these mechanisms are unified into one.

Unary Structures and Projection The module $\{ \}$ represents an empty structure, containing no components. The module $\{ \ell = mod \}$ introduces a structure containing a single component named ℓ ,

whose definition is mod . Any imports (resp. exports) of mod become imports (resp. exports) of $\{ \ell = mod \}$ as well, except the pathnames of those imports (resp. exports) now have “ ℓ .” in front of them. Thus, MixML modules are *hierarchically composable*.

The constructs we have discussed so far can be combined to give a direct encoding of ML-style type/term definitions/specifications:

$$\begin{array}{ll}
\text{val } v : tyc & \stackrel{\text{def}}{=} \{v = [:tyc]\} \\
\text{val } v = exp & \stackrel{\text{def}}{=} \{v = [exp]\} \\
\text{type } (\alpha_1, \dots, \alpha_n) \ t & \stackrel{\text{def}}{=} \{t = [:n]\} \\
\text{type } (\bar{\alpha}) \ t = tyc & \stackrel{\text{def}}{=} \{t = [\lambda(\bar{\alpha}).tyc]\}
\end{array}$$

Dual to $\{ \ell = mod \}$ is the module $mod.\ell$, which projects the ℓ component from the structure mod . The typing rule for $mod.\ell$ insists that any imports mod may have must be contained in the ℓ component. This guarantees that no import requirements of mod are dropped when we project out the ℓ component.

At the moment, all we have are unary structures. In order to support n -ary structures and signatures of the sort found in ML, we now present MixML's most versatile construct—*linking*.

Linking The linking module construct $(X = mod_1) \text{ with } mod_2$ is MixML's primary means of composing multiple modules together. Linking does several things:

- It performs mixin composition of mod_1 and mod_2 in the style of Bracha's merge operator, one difference being that we permit mod_1 and mod_2 to both export a *type* component with the same pathname if the definitions they provide are equal.
- It sequences effects. Any definitions of term components in mod_1 will be evaluated prior to any such definitions in mod_2 .
- It is the primary means of variable binding in the language. It binds X as a representative of mod_1 inside mod_2 .

Before exploring the recursive aspects of linking, we first show how it may be used to express local module definitions, which we use in turn to encode polyadic non-recursive structures and signatures.

Local Module Definitions For several encodings it is useful to have a module-level let construct, $\text{let } X = mod_1 \text{ in } mod_2$. Fortunately, such a construct is definable in the language. The encoding makes use of two distinct labels ℓ_1 and ℓ_2 , which are arbitrary.

$$\text{let } X = mod_1 \text{ in } mod_2 \stackrel{\text{def}}{=} ((X = \{ \ell_1 = mod_1 \}) \text{ with } \{ \ell_2 = mod_2[X.\ell_1/X] \}).\ell_2$$

The encoding links together a structure containing ℓ_1 (defined to be mod_1) together with a structure containing ℓ_2 (defined to be mod_2). The binding of the first structure to X allows the second structure to refer to it. However, since mod_2 expects X to refer to mod_1 (not $\{ \ell_1 = mod_1 \}$), we must replace references to X in mod_2 with references to $X.\ell_1$. Finally, we project out ℓ_2 from the result, so that the only components in the resulting module are those of mod_2 —the components of mod_1 are hidden.

Since mod_1 is hidden, the MixML type system will insist that it be *complete*, *i.e.*, that it have no imports. This is a useful property to be able to enforce. Thus, in general, if we wish to check that a module mod is complete, we can do so by just let -expanding it:

$$\text{let } X = mod \text{ in } X$$

Polyadic Structures and Signatures While, in ML, components of a structure or signature are for convenience only assigned a single name, most type-theoretic accounts of the ML module system employ a *label-variable distinction* [16] (or the equivalent [20]). This divides the name of each component into an *label* ℓ , which is unchangeable and is used as the “external” name of the component, and a *variable* X , which is alpha-convertible and is used

as the “internal” name of the component within subsequent definitions/specifications of the structure/signature. Under this approach, an n -ary structure can be modeled as

$$\{\ell_1 \triangleright X_1 = mod_1, \dots, \ell_n \triangleright X_n = mod_n\}$$

where each X_i is bound in the subsequent M_j ’s to the result of evaluating M_i .

Using the `let` construct defined above, the encoding of n -ary structures is as follows (assuming X is a suitably fresh variable):

$$\{\ell_1 \triangleright X_1 = mod_1, \dots\} \stackrel{\text{def}}{=} (X = \{\ell_1 = mod_1\}) \text{ with } (\text{let } X_1 = X.\ell_1 \text{ in } \{\dots\})$$

The encoding defines an n -ary structure as the linking of a unary structure, containing just the first component ℓ_1 , with a structure containing the rest of the components. Inside the linking, X stands for the structure containing just ℓ_1 . The `let` allows us to then define X_1 as shorthand for $X.\ell_1$ within the remainder of the structure. (Note that we assume here for simplicity that all the ℓ_i are distinct labels. There are well-known ways of allowing for shadowing [17], but they are beyond the scope of this paper.)

Typically, ML module type systems model n -ary signatures in a similar fashion to n -ary structures (yet as a distinct construct):

$$\{\ell_1 \triangleright X_1 : sig_1, \dots, \ell_n \triangleright X_n : sig_n\}$$

However, since ML signatures are encoded in MixML as modules (i.e., a *sig* is just a MixML module with no term exports), the encoding of n -ary signatures is exactly the same as for n -ary structures—just change the colons to equal signs. In essence, module specifications and module definitions are one and the same mechanism.

Adding Type Definitions to Signatures So far we have only shown examples of linking in which the namespaces of the linked modules were disjoint, but of course one of the whole points of mixin linking is to link modules with overlapping namespaces. One instance where this is useful is in modeling ML’s `with type` (or `where type`) mechanism for adding type definitions to signatures. The ML construct

$$sig \text{ with type } (\bar{\alpha}) \ t = tyc$$

can be modeled (quite directly!) as a form of linking:

$$sig \text{ with type } (\bar{\alpha}) \ t = tyc$$

where “`type` $(\bar{\alpha}) \ t = tyc$ ” is encoded as on the previous page.

While the above encoding does not exploit the ability to bind *sig* to a variable, that ability definitely adds some expressive power. For example, if *sig* contains two abstract type components u and t , and we wish to modify the signature so that t is transparently equal to u , the traditional ML `with type` construct does not permit this because u is not a valid type outside of the signature. With the above encoding, however, assuming *sig* is bound to X , we could define t to be $X.u$. (This is similar to a proposal of Ramsey *et al.* [30], but in our case the added functionality falls out directly from our module-linking-cum-variable-binding construct.)

Recursive Modules A more complex use of mixin linking occurs in the encoding of recursive modules. Recursive module extensions to ML typically have the form:

$$\text{rec } (X : sig) \ mod$$

Here, X is the variable by which *mod* refers to itself recursively, and *sig* is the *forward declaration*, a kind of template for *mod*, which serves as the signature of X during the typechecking of *mod*. The encoding of this construct in MixML is extremely simple:

$$(X = sig) \ \text{with } \ mod$$

Mixin linking will use the type definitions (type exports) of *mod* to fill in the corresponding abstract type specifications (type imports) of *sig*, and then check that the term definitions (term exports) of *mod* match the types from the corresponding term specifications (term imports) of *sig*.

One thing this encoding will not do in its present form is ensure that all the components forward-declared in *sig* actually get defined by *mod*. Any components that *mod* fails to define will just remain imports in the result of the linking. To ensure completeness, though, we can simply `let`-expand it as explained above.

Recursively Dependent Signatures All the existing recursive module proposals for ML also extend the signature language with a new construct called a *recursively dependent signature* [7]. In Russo’s extension to Moscow ML [32], it takes the form:

$$\text{rec } (X : sig_1) \ sig_2$$

This construct allows the signatures of mutually recursive modules (in *sig₂*) to refer recursively to each other’s type components through the variable X . Of course, since modules and signatures are both encoded in MixML as modules, this construct is encoded in the exact same way as the recursive module construct:

$$(X = sig_1) \ \text{with } \ sig_2$$

One point of note is that not all recursive module extensions to ML require the programmer to write down *sig₁*. Instead, they infer it from *sig₂*. We view such an inference step as a separable convenience. In any case, this encoding demonstrates that recursive modules and recursively dependent signatures can be understood as two stylized uses of the same mechanism.

Opaque Sealing is Opaque Linking None of the MixML constructs described thus far supports the creation of abstract data types. For this purpose MixML includes a second variant of the linking construct— $(X = mod_1) \ \text{seals } mod_2$ —which we call *opaque linking* (as opposed to the original form, which we call *transparent linking*).

Opaque linking is very similar to transparent linking, *except*:

- The only information that the rest of the program may know about the result of opaque linking is what it can tell from looking at *mod₁*—no information about *mod₂* may be revealed.

What this property means in effect is that any type imports of *mod₁* for which *mod₂* provides definitions will become *abstract type exports* of the linked module. Furthermore, *mod₂* must define *all* of *mod₁*’s imports. If *mod₂* were only to define some of them, we would have no way of knowing which components of the linked module were exports and which were imports without looking at *mod₂* (and thus violating data abstraction).

Using opaque linking, we arrive at a simple encoding of ML’s opaque sealing (or signature matching) construct. Specifically:

$$mod :> sig \stackrel{\text{def}}{=} sig \ \text{seals } mod$$

Transparent Signature Matching Standard ML’s *transparent signature matching* construct, written $mod : sig$, is also easily encodable, using (quite naturally) *transparent linking*. Transparent matching is similar to opaque matching in that it results in a module containing only the components specified in *sig*, but transparent matching does not create any abstract types—it merely fills in the definitions of any abstract type components in *sig* with their definitions in *mod*.

$$mod : sig \stackrel{\text{def}}{=} (\{\text{Src} \triangleright X = sig, \text{Tgt} = X\} \ \text{with } \ \{\text{Src} = mod\}).\text{Tgt}$$

The trick here is not simply to link *mod* and *sig*, for that would result in a module containing *all* the components of *mod* (including

ones not specified in *sig*). Instead, we link *mod* and *sig* under the module name *Src*, but before doing so we copy *Src* (bound to *sig*) to another name *Tgt*. Thus, even after we link the two *Src*'s, *Tgt* still contains a copy of *Src* in the shape that it had originally.

In order for this encoding to work, it is important that references to module variables are treated *lazily*. When *Tgt* is defined to be *X*, this does not result in an attempt to evaluate all the term components of *X* to values, for that would fail if *sig* contained even a single term specification! Rather, *Tgt*'s definition creates “links” to (*i.e.*, delayed copies of) the components of *Src*. For more details, see the discussion of MixML's dynamic semantics in Section 3.3.

Double Vision An important problem that arises in extending ML with recursive modules is the *double vision problem* [8]. Consider the following example:

$$(X = \{t = [:0], \dots\}) \text{ seals } \{t = [\text{int}], \dots\}$$

Here, we are defining a recursive sealed module with a type component *t* that is defined internally to be *int*. Within the r.h.s. of the *seals*, we know that *t* is implemented as *int*, so we ought to know that *X.t* (which is just a recursive reference to *t*) equals *int*, but the signature bound to *X* does not reflect this.

Fortunately, Dreyer has recently shown how to solve this problem [11]. His solution involves performing two passes when type-checking a recursive module. The first pass is a “static” pass, which computes the type components of the module (*e.g.*, discovering that *t* in the above example is implemented as *int*). The information from the static pass is then incorporated into the typing context (*e.g.*, changing *X*'s signature to reflect that *t* is *int*) prior to the second pass, which fully typechecks the module.

We generalize Dreyer's two-pass technique to work in the context of MixML's more general linking construct, but basically our approach is very similar to his. See Section 3.2 for details.

Units So far we have not introduced any means of *suspending* a module in the manner of an ML functor. To support this important feature, we introduce a new atomic module construct we call a *unit*. (As we explain in Section 4, our units are inspired by Flatt *et al.*'s units [15, 29], but are different in many respects.)

A unit, written [*mod*], is a suspension of the module *mod*. With units we can encode an ML functor (modeled here by a module-level λ -expression) as follows:

$$\lambda(X:\text{sig}).\text{mod} \stackrel{\text{def}}{=} [\{\text{Arg} > X = \text{sig}, \text{Res} = \text{mod}\}]$$

In other words, a functor is just a suspension of a module with one component *Arg* whose term (and possibly type) components are undefined, and one component *Res* that is fully defined. Unlike ML functors, though, units need not be so rigidly structured. Any module can be turned into a unit.

The elimination construct for units is written *new mod*. Here, *mod* is assumed to be a unit, and *new mod* has the effect of *instantiating* the unit by turning it back into a module. For example, suppose that the variable *F* has been bound to the functor expression shown above. Application of *F* to an argument *mod* is encoded as follows:

$$F(\text{mod}) \stackrel{\text{def}}{=} (\{\text{Arg} = \text{mod}\} \text{ with } \text{new } F).\text{Res}$$

The reason we put *new F* on the r.h.s. of the linking is to ensure that the term definitions in *mod* are evaluated before the term definitions in *F*, which may depend on them.

Note that every instantiation of a unit *F* generates a distinct instance of the module expression contained within *F*. In particular, each occurrence of *new F* will re-evaluate the term definitions in *F*'s constituent module and generate fresh abstract types corresponding to said module's abstract type exports. In this respect, unit instantiation is much like generative functor application in Standard

ML. We do not currently model the *applicative* behavior of functors in OCaml [19].

Signature Bindings In addition to modeling functors, units can be used to model ML's *signature* bindings. Suppose *sig* is an ML signature encoded as a MixML module, and that we wish to bind it to a *signature variable* *S* (to use as shorthand for *sig* in subsequent code). In MixML, we would bind *S* to [*sig*], thus encapsulating *sig* into a unit. Then, at any subsequent use of *S* we would write *new S* to get back the original *sig*. Note that, since ML signatures contain neither term definitions nor abstract type definitions (only specifications), performing *new* on a signature variable will never have any computational effects.

Separate Compilation of Recursive Modules At the start of the paper, the main criticism we gave of ML modules was that they do not support separate compilation of mutually recursive modules. In MixML, this functionality is provided by units.

Suppose we wish to define two modules named *A* and *B*, with signatures *sig_A* and *sig_B*, and definitions *mod_A* and *mod_B*, which refer recursively to themselves and to each other through the module variable *X*. For simplicity, let us assume that *sig_A* and *sig_B* do not refer to *X*.¹ Let the signature variable *S* be bound to the unit

$$[\{A = \text{sig}_A, B = \text{sig}_B\}]$$

Were we to write *A*'s and *B*'s definitions together, the MixML code would be:

$$(X = \text{new } S) \text{ with } \{A = \text{mod}_A, B = \text{mod}_B\}$$

But there is no need to define *A* and *B* together. We can instead first bind *U_A* to

$$[(X = \text{new } S) \text{ with } \{A = \text{mod}_A\}]$$

and *U_B* to

$$[(X = \text{new } S) \text{ with } \{B = \text{mod}_B\}]$$

The units *U_A* and *U_B* represent the separately compiled versions of *A* and *B*, respectively. *U_A* exports definitions for the components of *A*, but leaves *B*'s components as imports, and *U_B* is vice versa. Finally, when we want to link them we simply write:

$$\text{new } U_A \text{ with new } U_B$$

Of course, there is nothing requiring us to link *U_A* and *U_B* in this order or with each other. They are completely independent program units that can link with any other units that match their imports.

Higher-Order Units With the constructs presented so far, units can only be defined and *exported* from other units. If we wish to support the expressiveness of higher-order functors (a feature in many dialects of ML), then the language also has to support the specification of units as *imports*. For this purpose, we introduce the atomic module [*: usig*], which specifies such a unit import. Here, *usig* is a new syntactic construct—the *unit signature*.

A unit signature *usig* takes one of two symmetric forms, written *mod import* \overline{ls} and *mod export* \overline{ls} . In both forms, *mod* is a MixML module representing an ML signature, *i.e.*, it must have neither term exports nor abstract type exports. The *import* and *export* clauses serve to identify which components specified in *mod* are to be treated as imports and which as exports in the unit that the *usig* is describing. In the case of *mod import* \overline{ls} , the list \overline{ls} of paths enumerates all components of *mod* that are to be considered imports, treating all others as exports. Conversely, *mod export* \overline{ls} lists the exports, and treats all other components as imports. A path $ls \in \overline{ls}$ may point to an entire structure in *mod*,

¹ This restriction is only in place to make the example shorter. We can easily circumvent it through the use of recursively dependent signatures.

in which case the annotation applies to all subcomponents of that structure.

For example, we can assign the following unit signature to the unit U_A from the previous section:

$$S \text{ export } A$$

Alternatively, the same unit signature is expressed by

$$S \text{ import } B$$

We provide both forms merely as a convenience.

The encoding of the functor F given earlier can be described with a unit signature as follows (assuming that sig' is a suitable specification of its body mod):

$$\{\text{Arg} \triangleright X = sig, \text{Res} = sig'\} \text{ import Arg}$$

This corresponds to the ML functor signature $(X : sig) \rightarrow sig'$.

Linking a unit definition $[mod]$ against a unit specification $[: usig]$ involves higher-order width-and-depth subtyping via a suitable notion of matching on unit signatures. In particular, unit signature matching allows the target signature to have fewer exports and more imports than the source signature.

Cyclic Definitions Since linking is recursive, it can obviously introduce cycles between the definitions of term components. We assume a call-by-value semantics for the evaluation of term components in MixML, which means that cyclic linking can potentially cause a run time exception when a component is accessed that has not yet been evaluated. Static detection of such errors is a problem that is largely orthogonal to our work and has been addressed by Hirschowitz and Leroy [18] and Dreyer [9] among others.

Linking also potentially introduces cycles among type definitions. Supporting cyclic type definitions would require the introduction of higher-kinded *equi-recursive* types [7] into the type system, which are not well understood. MixML hence forbids cyclic type definitions. Unfortunately, in the presence of type abstraction and separate compilation, violation of this restriction cannot always be detected statically. For that reason, a dynamic occurs check is performed when a type definition is evaluated, and dynamic failure can occur in the same manner as for value definitions.

To support recursive type definitions, MixML separately provides ML-style datatypes. Similar to the encoding given by Harper and Stone [17] and Dreyer [8], a module $\{\ell \approx tyc\}$ provides an abstract type named ℓ that is isomorphic to tyc . Values of this type must be constructed and deconstructed by a pair of explicit injection and projection operations (named ℓ_{in} and ℓ_{out}) that come along with the type. Datatypes can also be specified as imports using a datatype specification $\{\ell \approx tyc\}$. Through linking, datatypes can be defined recursively. Cycles among type definitions are then permitted as long as they go through a datatype.

3. The MixML Type System

3.1 Semantic Objects

The MixML type system is based to a large extent on the RMC type system for recursive modules introduced by Dreyer [8]. RMC in turn inherits many aspects from the Definition of Standard ML [24]. In particular, it represents the types of modules by *semantic objects*. As in RMC, our semantic objects—shown in Figure 2—are essentially signatures from a simpler “internal” type system, enriched with annotations guiding typechecking.² These semantic signatures (Σ) include structure signatures, as well as three forms of atomic signature (for type, term, and unit components). Our semantic objects differ from RMC’s in that atomic term and unit signatures

²The internal type system is defined in the companion technical appendix, but is not required in order to understand the static semantics of MixML.

Type Constructors	$A, B, \tau ::= \alpha \mid b \mid \lambda(\bar{\alpha}).\tau \mid A(\bar{\tau})$
Base Types	$b ::= \forall[\bar{\alpha}].\tau_1 \Rightarrow \tau_2 \mid \dots$
Module Signatures	$\Sigma ::= \llbracket = A \rrbracket \mid \llbracket \tau \rrbracket^\pm \mid \llbracket \Phi \rrbracket^\pm \mid \{\ell : \Sigma\}$
Unit Signatures	$\Phi ::= \forall\bar{\alpha}. \exists\bar{\beta}. (\mathcal{L}_1; \mathcal{L}_2; \Sigma)$
Type Substitutions	$\delta ::= \{\alpha \mapsto A\}$
Type Locators	$\mathcal{L} ::= \llbracket = A \rrbracket \mid \{\ell : \mathcal{L}\}$
Module Contexts	$\Gamma ::= \emptyset \mid \Gamma, X : \Sigma$

$\forall\bar{\alpha}. \exists\bar{\beta}. (\mathcal{L}; \Sigma)$	$\stackrel{\text{def}}{=} \forall\bar{\alpha}. \exists\bar{\beta}. (\mathcal{L}; \mathcal{L}'; \Sigma)$ for some \mathcal{L}'
$\text{prefix}?(ls, ls)$	$\stackrel{\text{def}}{=} \text{true}$ if $\exists ls_1 \in \bar{ls}. \exists ls_2. \text{s.t. } ls = ls_1.l_{s_2}$
$\{\ell : A \approx B\}^\pm$	$\stackrel{\text{def}}{=} \{\ell : \llbracket = A \rrbracket, \ell_{in} : \llbracket \forall[\bar{\beta}]. B(\bar{\beta}) \Rightarrow A(\bar{\beta}) \rrbracket^\pm, \ell_{out} : \llbracket \forall[\bar{\beta}]. A(\bar{\beta}) \Rightarrow B(\bar{\beta}) \rrbracket^\pm\}$

$\Sigma.l_s$	$\stackrel{\text{def}}{=} \begin{cases} \Sigma & \text{if } l_s = \epsilon \\ \Sigma' & \text{if } l_s = l_s'.l \\ & \text{and } \Sigma.l_s' = \{\dots, l : \Sigma', \dots\} \\ \text{undefined} & \text{otherwise} \end{cases}$
$\Sigma(l_s)$	$\stackrel{\text{def}}{=} A$ if $\Sigma.l_s = \llbracket = A \rrbracket$
$\mathcal{L}(\alpha)$	$\stackrel{\text{def}}{=} l_s$ if $\mathcal{L}.l_s = \alpha$ and $\nexists l_s'$ such that $\mathcal{L}.l_s' = \alpha$
$\mathcal{L} \subseteq \Sigma$	$\stackrel{\text{def}}{=} \forall l_s, A. \mathcal{L}(l_s) = A \Rightarrow \Sigma(l_s) = A$
$\mathcal{L}_1 \cup \mathcal{L}_2$	$\stackrel{\text{def}}{=} \mathcal{L}$ such that $\forall l_s, A. \mathcal{L}(l_s) = A \Leftrightarrow (\mathcal{L}_1(l_s) = A \vee \mathcal{L}_2(l_s) = A)$

Figure 2. Semantic Objects and Auxiliary Definitions

are annotated with variances, in order to denote whether they are imports (−) or exports (+). The import/export distinction for type components is handled differently, as we explain below.

Unit signatures (Φ) contain *type locators* \mathcal{L}_1 and \mathcal{L}_2 , respectively mapping the import types $\bar{\alpha}$ and abstract export types $\bar{\beta}$ to paths in Σ . Locators are used to look up type definitions during linking. However, the export locator \mathcal{L}_2 is only needed when representing the translation of MixML-level *usig*’s; we omit it in other places. Unlike in RMC, we choose to express locators as a syntactic subcategory of signatures, which conveniently allows the sharing of meta-notation (given in Figure 2) for both kinds of objects. In addition, our syntax of locators allows the components of the locator to be arbitrary type constructors, not just variables.

We will refer to \mathcal{L} as a locator only if all its type components are distinct type variables, thus representing a bijective function between type variables and label paths. Otherwise we call it a *realizer*, which is a function from paths to types. Well-formed unit signatures are required to contain locators, but in other places the type system employs general realizers. We implicitly identify all realizers that represent the same mapping from paths to types, in effect ignoring differences with respect to empty substructures.

Semantic core-level types are the same as in RMC. For convenience, we assume that the set of type variables is partitioned into different kinds. This allows us to drop kind annotations from types and type variables, since they can always be derived syntactically. We write $\vdash A : K$ to assert that a constructor A has kind K . For type substitutions δ we demand implicitly that they are kind preserving. We also assume and maintain the invariant that types are kept in β -normal form. Substitutions are implicitly β -normalizing. Finally, we assume the existence of an ordering $<_{Paths}$ on paths.

3.2 Typing Rules

Figures 3 and 4 show the typing rules for MixML.

Rules 1 through 6 for (core) types and terms are standard. Projection ($Tyc(mod)$ or $Val(mod)$) requires the module mod to be complete, *i.e.*, have no pending imports. This is ensured by Rule 7, which checks that mod neither has static imports (by

Type Constructors: $\Gamma \vdash \text{tyc} \rightsquigarrow A \langle : K \rangle$

$$\frac{\Gamma \vdash \text{mod} : \llbracket =A \rrbracket}{\Gamma \vdash \text{Typ}(\text{mod}) \rightsquigarrow A} \quad (1) \quad \frac{\vdash \alpha : 0}{\Gamma \vdash \alpha \rightsquigarrow \alpha} \quad (2) \quad \frac{\Gamma \vdash \text{tyc} \rightsquigarrow \tau : 0 \quad \bar{\alpha} \text{ fresh}}{\Gamma \vdash \lambda(\bar{\alpha}).\text{tyc} \rightsquigarrow \lambda(\bar{\alpha}).\tau} \quad (3)$$

$$\frac{\Gamma \vdash \text{tyc}' \rightsquigarrow \lambda(\bar{\alpha}).\tau' : n \quad \Gamma \vdash \text{tyc} \rightsquigarrow \tau : 0 \quad \bar{\tau} = \tau_1, \dots, \tau_n}{\Gamma \vdash \text{tyc}'(\overline{\text{tyc}}) \rightsquigarrow \{\bar{\alpha} \mapsto \bar{\tau}\}\tau'} \quad (4) \quad \frac{\Gamma \vdash \text{tyc} \rightsquigarrow A \quad \vdash A : K}{\Gamma \vdash \text{tyc} \rightsquigarrow A : K} \quad (5)$$

Core-Language Expressions: $\Gamma \vdash \text{exp} : \tau$

$$\frac{\Gamma \vdash \text{mod} : \llbracket \tau \rrbracket^+}{\Gamma \vdash \text{val}(\text{mod}) : \tau} \quad (6)$$

Complete Modules and Units: $\Gamma \vdash \text{mod} : \Sigma \quad \Gamma \vdash \text{mod} : \Phi$

$$\frac{\Gamma; \{\!\!\}; \bar{\beta} \vdash \text{mod} : |\Sigma| \quad \bar{\beta} \text{ fresh} \quad \bar{\beta} \notin \text{FV}(\Sigma)}{\Gamma \vdash \text{mod} : |\Sigma|} \quad (7) \quad \frac{\Gamma; \mathcal{L}; \bar{\beta} \vdash \text{mod} : \Sigma \quad \vdash \mathcal{L} \text{ locates } \bar{\alpha} \quad \bar{\alpha}, \bar{\beta} \text{ fresh}}{\Gamma \vdash \text{mod} : \forall \bar{\alpha}. \exists \bar{\beta}. (\mathcal{L}; \Sigma)} \quad (8)$$

Modules: $\Gamma; \mathcal{L}; \bar{\beta} \vdash \text{mod} : \Sigma$

$$\frac{X : \Sigma \in \Gamma}{\Gamma; \{\!\!\}; \emptyset \vdash X : |\Sigma|} \quad (9) \quad \frac{}{\Gamma; \{\!\!\}; \emptyset \vdash \{\} : \{\!\!\}} \quad (10) \quad \frac{\vdash A : K}{\Gamma; \llbracket =A \rrbracket; \emptyset \vdash \llbracket :K \rrbracket : \llbracket =A \rrbracket} \quad (11) \quad \frac{\Gamma \vdash \text{tyc} \rightsquigarrow A}{\Gamma; \{\!\!\}; \emptyset \vdash \llbracket \text{tyc} \rrbracket : \llbracket =A \rrbracket} \quad (12)$$

$$\frac{\Gamma \vdash \text{tyc} \rightsquigarrow \tau : 0}{\Gamma; \{\!\!\}; \emptyset \vdash \llbracket : \text{tyc} \rrbracket : \llbracket \tau \rrbracket^-} \quad (13) \quad \frac{\Gamma \vdash \text{exp} : \tau}{\Gamma; \{\!\!\}; \emptyset \vdash \llbracket \text{exp} \rrbracket : \llbracket \tau \rrbracket^+} \quad (14)$$

$$\frac{\Gamma; \mathcal{L}; \bar{\beta} \vdash \text{mod} : \Sigma}{\Gamma; \{\!\!\}; \mathcal{L}; \bar{\beta} \vdash \{\ell = \text{mod}\} : \{\!\!\}; \Sigma} \quad (15) \quad \frac{\Gamma; \{\!\!\}; \mathcal{L}; \bar{\beta} \vdash \text{mod} : \{\!\!\}; \Sigma, \bar{\ell}' : |\Sigma'| \{\!\!\}}{\Gamma; \mathcal{L}; \bar{\beta} \vdash \text{mod}.\ell : \Sigma} \quad (16)$$

$$\frac{\vdash \mathcal{L}_1 \text{ locates } \bar{\alpha}_1 \quad \Gamma; \mathcal{L}_1 \uplus \mathcal{L}'_1; \bar{\beta}_1 \vdash \text{mod}_1 : \Sigma_1 \quad \vdash \mathcal{L}_2 \text{ locates } \bar{\alpha}_2 \quad \Gamma, X : \Sigma_1; \mathcal{L}_2 \uplus \mathcal{L}'_2; \bar{\beta}_2 \vdash_{\text{stat}} \text{mod}_2 : \Sigma_2 \quad \bar{\alpha}_1, \bar{\alpha}_2 \text{ fresh} \quad \vdash (\mathcal{L}_1; \Sigma_1) \rightleftharpoons (\mathcal{L}_2; \Sigma_2) \rightsquigarrow \delta \quad \Gamma, X : \delta \Sigma_1; \delta \mathcal{L}_2 \uplus \mathcal{L}'_2; \bar{\beta}_2 \vdash \text{mod}_2 : \Sigma'_2 \quad \vdash \delta \Sigma_1 + \Sigma'_2 \Rightarrow \Sigma}{\Gamma; \mathcal{L}'_1 \cup \mathcal{L}'_2; \bar{\beta}_1, \bar{\beta}_2 \vdash (X = \text{mod}_1) \text{ with } \text{mod}_2 : \Sigma} \quad (17)$$

$$\frac{\vdash \mathcal{L}_1 \text{ locates } \bar{\alpha}_1 \quad \Gamma; \mathcal{L}_1; \bar{\beta}_1 \vdash \text{mod}_1 : \Sigma_1 \quad \vdash \mathcal{L}_2 \text{ locates } \bar{\alpha}_2 \quad \Gamma, X : \Sigma_1; \mathcal{L}_2; \bar{\beta}_2 \vdash_{\text{stat}} \text{mod}_2 : \Sigma_2 \quad \bar{\alpha}_2, \bar{\beta}_2 \text{ fresh} \quad \vdash (\mathcal{L}_1; \Sigma_1) \rightleftharpoons (\mathcal{L}_2; \Sigma_2) \rightsquigarrow \delta \quad \delta \Gamma, X : \delta \Sigma_1; \delta \mathcal{L}_2; \bar{\beta}_2 \vdash \text{mod}_2 : \Sigma'_2 \quad \vdash \delta \Sigma_1 + \Sigma'_2 \Rightarrow |\Sigma|}{\Gamma; \{\!\!\}; \bar{\beta}_1, \bar{\alpha}_1 \vdash (X = \text{mod}_1) \text{ seals } \text{mod}_2 : |\Sigma_1|} \quad (18)$$

$$\frac{\Gamma \vdash \text{tyc} \rightsquigarrow B : K \quad \vdash A : K}{\Gamma; \{\!\!\}; \llbracket =A \rrbracket; \emptyset \vdash \{\ell \approx \text{tyc}\} : \{\!\!\}; A \approx B}^- \quad (19) \quad \frac{\Gamma \vdash \text{tyc} \rightsquigarrow B : K \quad \vdash \beta : K}{\Gamma; \{\!\!\}; \beta \vdash \{\ell \approx \text{tyc}\} : \{\!\!\}; \beta \approx B}^+ \quad (20)$$

$$\frac{\Gamma \vdash \text{mod} : \Phi}{\Gamma; \{\!\!\}; \emptyset \vdash \llbracket \text{mod} \rrbracket : \llbracket \Phi \rrbracket^+} \quad (21)$$

$$\frac{\Gamma \vdash \text{mod} : \llbracket \forall \bar{\alpha}. \exists \bar{\beta}. (\mathcal{L}; \Sigma) \rrbracket^+ \quad \text{dom}(\delta) = \{\bar{\alpha}, \bar{\beta}\}}{\Gamma; \delta \mathcal{L}; \delta \bar{\beta} \vdash \text{new mod} : \delta \Sigma} \quad (22)$$

$$\frac{\Gamma \vdash \text{usig} \rightsquigarrow \Phi}{\Gamma; \{\!\!\}; \emptyset \vdash \llbracket : \text{usig} \rrbracket : \llbracket \Phi \rrbracket^-} \quad (23)$$

Unit Signatures: $\Gamma \vdash \text{usig} \rightsquigarrow \Phi$

$$\frac{\Gamma \vdash \text{mod} : \forall \bar{\alpha}. \exists \emptyset. (\mathcal{L}; \Sigma') \quad |\Sigma| = |\Sigma'| = -\Sigma' \quad \mathcal{L} = \mathcal{L}_1 \uplus \mathcal{L}_2 \quad \vdash \mathcal{L}_1 \text{ locates } \bar{\alpha}_1 \quad \vdash \mathcal{L}_2 \text{ locates } \bar{\alpha}_2 \quad \{\bar{\alpha}_1\} = \{\alpha \mid \text{prefix?}(\bar{\ell}s, \mathcal{L}(\alpha))\} \quad \forall ls : \Sigma'. ls = \llbracket \tau \rrbracket^- \text{ or } \llbracket \Phi \rrbracket^- \Rightarrow (\Sigma.ls = \Sigma'.ls \Leftrightarrow \text{prefix?}(\bar{\ell}s, ls)) \quad \forall ls \in \bar{\ell}s : \Sigma.ls \text{ is defined}}{\Gamma \vdash \text{mod import } \bar{\ell}s \rightsquigarrow \forall \bar{\alpha}_1. \exists \bar{\alpha}_2. (\mathcal{L}_1; \mathcal{L}_2; \Sigma)} \quad (24)$$

$$\frac{\Gamma \vdash \text{mod import } \bar{\ell}s \rightsquigarrow \forall \bar{\alpha}_1. \exists \bar{\alpha}_2. (\mathcal{L}_1; \mathcal{L}_2; \Sigma)}{\Gamma \vdash \text{mod export } \bar{\ell}s \rightsquigarrow \forall \bar{\alpha}_2. \exists \bar{\alpha}_1. (\mathcal{L}_2; \mathcal{L}_1; -\Sigma)} \quad (25)$$

Figure 3. Typing Rules for MixML

$$\begin{array}{llll} \llbracket = A \rrbracket \stackrel{\text{def}}{=} \llbracket = A \rrbracket & \llbracket \Phi \rrbracket^\pm \stackrel{\text{def}}{=} \llbracket \Phi \rrbracket^\pm & -\llbracket = A \rrbracket \stackrel{\text{def}}{=} \llbracket = A \rrbracket & -\llbracket \Phi \rrbracket^\pm \stackrel{\text{def}}{=} \llbracket \Phi \rrbracket^\mp \\ \llbracket \tau \rrbracket^\pm \stackrel{\text{def}}{=} \llbracket \tau \rrbracket^\pm & \llbracket \ell : \Sigma \rrbracket \stackrel{\text{def}}{=} \llbracket \ell : \Sigma \rrbracket & -\llbracket \tau \rrbracket^\pm \stackrel{\text{def}}{=} \llbracket \tau \rrbracket^\mp & -\llbracket \ell : \Sigma \rrbracket \stackrel{\text{def}}{=} \llbracket \ell : -\Sigma \rrbracket \end{array}$$

Type Locators: $\vdash \mathcal{L} \text{ locates } \bar{\alpha}$

$$\frac{\forall \ell s. \mathcal{L}(\ell s) = A \Rightarrow A \in \bar{\alpha} \quad \forall \alpha \in \bar{\alpha}. \exists \text{ unique } \ell s. \mathcal{L}(\ell s) = \alpha \quad \mathcal{L}(\alpha_i) <_{\text{Paths}} \mathcal{L}(\alpha_j) \text{ (for } i < j)}{\vdash \mathcal{L} \text{ locates } \bar{\alpha}} \quad (26)$$

Bidirectional Type Lookup: $\vdash (\mathcal{L}_1; \Sigma_1) \rightleftarrows (\mathcal{L}_2; \Sigma_2) \rightsquigarrow \delta$

$$\frac{\delta_0 = \{\} \quad (\Sigma_2 \circ \mathcal{L}_1) \uplus (\Sigma_1 \circ \mathcal{L}_2) = \{\alpha_1 \mapsto A_1, \dots, \alpha_n \mapsto A_n\} \quad \forall i, j \text{ (s.t. } i \leq j) \in 1..n : \alpha_j \notin \text{FV}(A_i) \quad \delta_i = \delta_{i-1} \uplus \{\alpha_i \mapsto \delta_{i-1} A_i\}}{\vdash (\mathcal{L}_1; \Sigma_1) \rightleftarrows (\mathcal{L}_2; \Sigma_2) \rightsquigarrow \delta_n} \quad (27)$$

Signature Merging: $\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma$

$$\frac{}{\vdash \Sigma_2 + \Sigma_1 \Rightarrow \Sigma} \quad (28) \quad \frac{}{\vdash \llbracket = A \rrbracket + \llbracket = A \rrbracket \Rightarrow \llbracket = A \rrbracket} \quad (29) \quad \frac{\vdash \tau_1 \leq \tau_2}{\vdash \llbracket \tau_1 \rrbracket^\pm + \llbracket \tau_2 \rrbracket^\mp \Rightarrow \llbracket \tau_1 \rrbracket^\pm} \quad (30)$$

$$\frac{\vdash \Phi_1 \leq \Phi_2}{\vdash \llbracket \Phi_1 \rrbracket^+ + \llbracket \Phi_2 \rrbracket^- \Rightarrow \llbracket \Phi_1 \rrbracket^+} \quad (31) \quad \frac{}{\vdash \llbracket \Phi \rrbracket^- + \llbracket \Phi \rrbracket^- \Rightarrow \llbracket \Phi \rrbracket^-} \quad (32) \quad \frac{}{\vdash \Sigma + \{\} \Rightarrow \Sigma} \quad (33)$$

$$\frac{\ell \notin \bar{\ell}_2 \quad \vdash \llbracket \ell_1 : \Sigma_1 \rrbracket + \llbracket \ell_2 : \Sigma_2 \rrbracket \Rightarrow \llbracket \ell_3 : \Sigma_3 \rrbracket}{\vdash \llbracket \ell : \Sigma, \ell_1 : \Sigma_1 \rrbracket + \llbracket \ell_2 : \Sigma_2 \rrbracket \Rightarrow \llbracket \ell : \Sigma, \ell_3 : \Sigma_3 \rrbracket} \quad (34) \quad \frac{\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma_3 \quad \vdash \llbracket \ell_1 : \Sigma'_1 \rrbracket + \llbracket \ell_2 : \Sigma'_2 \rrbracket \Rightarrow \llbracket \ell_3 : \Sigma'_3 \rrbracket}{\vdash \llbracket \ell : \Sigma_1, \ell_1 : \Sigma'_1 \rrbracket + \llbracket \ell : \Sigma_2, \ell_2 : \Sigma'_2 \rrbracket \Rightarrow \llbracket \ell : \Sigma_3, \ell_3 : \Sigma'_3 \rrbracket} \quad (35)$$

Unit Signature Matching: $\vdash \Phi_1 \leq \Phi_2$

$$\frac{\vdash (\mathcal{L}_{11}; \Sigma_1) \rightleftarrows (\mathcal{L}_{22}; \Sigma_2) \rightsquigarrow \delta \quad \vdash \delta \Sigma_1 + -\delta \Sigma_2 \Rightarrow |\Sigma|}{\vdash \forall \bar{\alpha}_1. \exists \bar{\beta}_1. (\mathcal{L}_{11}; \mathcal{L}_{12}; \Sigma_1) \leq \forall \bar{\alpha}_2. \exists \bar{\beta}_2. (\mathcal{L}_{21}; \mathcal{L}_{22}; \Sigma_2)} \quad (36)$$

Figure 4. Auxiliary MixML Judgments and Definitions

passing in an empty realizer) nor dynamic imports (a signature of the form $|\Sigma|$ cannot contain negative variances). Local abstract types $\bar{\beta}$ may not escape their scope by appearing in Σ .

Rule 8 typechecks a module as a self-contained unit. It introduces fresh names for import types ($\bar{\alpha}$) and export types ($\bar{\beta}$), which become quantified in the resulting unit signature. While the rule appears to have to guess the structure of the import type locator \mathcal{L} , as well as the number, order, and kinds of $\bar{\alpha}$ and $\bar{\beta}$, out of nowhere, there is in fact only one way to choose them, which is easy to compute algorithmically. (See the technical appendix for details.)

The main typing judgment for MixML modules has the form $\Gamma; \mathcal{L}; \bar{\beta} \vdash \text{mod} : \Sigma$. Here, $\bar{\beta}$ is a list of distinct abstract types exported by mod , while the realizer \mathcal{L} captures mod 's type imports. As in RMC, the variables $\bar{\beta}$ may also appear free in the bindings of module variables in Γ , which is crucial to avoiding double vision in the presence of recursive modules and sealing [8]. As explained above, we differ from RMC in that we do not attempt to statically detect type cycles. Thanks to this simplification, and the implicit kinding of type variables, type contexts degenerate into simple sets. Because a suitable type context Δ binding all free type variables in a judgment is trivially inferable, we leave it implicit in our rules. We write $\bar{\alpha}$ fresh in premises to mean that $\bar{\alpha} \notin$ the implicit Δ .

Following the approach of RMC, we use shading of certain premises in the typing rules to denote the delta between the main typing judgment and the *static* typing judgment (\vdash_{stat}). The static judgment is used to implement the static pass of recursive linking, as described in the Double Vision subsection of Section 2. To obtain the static version of any rule, simply remove all shaded premises,

and erase all occurrences of term signatures $\llbracket \tau \rrbracket^\pm$ to the empty signature $\{\}$.

Most of the rules for basic modules are fairly straightforward. Notably, Rule 9 for variables X returns the signature $|\Sigma|$, which turns all imports of Σ into exports. This makes sense: regardless of whether the module that X is bound to—call it mod —is fully defined, the module expression X is a *definite* reference to mod and is therefore fully defined. As a result, projections of the form $\text{Tyc}(X.\ell s)$ or $\text{Val}(X.\ell s)$ are always acceptable. The latter may, however, raise a runtime “blackhole” exception if the component $X.\ell s$ refers to is as yet undefined.

Rule 11 for $[:K]$ uses the given realizer to look up the definition of its type import (which is A). Rule 19 for datatype imports works analogously, while Rule 20 conversely produces an export type.

Type-checking structure creation (Rule 15) and projection (Rule 16) is equally straightforward. Note that the latter rule requires the components other than the one being projected out to be fully defined. The reasons for this are discussed in Section 2.

Rule 17 is the central rule of our system: it handles recursive linking, and while it is admittedly somewhat complex, it is essentially just a bidirectional generalization of RMC's typing rule for recursive modules. Let us first step through the rule ignoring the locators. Type checking proceeds by first checking mod_1 , producing a signature Σ_1 for X . As in RMC, checking mod_2 requires two passes. The first, *static* pass, only collects *type* specifications and definitions from mod_2 . The linking rule then uses the bidirectional lookup judgment (Rule 27) to look up mod_1 's type imports

in mod_2 and vice versa. (This step generalizes RMC’s unidirectional lookup.)

The bidirectional lookup judgment will fail if it detects any transparent type cycles. Assuming it succeeds, it yields a type substitution δ , which is then applied to the signature Σ_1 previously computed for mod_1 , intuitively “patching” it with the appropriate type definitions from mod_2 . In this way, when we typecheck mod_2 fully in the subsequent main pass, we see no difference between mod_2 ’s type components and the components with the same name in X . This is the key to avoiding double vision. Lastly, the signatures of mod_1 and mod_2 are merged, yielding the final signature Σ . Merging is defined by an auxiliary judgment described below.

Now about those locators: To deal with type imports, the linking rule has to properly adjust locators and realizers. The joint input realizer is first separated into \mathcal{L}'_1 and \mathcal{L}'_2 , where the former contains imports stemming from mod_1 and the latter those from mod_2 . Note that they may overlap. In addition, each module may have additional *local* imports, *i.e.*, imports that the other module will define. These are handled by locally extending the realizer with fresh locators \mathcal{L}_1 and \mathcal{L}_2 , and later using these for mutual lookup.

Rule 18 for opaque linking is very similar to Rule 17, but slightly simpler because the result of opaque linking is not permitted to have any residual imports. (This is guaranteed by requiring that the merged signature have the form $|\Sigma|$.) In addition, the import types of mod_1 (the $\bar{\alpha}_1$) double as abstract type exports for the whole module, thus implementing sealing. Finally, note that the final signature of the module is derived solely from the signature of mod_1 —all information about mod_2 is kept secret.

Rules 21–23 dealing with atomic unit components and unit instantiation are straightforward. Like type or term projection, the instantiation $\text{new } mod$ in Rule 22 requires that mod be complete, *i.e.*, that it actually contain a unit. The unit it contains, though, is free to have imports, which will become imports of $\text{new } mod$ itself.

The two rules for elaborating unit signatures are simpler than they might look. Rule 24 infers a unit signature for mod and requires that it is free of exports (no export type variables, and $-\Sigma' = |\Sigma'|$). It then partitions the import type variables according to the paths listed in $\bar{\ell}s$, turning unreachable ones into exports. The second form of unit signature (with an `export` clause) is handled in the dual manner.

The merging rules are largely straightforward. One interesting restriction is that the merging of two unit imports (Rule 32) does not allow their signatures to differ. This restriction is in place, for somewhat technical reasons, in order to ensure decidability. It is worth noting, though, that the restriction does not place any limitations on the MixML encoding of ML-style higher-order functors (since that encoding will never attempt to merge two unit imports).

When matching unit exports against unit imports, however (Rule 31), merging allows subtyping in the form of unit signature matching. Rule 36 defines signature matching in terms of linking. The rule checks whether the exports of Φ_1 subsume the exports of Φ_2 and, contravariantly, the imports of Φ_2 subsume those of Φ_1 . It does so by *inverting* the module signature Σ_2 from Φ_2 (*i.e.*, swapping imports and exports) and trying to link it against Σ_1 . Type components are dealt with by using the bidirectional lookup judgment to simultaneously look up the type *exports* of Σ_2 in Σ_1 and the *imports* of Σ_1 in Σ_2 . Notably, this is the only rule that makes use of export locators. In the case of Φ_2 , an export locator is guaranteed to exist because Φ_2 is a target signature—invariants of the type system (formalized in the appendix) ensure that Φ_2 must be the translation of some MixML *usig*.

3.3 Dynamic Semantics and Type Soundness

The dynamic semantics of MixML is given by evidence translation into a simpler internal language (IL) closely based on Dreyer’s

RTG calculus for recursive type generativity [11]. As for RMC, soundness of the translation together with soundness of the IL is then sufficient to establish soundness of MixML.

The structure of the evidence translation follows that of the typing rules for modules in Figure 3, except that each judgment produces an IL term as additional output. The type of this output term corresponds in a precise way to the semantic signature Σ used to classify the input module. It thus serves as *evidence* that the original MixML module is correctly classified by Σ .

A structure is represented as an IL record, and atomic type components directly map to their counterparts in the IL. On the other hand, dynamic components—*i.e.*, atomic values and units—are represented by lazy reference cells, thereby implementing recursive linking via backpatching of uninitialized cells. This follows in the style of Flatt and Felleisen’s *unit* language [15, 29] (discussed below). Consequently, the translation of a unit with signature $\forall \bar{\alpha}. \exists \bar{\beta}. (\mathcal{L}; \Sigma)$ is a function in *destination-passing style* [11] that has type $\forall \bar{\alpha}. \exists \bar{\beta}. \Sigma \rightarrow \{\!\!\}\}$. That is, it takes import types $\bar{\alpha}$, export type names $\bar{\beta}$, and a module of signature Σ with the dynamic content of Σ ’s exports (the positive fields) yet uninitialized. It defines the $\bar{\beta}$ and fills in definitions for all Σ ’s exports.

For more details, we refer the reader to the technical appendix.

4. Related and Future Work

There is a large body of work on ML modules and mixin modules independently, some of which we cited in the introduction. For space reasons, we confine our discussion of related work in this section to modularity mechanisms that attempt a synthesis of ML-style and mixin-style features.

Mixin Modules for ML Duggan and Sourelis were the first to integrate a notion of mixin composition into ML modules with type definitions [14]. However, they divide mixin modules into three parts, and the middle “mixinable” part may only contain `datatype` and `fun` bindings. In addition, their focus lies mainly on merging datatype variants and function clauses in order to support extensible datatypes. They consider neither type abstraction nor hierarchical structures, and expressly disallow separate compilation.

Recursive Modules As mentioned in the introduction, there are several proposals for extending ML with recursive modules [7, 32, 22, 26, 8], but none provides support for recursive linking of separately compiled modules, and most of them run afoul of the double vision problem [8]. Russo [32] suggests how to use functors to support separate compilation for a limited class of recursive modules that do not use sealing and only export term components of function type. This technique does not scale to the general case.

Units Units were originally proposed by Flatt and Felleisen [15] as a recursive module extension to Scheme, which they extended with support for abstract type components. Later work by Owens and Flatt extended units with hierarchical namespaces (called *modules*) and translucent type components [29]. Like MixML, the system presented in the latter paper (hereafter, OF) provides units as a form of mixin module that may contain type components and nested structures, but excludes overriding. Units are first-class in OF, subsuming higher-order units as present in MixML, but also introducing subtyping into the core language. In OF, as in other mixin-based languages, units are not hierarchically composable into other units. Moreover, OF strictly separate imports from exports, making units more functor-like and less flexible than MixML modules.

OF require substantially more bookkeeping annotations from the programmer than MixML. In particular, every unit and linking expression includes explicit specifications of all its imports and exports, and all wiring needed in a linking step must be spelled out

explicitly. While this may offer some added flexibility, it becomes extremely burdensome for encoding ML-style modules. Specifically, OF show how to encode an ML-like module system, but one in which modules require signature annotations on essentially every subterm. (For example, each functor application involves three distinct signature annotations—one for the functor, one for the argument, and one for the result.) In contrast, our MixML encoding of ML modules is simple and direct, and does not assume that the source of the encoding has any additional annotations.

Nevertheless, the work on units has served as a key inspiration for us, and we borrow their terminology of *modules* and *units*, albeit with a somewhat different meaning.

Duggan [13] presents a system similar to OF units, enriched with sealing for units (called modules in his paper) and with an orthogonal construct for dynamic typing. As in other mixin approaches, his modules are not hierarchically composable, and the system does not support transparent type definitions, only datatype definitions and sharing constraints between abstract types. As in MixML, compound structures are built from atomic forms, but using a concatenation operator, separate from mixin linking.

Scala Scala is a language developed by Odersky *et al.* [27, 28] combining object-oriented mixin class composition with ML-style type components. It provides several linking operations; in particular, its mixin composition is very similar to our transparent linking, except that it allows overriding and restricts specialization of abstract fields to be left-to-right. The inheritance mechanism can also be viewed as a form of transparent linking, combined with typical OO-style data abstraction via access modifiers on class fields. This form of data abstraction is less expressive than ML-style sealing, which we model using opaque linking. Scala classes are not hierarchically composable in the manner of MixML modules.

An interesting restriction in Scala is that mixin composition is only allowed with a concrete class, not a class represented by an abstract type. This is due in part to Scala's lack of class signatures. In MixML, the unit import construct `[: usig]` enables one to include an undefined unit with signature *usig* as a component of a module, and to link against it in subsequent code.

Scala introduces *selftypes* as a remedy for this restriction. Basically, selftype annotations can be viewed as another form of linking that specifically allows recursive linking against undefined classes. In doing so, it delays a certain amount of type checking, which is then performed when the class is finally instantiated.

The success of Scala was a major impetus for us to figure out how to incorporate mixin composition into the ML module system.

Future Work We have built a prototype interpreter for a language based on MixML, which includes built-in support for a number of the encodings given in Section 2. We are currently investigating the possibility of incorporating some of the ideas into one of the existing ML implementations.

We believe that our design already provides a fairly complete basis for a practical module system. To further expand its utility, though, we are interested in extending it with a form of first-class unit (*e.g.*, in the style of either Dreyer *et al.*'s *modules-as-first-class-values* [12] or Rossberg's *packages* [31]), as well as OCaml-style applicative functors [19], among other features.

References

- [1] D. Ancona, S. Fagorzi, E. Moggi, and E. Zucca. Mixin modules and computational effects. In *ICALP '03*.
- [2] Davide Ancona and Elena Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998.
- [3] Davide Ancona and Elena Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2):91–132, 2002.
- [4] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA '90*.
- [5] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *ICCL '92*.
- [6] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 Report (revised). Tech. Rep. 52, DEC-SRC, 1989.
- [7] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *PLDI '99*.
- [8] Derek Dreyer. A type system for recursive modules. In *ICFP '07*.
- [9] Derek Dreyer. A type system for well-founded recursion. In *POPL '04*.
- [10] Derek Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 2005.
- [11] Derek Dreyer. Recursive type generativity. *Journal of Functional Programming*, 17(4&5):433–471, 2007.
- [12] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *POPL '03*.
- [13] Dominic Duggan. Type-safe linking with recursive DLLs and shared libraries. *ACM Transactions on Programming Languages and Systems*, 24(6):711–804, 2002.
- [14] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *ICFP '96*.
- [15] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *PLDI '98*.
- [16] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL '94*.
- [17] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.
- [18] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In *ESOP '02*.
- [19] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *POPL '95*.
- [20] Xavier Leroy. Manifest types, modules, and separate compilation. In *POPL '94*.
- [21] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.
- [22] Xavier Leroy. A proposal for recursive modules in Objective Caml, 2003. Available at: http://caml.inria.fr/pub/papers/xleroy-recursive_modules-03.pdf.
- [23] David MacQueen. Modules for Standard ML. In *LFP '84*.
- [24] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [25] D. A. Moon. Object-oriented programming with Flavors. In *OOPSLA '86*.
- [26] Keiko Nakata and Jacques Garrigue. Recursive modules for programming. In *ICFP '06*.
- [27] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *ECOOP '03*.
- [28] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA '05*.
- [29] Scott Owens and Matthew Flatt. From structures and functors to modules and units. In *ICFP '06*.
- [30] Norman Ramsey, Kathleen Fisher, and Paul Govereau. An expressive language of signatures. In *ICFP '05*.
- [31] Andreas Rossberg. The missing link – dynamic components for ML. In *ICFP '06*.
- [32] Claudio V. Russo. Recursive structures for Standard ML. In *ICFP '01*.

Signatures	$\Sigma ::= \llbracket = A \rrbracket \mid \llbracket \tau \rrbracket \mid \{\{\ell : \Sigma\}\} \mid \forall \bar{\alpha}. \Sigma \mid \exists \bar{\alpha}. \Sigma_1 \rightarrow \Sigma_2 \mid \Σ
Terms	$e ::= \mathbf{fold}_{\Delta} \mid \mathbf{unfold}_{\Delta} \mid e_1 \overline{[A]}(e_2) \mid \mathbf{val}(M)$
Modules	$M, F ::= X \mid [A] \mid [e] \mid \{\ell \triangleright X = M\} \mid M.\ell \mid \Lambda \bar{\alpha}. M \mid F[A] \mid \Lambda^{\dagger} \bar{\alpha}. \lambda X : \Sigma. M \mid F[\bar{\beta}](M) \mid \mathbf{new} \bar{\alpha} \mathbf{in} M \mid \mathbf{new}(\Sigma) \mid !M \mid M_1 := M_2 \mid (\mathbf{def} \alpha := A \mathbf{in} M)_{\xi; \Xi} \mid \mathbf{def} \alpha \approx A \mathbf{in} M$
Contexts	$\Delta ::= \emptyset \mid \Delta, \alpha \mid \Delta, \alpha = A \mid \Delta, \alpha \approx A$ $\Gamma ::= \emptyset \mid \Gamma, X : \Sigma$ $\Xi ::= (\Delta; \Gamma; \bar{\beta})$
Substitutions	$\gamma ::= \{\bar{X} \mapsto M\}$ $\xi ::= \delta\gamma$
Type Effects	$\varphi ::= \alpha := A \mid \alpha \approx A$

$\mathbf{let} X = M_1 \mathbf{in} M_2$	$\stackrel{\text{def}}{=} \{1 \triangleright X = M_1, 2 = M_2\}.2$
$\Sigma_1 \rightarrow \Sigma_2$	$\stackrel{\text{def}}{=} \exists \emptyset. \Sigma_1 \rightarrow \Sigma_2$
$\lambda X : \Sigma. M$	$\stackrel{\text{def}}{=} \Lambda^{\dagger} \emptyset. \lambda X : \Sigma. M$
$F(M)$	$\stackrel{\text{def}}{=} F[\emptyset](M)$

$(\mathbf{def} \emptyset \mathbf{in} M)_{\xi; \Xi}$	$\stackrel{\text{def}}{=} \xi M$
$(\mathbf{def} \alpha_1 := A_1, \bar{\alpha} := A \mathbf{in} M)_{\xi; \Xi}$	$\stackrel{\text{def}}{=} (\mathbf{def} \alpha_1 := A_1 \mathbf{in} (\mathbf{def} \bar{\alpha} := A \mathbf{in} M)_{\text{id}; \Xi'})_{\xi; \Xi}$ where $\Xi' = \Xi @ \alpha_1 := A_1$

Figure 5. IL Syntax

A. Internal Language (IL) Type System

The internal language (IL) type system is similar to RMC's. Here, we highlight the major differences in the meta-theory.

Unlike in RMC, we do not attempt to detect all type cycles statically. As a result, certain aspects of the type system become simpler. First, we only have one abstract type binding α , which is like $\alpha : K$ in RMC's IL (and sometimes we write it that way if we want to make the kind K of α explicit). Second, we are able to use a linear type system to track type variable definitions, instead of the effect system used by RMC. Specifically, when an RMC typing judgment writes “with $\bar{\beta} \downarrow$ ” at the end, we write $\bar{\beta}$ as a third context left of the turnstile. The $\bar{\beta}$ contexts are treated linearly, and this ensures that variables get defined exactly once.

The module $\mathbf{new}(\Sigma)$ creates an uninitialized memory cell X (of signature Σ) that will eventually contain a module of signature Σ . The module $M_1 := M_2$ slides M_2 (unevaluated) into the memory cell referenced by M_1 . The module $!M$ forces the computation of the memory cell referenced by M , and backpatches it with the result of the computation. The other modules are similar to ones in RMC.

Lastly, the first \mathbf{def} construct is the same as in RMC, except that it carries a substitution ξ along with it, which is essentially a closure. (To apply a substitution ξ' to this construct, we just replace ξ with $\xi'\xi$, but do not descend inside M .) The reason for this is that, due to the dynamic detection of type cycles, it is possible that a substitution into the \mathbf{def} might render the definition $\alpha := A$ ill-formed (cyclic). Using a closure enables us to prove a substitution property for the language. Consequently, at run time, when we evaluate this construct, we apply the closing substitution ξ to it, check dynamically that $\xi\alpha := \xi A$ is still well-formed (acyclic), apply ξ to M and proceed. If there is a cycle, we raise a run-time BlackHole exception.

Definition A.1 (Well-Formed Substitutions)

We say that a substitution ξ maps Ξ to Ξ' , written $\Xi' \vdash \xi : \Xi$, if:

1. $\xi = \delta\gamma$, $\Xi = (\Delta; \Gamma; \bar{\beta})$ and $\Xi' = (\Delta'; \Gamma'; \bar{\beta}')$
2. $\vdash \Xi$ and $\vdash \Xi'$
3. $\Delta' \vdash \delta : \Delta$
4. $\delta\bar{\beta} = \bar{\beta}'$
5. $\text{dom}(\gamma) = \text{dom}(\Gamma)$
6. $\forall X : \Sigma \in \Gamma. \Delta'; \Gamma' \vdash \gamma X : \delta\Sigma$

Proposition A.2 (Substitution)

Suppose $\Xi' \vdash \xi : \Xi$, where $\Xi' = (\Delta'; \Gamma'; \bar{\beta}')$ and $\Xi = (\Delta; \Gamma; \bar{\beta})$.

1. If $\Delta; \Gamma \vdash e : \tau$, then $\Delta'; \Gamma' \vdash \xi e : \xi\tau$.
2. If $\Xi \vdash M : \Sigma$, then $\Xi' \vdash \xi M : \xi\Sigma$.

Definition A.3 (Run-Time Module Contexts)

We say that a module context Γ is *run-time* if it only contains bindings of the form $X : \$\Sigma$.

Definition A.4 (Well-Formed Machine Stores)

We say that a machine store ω is well-formed in Δ and has type Γ , written $\Delta \vdash \omega : \Gamma$, if:

1. $\Delta \vdash \Gamma$ and Γ is run-time
2. $\text{dom}(\omega) = \text{dom}(\Gamma)$
3. $\forall X : \$\Sigma \in \Gamma. \text{either } \omega(X) = ? \text{ or } \Delta; \Gamma \vdash \omega(X) : \Sigma$

Definition A.5 (Well-Formed Machine States)

We say that a machine state Ω is well-formed, written $\vdash \Omega$, if either:

- $\Omega = \text{BlackHole}$
- or $\Omega = (\Delta; \omega; \mathbb{C}; e)$ and there exist Γ, τ , and $\bar{\beta} \subseteq \Delta$ s.t.:
 1. $\Delta \vdash \omega : \Gamma$
 2. $\Delta; \Gamma \vdash e : \tau$
 3. $\Delta; \Gamma; \bar{\beta} \vdash \mathbb{C} : \tau \text{ cont}$
- or $\Omega = (\Delta; \omega; \mathbb{C}; M)$ and there exist $\Gamma, \Sigma, \bar{\beta}_1 \uplus \bar{\beta}_2 \subseteq \Delta$ s.t.:
 1. $\Delta \vdash \omega : \Gamma$
 2. $\Delta; \Gamma; \bar{\beta}_1 \vdash M : \Sigma$
 3. $\Delta; \Gamma; \bar{\beta}_2 \vdash \mathbb{C} : \Sigma \text{ cont}$

Theorem A.6 (Preservation)

If $\vdash \Omega$ and $\Omega \mapsto \Omega'$, then $\vdash \Omega'$.

Lemma A.7 (Canonical Forms)

Suppose $\Delta; \Gamma \Vdash v : \tau$ and $\Delta; \Gamma \Vdash V : \Sigma$ and Γ is run-time.

1. If $\tau = \forall[\bar{\alpha}]. \tau_1 \Rightarrow \tau_2$, then v is of the form \mathbf{fold}_{β} or \mathbf{unfold}_{β} .
2. If $\tau = \alpha$ or $\tau = \alpha(\bar{A})$, where $\alpha \approx A' \in \Delta$, then v is of the form $\mathbf{fold}_{\beta}[\bar{B}](v')$.
3. If $\Sigma = \llbracket = A \rrbracket$, then V is of the form $[B]$.
4. If $\Sigma = \llbracket \tau' \rrbracket$, then V is of the form $[v']$.
5. If $\Sigma = \{\{\ell : \Sigma'\}\}$, then V is of the form $\{\{\ell = V'\}\}$.
6. If $\Sigma = \forall \bar{\alpha}. \Sigma'$, then V is of the form $\Lambda \bar{\alpha}. M$.
7. If $\Sigma = \exists \bar{\alpha}. \Sigma_1 \rightarrow \Sigma_2$, then V is of the form $\Lambda^{\dagger} \bar{\alpha}. \lambda X : \Sigma'_1. M$.
8. If $\Sigma = \$\Sigma'$, then V is of the form X .

Theorem A.8 (Progress)

If $\vdash \Omega$, then Ω is not stuck.

Corollary A.9 (Type Soundness)

If $\emptyset; \emptyset \vdash M : \Sigma$, then for all $\Omega, (\emptyset; \emptyset; \bullet; M) \mapsto^* \Omega$ implies that Ω is not stuck.

Well-formed type effects: $\Delta \vdash \varphi$

$$\frac{\alpha : K \in \Delta \quad \Delta \vdash A : K \quad \alpha \notin \text{FV}(\text{norm}_\Delta(A))}{\Delta \vdash \alpha := A}$$

$$\frac{\alpha : K \in \Delta \quad \Delta \vdash A : K}{\Delta \vdash \alpha \approx A}$$

Application of a type effect: $\Delta @ \varphi$

$$\Delta @ \alpha := A \stackrel{\text{def}}{=} \Delta \setminus \{\alpha\} \cup \{\alpha = \text{norm}_\Delta(A)\}$$

$$\Delta @ \alpha \approx A \stackrel{\text{def}}{=} \Delta \setminus \{\alpha\} \cup \{\alpha \approx A\}$$

Well-formed terms: $\Delta; \Gamma \vdash e : \tau$

$$\frac{\Delta; \Gamma \vdash M : \llbracket \tau \rrbracket}{\Delta; \Gamma \vdash \text{val}(M) : \tau} \quad \frac{\alpha : 0 \approx \tau \in \Delta}{\Delta; \Gamma \vdash \text{fold}_\alpha : \forall \square. \tau \Rightarrow \alpha} \quad \frac{\alpha : 0 \approx \tau \in \Delta}{\Delta; \Gamma \vdash \text{unfold}_\alpha : \forall \square. \alpha \Rightarrow \tau}$$
$$\frac{\alpha : n \approx A \in \Delta \quad \bar{\beta} (= \beta_1, \dots, \beta_n) \cap \text{dom}(\Delta) = \emptyset}{\Delta; \Gamma \vdash \text{fold}_\alpha : \forall \bar{\beta}. A(\bar{\beta}) \Rightarrow \alpha(\bar{\beta})} \quad \frac{\alpha : n \approx A \in \Delta \quad \bar{\beta} (= \beta_1, \dots, \beta_n) \cap \text{dom}(\Delta) = \emptyset}{\Delta; \Gamma \vdash \text{unfold}_\alpha : \forall \bar{\beta}. \alpha(\bar{\beta}) \Rightarrow A(\bar{\beta})}$$
$$\frac{\Delta; \Gamma \vdash e_1 : \forall \bar{\alpha}. \tau_2 \Rightarrow \tau \quad \Delta \vdash A : 0 \quad \Delta; \Gamma \vdash e_2 : \{\overline{\alpha \mapsto A}\} \tau_2}{\Delta; \Gamma \vdash e_1[\bar{A}](e_2) : \{\overline{\alpha \mapsto A}\} \tau} \quad \frac{\Delta; \Gamma \vdash e : \tau' \quad \Delta \vdash \tau' \equiv \tau : 0}{\Delta; \Gamma \vdash e : \tau}$$

Well-formed modules: $\Delta; \Gamma; \bar{\beta} \vdash M : \Sigma$

We write $\Delta; \Gamma \vdash M : \Sigma$ as shorthand for $\Delta; \Gamma; \emptyset \vdash M : \Sigma$.

$$\frac{X : \Sigma \in \Gamma}{\Delta; \Gamma \vdash X : \Sigma} \quad \frac{\Delta \vdash A : K}{\Delta; \Gamma \vdash [A] : \llbracket A \rrbracket} \quad \frac{\Delta; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash [e] : \llbracket \tau \rrbracket} \quad \frac{}{\Delta; \Gamma \vdash \{\} : \{\}}$$
$$\frac{\Delta; \Gamma; \bar{\beta}_1 \vdash M_1 : \Sigma_1 \quad \Delta; \Gamma; X_1 : \Sigma_1; \bar{\beta}_2 \vdash \{\ell \triangleright X = M\} : \{\ell : \Sigma\}}{\Delta; \Gamma; \bar{\beta}_1, \bar{\beta}_2 \vdash \{\ell_1 \triangleright X_1 = M_1, \ell \triangleright X = M\} : \{\ell_1 : \Sigma_1, \ell : \Sigma\}} \quad \frac{\Delta; \Gamma; \bar{\beta} \vdash M : \{\dots, \ell : \Sigma, \dots\}}{\Delta; \Gamma; \bar{\beta} \vdash M.l : \Sigma}$$
$$\frac{\Delta, \bar{\alpha}; \Gamma \vdash M : \Sigma}{\Delta; \Gamma \vdash \Lambda \bar{\alpha}. M : \forall \bar{\alpha}. \Sigma} \quad \frac{\Delta; \Gamma \vdash F : \forall \bar{\alpha}. \Sigma \quad \Delta \vdash A : \bar{K}}{\Delta; \Gamma \vdash F[\bar{A}] : \{\overline{\alpha \mapsto A}\} \Sigma}$$
$$\frac{\Delta \vdash \Sigma_1 \quad \Delta, \bar{\alpha}; \Gamma; X : \Sigma_1; \bar{\alpha} \vdash M : \Sigma_2}{\Delta; \Gamma \vdash \Lambda \bar{\alpha}. \lambda X : \Sigma_1. M : \exists \bar{\alpha}. \Sigma_1 \rightarrow \Sigma_2} \quad \frac{\Delta; \Gamma \vdash F : \exists \bar{\alpha} : \bar{K}. \Sigma_1 \rightarrow \Sigma_2 \quad \bar{\beta} : \bar{K} \subseteq \Delta \quad \Delta; \Gamma \vdash M : \{\overline{\alpha \mapsto \bar{\beta}}\} \Sigma_1}{\Delta; \Gamma; \bar{\beta} \vdash F[\bar{\beta}](M) : \{\overline{\alpha \mapsto \bar{\beta}}\} \Sigma_2}$$
$$\frac{\Delta, \bar{\alpha}; \Gamma; \bar{\beta} \vdash M : \Sigma \quad \bar{\alpha} \notin \text{FV}(\Sigma)}{\Delta; \Gamma; \bar{\beta} \setminus \bar{\alpha} \vdash \text{new } \bar{\alpha} \text{ in } M : \Sigma} \quad \frac{\Delta \vdash \Sigma}{\Delta; \Gamma \vdash \text{new}(\Sigma) : \$\Sigma} \quad \frac{\Delta; \Gamma \vdash M : \$\Sigma}{\Delta; \Gamma \vdash !M : \Sigma} \quad \frac{\Delta; \Gamma \vdash M_1 : \$\Sigma \quad \Delta; \Gamma \vdash M_2 : \Sigma}{\Delta; \Gamma \vdash M_1 := M_2 : \{\}}$$
$$\frac{\Xi' \vdash \xi : \Xi = (\Delta; \Gamma; \alpha, \bar{\beta}) \quad \Delta \vdash \alpha := A \quad \Delta @ \alpha := A; \Gamma; \bar{\beta} \vdash M : \{\}}{\Xi' \vdash (\text{def } \alpha := A \text{ in } M)_{\xi; \Xi} : \{\}}$$
$$\frac{\Delta \vdash \alpha \approx A \quad \Delta @ \alpha \approx A; \Gamma; \bar{\beta} \vdash M : \{\}}{\Delta; \Gamma; \alpha, \bar{\beta} \vdash \text{def } \alpha \approx A \text{ in } M : \{\}} \quad \frac{\Delta; \Gamma; \bar{\beta} \vdash M : \Sigma' \quad \Delta \vdash \Sigma' \equiv \Sigma}{\Delta; \Gamma; \bar{\beta} \vdash M : \Sigma}$$

Figure 6. IL Static Semantics

Core Values	$v ::= \text{fold}_\alpha \mid \text{unfold}_\alpha \mid \text{fold}_\alpha[\bar{A}](v)$
Module Values	$V ::= X \mid [A] \mid [v] \mid \{\ell = \bar{V}\} \mid \Lambda \bar{\alpha}. M \mid \Lambda^\dagger \bar{\alpha}. \lambda X. \Sigma. M$
Machine States	$\Omega ::= (\Delta; \omega; \mathbb{C}; e) \mid (\Delta; \omega; \mathbb{C}; M) \mid \text{BlackHole}$
Machine Stores	$\omega ::= \emptyset \mid \omega, X \mapsto M \mid \omega, X \mapsto ?$
Continuations	$\mathbb{C} ::= \bullet \mid \mathbb{C} \circ \mathbb{F}$
Continuation Frames	$\mathbb{F} ::= \bullet[\bar{A}](e) \mid v[\bar{A}](\bullet) \mid \text{Val}(\bullet) \mid$ $[\bullet] \mid \{\ell_1 = \bar{V}_1, \ell_2 \triangleright X = \bullet, \ell_2 \triangleright X_2 = \bar{M}_2\} \mid \bullet.l \mid$ $\bullet[\bar{A}] \mid \bullet[\bar{\alpha}](M) \mid V[\bar{\alpha}](\bullet) \mid !\bullet \mid \bullet := M \mid X := \bullet$

Machine state transitions: $\Omega \mapsto \Omega'$

$$\begin{array}{c}
\frac{e = e_1[\bar{A}](e_2) \quad e \text{ not a value}}{(\Delta; \omega; \mathbb{C}; e) \mapsto (\Delta; \omega; \mathbb{C} \circ \bullet[\bar{A}](e_2); e_1)} \quad \frac{}{(\Delta; \omega; \mathbb{C} \circ \bullet[\bar{A}](e); v) \mapsto (\Delta; \omega; \mathbb{C} \circ v[\bar{A}](\bullet); e)} \\
\frac{}{(\Delta; \omega; \mathbb{C} \circ \text{fold}_\alpha[\bar{A}](\bullet); v) \mapsto (\Delta; \omega; \mathbb{C}; \text{fold}_\alpha[\bar{A}](v))} \quad \frac{}{(\Delta; \omega; \mathbb{C} \circ \text{unfold}_\alpha[\bar{A}](\bullet); \text{fold}_\beta[\bar{B}](v)) \mapsto (\Delta; \omega; \mathbb{C}; v)} \\
\frac{}{(\Delta; \omega; \mathbb{C}; \text{Val}(M)) \mapsto (\Delta; \omega; \mathbb{C} \circ \text{Val}(\bullet); M)} \quad \frac{}{(\Delta; \omega; \mathbb{C} \circ \text{Val}(\bullet); [v]) \mapsto (\Delta; \omega; \mathbb{C}; v)} \\
\frac{e \text{ not a value}}{(\Delta; \omega; \mathbb{C}; [e]) \mapsto (\Delta; \omega; \mathbb{C} \circ [\bullet]; e)} \quad \frac{}{(\Delta; \omega; \mathbb{C} \circ [\bullet]; v) \mapsto (\Delta; \omega; \mathbb{C}; [v])} \\
\frac{M = \{\ell_1 \triangleright X_1 = M_1, \ell_2 \triangleright X_2 = \bar{M}_2\} \quad M \text{ not a value}}{(\Delta; \omega; \mathbb{C}; M) \mapsto (\Delta; \omega; \mathbb{C} \circ \{\ell_1 \triangleright X_1 = \bullet, \ell_2 \triangleright X_2 = \bar{M}_2\}; M_1)} \\
\frac{}{(\Delta; \omega; \mathbb{C} \circ \{\ell_1 = \bar{V}_1, \ell_2 \triangleright X_2 = \bullet, \ell_3 \triangleright X_3 = M_3, \ell_4 \triangleright X_4 = \bar{M}_4\}; V_2) \mapsto} \\
\frac{}{(\Delta; \omega; \mathbb{C} \circ \{\ell_1 = \bar{V}_1, \ell_2 = V_2, \ell_3 \triangleright X_3 = \bullet, \ell_4 \triangleright X_4 = \{X_2 \mapsto V_2\}M_4\}; \{X_2 \mapsto V_2\}M_3)} \\
\frac{}{(\Delta; \omega; \mathbb{C} \circ \{\ell_1 = \bar{V}_1, \ell_2 = \bullet\}; V_2) \mapsto (\Delta; \omega; \mathbb{C}; \{\ell_1 = \bar{V}_1, \ell_2 = V_2\})} \\
\frac{V = \{\dots, \ell = V_\ell, \dots\}}{(\Delta; \omega; \mathbb{C}; M.l) \mapsto (\Delta; \omega; \mathbb{C} \circ \bullet.l; M)} \quad \frac{}{(\Delta; \omega; \mathbb{C} \circ \bullet.l; V) \mapsto (\Delta; \omega; \mathbb{C}; V_\ell)} \\
\frac{}{(\Delta; \omega; \mathbb{C}; \mathbb{F}[\bar{A}]) \mapsto (\Delta; \omega; \mathbb{C} \circ \bullet[\bar{A}]; \mathbb{F})} \quad \frac{}{(\Delta; \omega; \mathbb{C} \circ \bullet[\bar{A}]; \Lambda \bar{\alpha}. M) \mapsto (\Delta; \omega; \mathbb{C}; \{\bar{\alpha} \mapsto \bar{A}\}M)} \\
\frac{}{(\Delta; \omega; \mathbb{C}; \mathbb{F}[\bar{\beta}](M)) \mapsto (\Delta; \omega; \mathbb{C} \circ \bullet[\bar{\beta}](M); \mathbb{F})} \quad \frac{}{(\Delta; \omega; \mathbb{C} \circ \bullet[\bar{\beta}](M); V) \mapsto (\Delta; \omega; \mathbb{C} \circ V[\bar{\beta}](\bullet); M)} \\
\frac{}{(\Delta; \omega; \mathbb{C} \circ (\Lambda^\dagger \bar{\alpha}. \lambda X. \Sigma. M)[\bar{\beta}](\bullet); V) \mapsto (\Delta; \omega; \mathbb{C}; \{\bar{\alpha} \mapsto \bar{\beta}\}\{X \mapsto V\}M)} \\
\frac{\bar{\alpha} \notin \text{dom}(\Delta)}{(\Delta; \omega; \mathbb{C}; \text{new } \bar{\alpha} \text{ in } M) \mapsto (\Delta, \bar{\alpha}; \omega; \mathbb{C}; M)} \quad \frac{X \notin \text{dom}(\omega)}{(\Delta; \omega; \mathbb{C}; \text{new}(\Sigma)) \mapsto (\Delta; \omega, X \mapsto ?; \mathbb{C}; X)} \\
\frac{}{(\Delta; \omega; \mathbb{C}; !M) \mapsto (\Delta; \omega; \mathbb{C} \circ !\bullet; M)} \quad \frac{X \mapsto ? \in \omega}{(\Delta; \omega; \mathbb{C} \circ !\bullet; X) \mapsto \text{BlackHole}} \quad \frac{X \mapsto M \in \omega}{(\Delta; \omega; \mathbb{C} \circ !\bullet; X) \mapsto (\Delta; \omega; \mathbb{C} \circ X := \bullet; M)} \\
\frac{X \in \text{dom}(\omega)}{(\Delta; \omega; \mathbb{C} \circ X := \bullet; V) \mapsto (\Delta; \omega @ X := V; \mathbb{C}; V)} \quad \frac{}{(\Delta; \omega; \mathbb{C}; M_1 := M_2) \mapsto (\Delta; \omega; \mathbb{C} \circ \bullet := M_2; M_1)} \\
\frac{X \in \text{dom}(\omega)}{(\Delta; \omega; \mathbb{C} \circ \bullet := M; X) \mapsto (\Delta; \omega @ X := M; \mathbb{C}; \{\})} \quad \frac{\alpha \in \Delta}{(\Delta; \omega; \mathbb{C}; \text{def } \alpha := A \text{ in } M) \mapsto (\Delta @ \alpha := A; \omega; \mathbb{C}; M)} \\
\frac{\Delta \vdash \xi \alpha := \xi A}{(\Delta; \omega; \mathbb{C}; (\text{def } \alpha := A \text{ in } M)_{\xi; \Xi}) \mapsto (\Delta @ \xi \alpha := \xi A; \omega; \mathbb{C}; \xi M)} \quad \frac{\xi \alpha \in \text{FV}(\text{norm}_\Delta(\xi A))}{(\Delta; \omega; \mathbb{C}; (\text{def } \alpha := A \text{ in } M)_{\xi; \Xi}) \mapsto \text{BlackHole}}
\end{array}$$

Figure 7. IL Dynamic Semantics

Well-formed machine states: $\vdash \Omega$

$$\frac{}{\vdash \text{BlackHole}} \quad \frac{\Delta \vdash \omega : \Gamma \quad \Delta; \Gamma \vdash e : \tau \quad \Delta; \Gamma; \bar{\beta} \vdash \mathbb{C} : \tau \text{ cont}}{\vdash (\Delta; \omega; \mathbb{C}; e)}$$

$$\frac{\Delta \vdash \omega : \Gamma \quad \Delta; \Gamma; \bar{\beta}_1 \vdash M : \Sigma \quad \Delta; \Gamma; \bar{\beta}_2 \vdash \mathbb{C} : \Sigma \text{ cont} \quad \bar{\beta}_1 \cap \bar{\beta}_2 = \emptyset}{\vdash (\Delta; \omega; \mathbb{C}; M)}$$

Well-formed machine stores: $\Delta \vdash \omega : \Gamma$

$$\frac{\Delta \vdash \Gamma \quad \text{dom}(\omega) = \text{dom}(\Gamma) \quad \forall X : \Sigma \in \Gamma. \text{ either } \omega(X) = ? \text{ or } \Delta; \Gamma \vdash \omega(X) : \Sigma}{\Delta \vdash \omega : \Gamma}$$

Well-formed continuations: $\Delta; \Gamma; \bar{\beta} \vdash \mathbb{C} : \tau/\Sigma \text{ cont}$

$$\frac{\Delta \vdash \tau : 0/\Delta \vdash \Sigma \quad \Delta; \Gamma; \bar{\beta}_1 \vdash \mathbb{F} : \tau_1/\Sigma_1 \rightsquigarrow \tau_2/\Sigma_2 \quad \Delta; \Gamma; \bar{\beta}_2 \vdash \mathbb{C} : \tau_2/\Sigma_2 \text{ cont}}{\Delta; \Gamma; \emptyset \vdash \bullet : \tau/\Sigma \text{ cont} \quad \Delta; \Gamma; \bar{\beta}_1, \bar{\beta}_2 \vdash \mathbb{C} \circ \mathbb{F} : \tau_1/\Sigma_1 \text{ cont}}$$

$$\frac{\Delta; \Gamma; \bar{\beta} \vdash \mathbb{C} : \tau'/\Sigma' \text{ cont} \quad \Delta \vdash \tau' \equiv \tau : 0/\Delta \vdash \Sigma' \equiv \Sigma}{\Delta; \Gamma; \bar{\beta} \vdash \mathbb{C} : \tau/\Sigma \text{ cont}}$$

Well-formed continuation frames: $\Delta; \Gamma; \bar{\beta} \vdash \mathbb{F} : \tau_1/\Sigma_1 \rightsquigarrow \tau_2/\Sigma_2$

Notation: We may omit $\bar{\beta}$ if $\bar{\beta} = \emptyset$.

$$\frac{\overline{\Delta \vdash A : 0} \quad \Delta; \Gamma \vdash e : \{\overline{\alpha \mapsto A}\} \tau_1 \quad \Delta, \overline{\alpha : 0} \vdash \tau_2 : 0}{\Delta; \Gamma \vdash \bullet[\overline{A}](e) : \{\forall \overline{\alpha}. \tau_1 \Rightarrow \tau_2\} \rightsquigarrow \{\overline{\alpha \mapsto A}\} \tau_2} \quad \frac{\overline{\Delta \vdash A : 0} \quad \Delta; \Gamma \vdash v : \forall \overline{\alpha}. \tau_1 \Rightarrow \tau_2}{\Delta; \Gamma \vdash v[\overline{A}](\bullet) : \{\overline{\alpha \mapsto A}\} \tau_1 \rightsquigarrow \{\overline{\alpha \mapsto A}\} \tau_2}$$

$$\frac{\Delta \vdash \tau : 0}{\Delta; \Gamma \vdash \text{val}(\bullet) : \llbracket \tau \rrbracket \rightsquigarrow \tau} \quad \frac{\Delta \vdash \tau : 0}{\Delta; \Gamma \vdash \bullet : \tau \rightsquigarrow \llbracket \tau \rrbracket} \quad \frac{\Delta \vdash \Sigma \quad \Sigma = \{\dots, \ell : \Sigma_\ell, \dots\}}{\Delta; \Gamma \vdash \bullet.\ell : \Sigma \rightsquigarrow \Sigma_\ell}$$

$$\frac{\overline{\Delta; \Gamma \vdash V_1 : \Sigma_1} \quad \Delta \vdash \Sigma \quad \Delta; \Gamma, X : \Sigma; \bar{\beta} \vdash \{\overline{\ell_2 \triangleright X_2 = M_2}\} : \{\overline{\ell_2 : \Sigma_2}\}}{\Delta; \Gamma; \bar{\beta} \vdash \{\overline{\ell_1 = V_1, \ell \triangleright X = \bullet, \overline{\ell_2 \triangleright X_2 = M_2}\} : \Sigma \rightsquigarrow \{\overline{\ell_1 : \Sigma_1, \ell : \Sigma, \ell_2 : \Sigma_2}\}} \quad \frac{\Delta, \overline{\alpha} \vdash \Sigma \quad \overline{\Delta \vdash A : K}}{\Delta; \Gamma \vdash \bullet[\overline{A}] : \forall \overline{\alpha}. \Sigma \rightsquigarrow \{\overline{\alpha \mapsto A}\} \Sigma}$$

$$\frac{\Delta; \Gamma \vdash M : \{\overline{\alpha \mapsto \beta}\} \Sigma_1 \quad \Delta \vdash \exists \overline{\alpha}. \Sigma_1 \rightarrow \Sigma_2}{\Delta; \Gamma; \bar{\beta} \vdash \bullet[\overline{\beta}](M) : (\exists \overline{\alpha}. \Sigma_1 \rightarrow \Sigma_2) \rightsquigarrow \{\overline{\alpha \mapsto \beta}\} \Sigma_2} \quad \frac{\Delta; \Gamma \vdash V : \exists \overline{\alpha}. \Sigma_1 \rightarrow \Sigma_2}{\Delta; \Gamma; \bar{\beta} \vdash V[\overline{\beta}](\bullet) : \{\overline{\alpha \mapsto \beta}\} \Sigma_1 \rightsquigarrow \{\overline{\alpha \mapsto \beta}\} \Sigma_2}$$

$$\frac{\Delta \vdash \Sigma}{\Delta; \Gamma \vdash !\bullet : \$\Sigma \rightsquigarrow \Sigma} \quad \frac{\Delta; \Gamma \vdash M : \Sigma}{\Delta; \Gamma \vdash \bullet := M : \$\Sigma \rightsquigarrow \{\}} \quad \frac{\Delta; \Gamma \vdash X : \$\Sigma}{\Delta; \Gamma \vdash X := \bullet : \Sigma \rightsquigarrow \Sigma}$$

B. Evidence Translation and Soundness

We define the dynamic semantics of MixML by translation to the IL. The translation is given by the rules in Figures 8 and 9. The structure of the translation rules is identical to that of the typing rules from Section 3, except that each judgement produces an IL term as additional output.

Figure 9 defines an erasure $(\cdot)^\circ$ from MixML semantic objects (Figure 2) to IL signatures, by removing all locators and variance annotations. Erasure extends pointwise to module contexts Γ . The translation produces IL terms that correspond to the erased semantic signature derived by the typing rules, i.e., the derived term serves as *evidence* for the derived signature.

A structure is represented as an IL record. Atomic type components directly map to their counterparts in the IL. Dynamic modules – i.e., values and higher-order units – are represented as lazy cells, thereby enabling a translation of recursive linking into an expres-

sion that backpatches uninitialized import cells with the content of initialized output cells. We call this term the module *initializer*.

Consequently, a unit of signature $\forall\bar{\alpha}. \exists\bar{\beta}. (\mathcal{L}; \Sigma)$ is represented by a polymorphic initialisation function $\forall\bar{\alpha}. \exists\bar{\beta}. \Sigma^\circ \rightarrow \{\!\!\}\}$ in destination passing style [11]. As explained in Section 3.3, the function takes import types $\bar{\alpha}$, export type names $\bar{\beta}$, and the representation of the complete module with all dynamic content being yet uninitialized. It defines the export types and fills in all dynamic content. The main judgement for translating modules is thus defined relative to a path expression P, which provides a handle to the representation of the (sub)module being defined (P can be viewed as a reference to *self*).

Given these ideas, most of the evidence translation is rather straightforward, although being in destination passing style, it notably is “backward” with respect to the module it initialises. That is, wherever the typing rules produce a smaller module from larger operands (e.g. for projection or opaque linking), the translation must create larger modules for use in the respective operand initialisers.

The main points of interest are thus the following. Rule 39 closes an initialiser over all its arguments, producing a stand-alone unit function. Conversely, Rule 54 applies this function to initialise a unit. Rule 38 performs the actual creation of a fresh module and its export type names, and evaluates the initialiser expression M.

New modules also have to be created in Rule 47 for projection and Rule 49 for opaque linking, which are the two constructs that mask parts of a larger module – the evidence expression thus must inversely create these larger modules. Specifically, the latter rule must locally create the combined module taking up both mod_1 and mod_2 , and then copy out the restricted export to the destination. Moreover, it defines the abstract types $\bar{\alpha}_1$ that are introduced by the linkage.

Both linking rules rely on the merging rules producing evidences M_1 and M_2 for projecting each of the operand modules back out of the linked result – again in accordance with the backward nature of destination passing. These submodules are then used to initialise the operands of linking.

The only interesting bits in the evidence translation of merging is the treatment of dynamic components. Because we allow linking to create subtypes, it is necessary to insert a coercion function to go back from the subtype to the supertype in the less specific operand. This is done by creating an auxiliary cell that (lazily) applies the coercion function obtained as evidence of the respective subtyping judgement. In the case of units, the coercion function is created as evidence of the signature matching rule 63.

The evidence of signature matching is a higher-order function taking a unit function F of the smaller type and delivering one of the larger. To do so, the resulting function creates evidence for an auxiliary module X of signature $|\Sigma|$, which produces the connection between the smaller internal and the larger external module signature. It also creates a fresh set of export types $\bar{\beta}_1$ for the original unit F and defines its own exports $\bar{\beta}_2$ using the type substitution δ derived by the typing rule. In a similar manner, substituted import types $\bar{\alpha}_1$ are passed to F. The most subtle feature about this part of the translation is the use of the Copy meta-operator to wire the exports from the projected module X'_2 , which represents $-\delta\Sigma_2$, back to the destination X_2 of the constructed unit function – and vice versa the imports. Since this wiring is bidirectional – i.e., depends on the variance of the individual components – the definition of Copy is such that the assignment is done in the appropriate direction for each component, depending on its variance.

The following properties show that the evidence translation is sound and complete. They are proved by mostly straightforward simultaneous induction on the derivation.

Core-language Expressions: $\Gamma \vdash \text{exp} : \tau \rightsquigarrow e$

$$\frac{\Gamma \vdash \text{mod} : \llbracket \tau \rrbracket^+ \rightsquigarrow M}{\Gamma \vdash \text{Val}(\text{mod}) : \tau \rightsquigarrow \text{Val}(!M)} \quad (37)$$

Complete Modules and Units: $\Gamma \vdash \text{mod} : \Sigma \rightsquigarrow M \quad \Gamma \vdash \text{mod} : \Phi \rightsquigarrow F$

$$\frac{\Gamma; \{\!\!\}; \bar{\beta} \vdash \text{mod} : |\Sigma| \rightsquigarrow X \triangleright M \quad \bar{\beta} \text{ fresh} \quad \bar{\beta} \notin \text{FV}(\Sigma)}{\Gamma \vdash \text{mod} : |\Sigma| \rightsquigarrow \text{new } \bar{\beta} \text{ in let } X = \text{Create}(|\Sigma|) \text{ in } M; X} \quad (38)$$

$$\frac{\Gamma; \mathcal{L}; \bar{\beta} \vdash \text{mod} : \Sigma \rightsquigarrow X \triangleright M \quad \vdash \mathcal{L} \text{ locates } \bar{\alpha} \quad \bar{\alpha}, \bar{\beta} \text{ fresh}}{\Gamma \vdash \text{mod} : \forall \bar{\alpha}. \exists \bar{\beta}. (\mathcal{L}; \Sigma) \rightsquigarrow \Lambda \bar{\alpha}. \Lambda^! \bar{\beta}. \lambda X : \Sigma^\circ. M} \quad (39)$$

Modules: $\Gamma; \mathcal{L}; \bar{\beta} \vdash \text{mod} : \Sigma \rightsquigarrow P \triangleright M$

$$\frac{X : \Sigma \in \Gamma}{\Gamma; \{\!\!\}; \emptyset \vdash X : |\Sigma| \rightsquigarrow P \triangleright \text{Copy}(X, P : |\Sigma|)} \quad (40) \quad \frac{}{\Gamma; \{\!\!\}; \emptyset \vdash \{\} : \{\!\!\} \rightsquigarrow P \triangleright \{\}} \quad (41)$$

$$\frac{\vdash A : K}{\Gamma; \llbracket = A \rrbracket; \emptyset \vdash \llbracket : K \rrbracket : \llbracket = A \rrbracket \rightsquigarrow P \triangleright \{\}} \quad (42) \quad \frac{\Gamma \vdash \text{tyc} \rightsquigarrow A}{\Gamma; \{\!\!\}; \emptyset \vdash \llbracket \text{tyc} \rrbracket : \llbracket = A \rrbracket \rightsquigarrow P \triangleright \{\}} \quad (43)$$

$$\frac{\Gamma \vdash \text{tyc} \rightsquigarrow \tau : 0}{\Gamma; \{\!\!\}; \emptyset \vdash \llbracket : \text{tyc} \rrbracket : \llbracket \tau \rrbracket^- \rightsquigarrow P \triangleright \{\}} \quad (44) \quad \frac{\Gamma \vdash \text{exp} : \tau \rightsquigarrow e}{\Gamma; \{\!\!\}; \emptyset \vdash \llbracket \text{exp} \rrbracket : \llbracket \tau \rrbracket^+ \rightsquigarrow P \triangleright \text{let } X = [e] \text{ in } P := X} \quad (45)$$

$$\frac{\Gamma; \mathcal{L}; \bar{\beta} \vdash \text{mod} : \Sigma \rightsquigarrow P.l \triangleright M}{\Gamma; \{\!\!\}; \mathcal{L}; \bar{\beta} \vdash \{\ell = \text{mod}\} : \{\!\!\}; \Sigma \rightsquigarrow P \triangleright M} \quad (46)$$

$$\frac{\Gamma; \{\!\!\}; \mathcal{L}; \bar{\beta} \vdash \text{mod} : \{\!\!\}; \Sigma, \ell' : |\Sigma'| \rightsquigarrow X \triangleright M}{\Gamma; \mathcal{L}; \bar{\beta} \vdash \text{mod}.l : \Sigma \rightsquigarrow P \triangleright \text{let } X' = \text{Create}(\{\!\!\}' : |\Sigma'| \rightsquigarrow) \text{ in let } X = \{\ell = P, \ell' = X'.\ell'\} \text{ in } M} \quad (47)$$

$$\frac{\begin{array}{c} \vdash \mathcal{L}_1 \text{ locates } \bar{\alpha}_1 \quad \Gamma; \mathcal{L}_1 \uplus \mathcal{L}'_1; \bar{\beta}_1 \vdash \text{mod}_1 : \Sigma_1 \rightsquigarrow X_1 \triangleright M_1 \\ \vdash \mathcal{L}_2 \text{ locates } \bar{\alpha}_2 \quad \Gamma, X_1 : \Sigma_1; \mathcal{L}_2 \uplus \mathcal{L}'_2; \bar{\beta}_2 \vdash \text{mod}_2 : \Sigma_2 \quad \bar{\alpha}_1, \bar{\alpha}_2 \text{ fresh} \\ \vdash (\mathcal{L}_1; \Sigma_1) \rightleftharpoons (\mathcal{L}_2; \Sigma_2) \rightsquigarrow \delta \quad \Gamma, X_1 : \delta \Sigma_1; \delta \mathcal{L}_2 \uplus \mathcal{L}'_2; \bar{\beta}_2 \vdash \text{mod}_2 : \Sigma'_2 \rightsquigarrow X_2 \triangleright M_2 \quad \vdash \delta \Sigma_1 + \Sigma'_2 \Rightarrow \Sigma \rightsquigarrow P \triangleright M'_1, M'_2 \end{array}}{\Gamma; \mathcal{L}'_1 \cup \mathcal{L}'_2; \bar{\beta}_1, \bar{\beta}_2 \vdash (X_1 = \text{mod}_1) \text{ with } \text{mod}_2 : \Sigma \rightsquigarrow P \triangleright \text{let } X_1 = M'_1 \text{ in let } X_2 = M'_2 \text{ in } \delta M_1; M_2} \quad (48)$$

$$\frac{\begin{array}{c} \vdash \mathcal{L}_1 \text{ locates } \bar{\alpha}_1 \quad \Gamma; \mathcal{L}_1; \bar{\beta}_1 \vdash \text{mod}_1 : \Sigma_1 \rightsquigarrow X_1 \triangleright M_1 \\ \vdash \mathcal{L}_2 \text{ locates } \bar{\alpha}_2 \quad \Gamma, X_1 : \Sigma_1; \mathcal{L}_2; \bar{\beta}_2 \vdash \text{mod}_2 : \Sigma_2 \quad \bar{\alpha}_2, \bar{\beta}_2 \text{ fresh} \\ \vdash (\mathcal{L}_1; \Sigma_1) \rightleftharpoons (\mathcal{L}_2; \Sigma_2) \rightsquigarrow \delta \quad \delta \Gamma, X_1 : \delta \Sigma_1; \delta \mathcal{L}_2; \bar{\beta}_2 \vdash \text{mod}_2 : \Sigma'_2 \rightsquigarrow X_2 \triangleright M_2 \quad \vdash \delta \Sigma_1 + \Sigma'_2 \Rightarrow |\Sigma| \rightsquigarrow X \triangleright M'_1, M'_2 \end{array}}{\Gamma; \{\!\!\}; \bar{\beta}_1, \bar{\alpha}_1 \vdash (X_1 = \text{mod}_1) \text{ seals } \text{mod}_2 : |\Sigma_1| \rightsquigarrow P \triangleright \text{new } \bar{\beta}_2 \text{ in def } \bar{\alpha}_1 := \delta \bar{\alpha}_1 \text{ in let } X = \text{Create}(|\Sigma|) \text{ in let } X_1 = M'_1 \text{ in let } X_2 = M'_2 \text{ in Copy}(X_1, P : |\Sigma_1|); \delta M_1; M_2} \quad (49)$$

$$\frac{\Gamma \vdash \text{tyc} \rightsquigarrow B : K \quad \vdash A : K}{\Gamma; \{\!\!\}; \llbracket = A \rrbracket; \emptyset \vdash \{\ell \approx \text{tyc}\} : \{\!\!\}; A \approx B \rightsquigarrow P \triangleright \{\}} \quad (50)$$

$$\frac{\Gamma \vdash \text{tyc} \rightsquigarrow B : K \quad \vdash \beta : K}{\Gamma; \{\!\!\}; \beta \vdash \{\ell \approx \text{tyc}\} : \{\!\!\}; \beta \approx B \rightsquigarrow P \triangleright \text{def } \beta \approx B \text{ in } P.l \text{ in} := [\text{fold}_\beta]; P.l \text{ out} := [\text{unfold}_\beta]} \quad (51)$$

$$\frac{\Gamma \vdash \text{mod} : \Phi \rightsquigarrow F}{\Gamma; \{\!\!\}; \emptyset \vdash \llbracket \text{mod} \rrbracket : \llbracket \Phi \rrbracket^+ \rightsquigarrow P \triangleright P := F} \quad (52) \quad \frac{\Gamma \vdash \text{usig} \rightsquigarrow \Phi}{\Gamma; \{\!\!\}; \emptyset \vdash \llbracket : \text{usig} \rrbracket : \llbracket \Phi \rrbracket^- \rightsquigarrow P \triangleright \{\}} \quad (53)$$

$$\frac{\Gamma \vdash \text{mod} : \llbracket \forall \bar{\alpha}. \exists \bar{\beta}. (\mathcal{L}; \Sigma) \rrbracket^+ \rightsquigarrow M \quad \text{dom}(\delta) = \{\bar{\alpha}, \bar{\beta}\}}{\Gamma; \delta \mathcal{L}; \delta \bar{\beta} \vdash \text{new } \text{mod} : \delta \Sigma \rightsquigarrow P \triangleright (!M)[\delta \bar{\alpha}][\delta \bar{\beta}](P)} \quad (54)$$

Figure 8. Evidence Translation Rules for MixML

Signature Merging: $\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma \rightsquigarrow P \triangleright M_1, M_2$

$$\frac{\vdash \Sigma_2 + \Sigma_1 \Rightarrow \Sigma \rightsquigarrow P \triangleright M_2, M_1}{\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma \rightsquigarrow P \triangleright M_1, M_2} \quad (55) \quad \frac{}{\vdash \llbracket = A \rrbracket + \llbracket = A \rrbracket \Rightarrow \llbracket = A \rrbracket \rightsquigarrow P \triangleright P, P} \quad (56)$$

$$\frac{\vdash \tau_1 \leq \tau_2 \rightsquigarrow f}{\vdash \llbracket \tau_1 \rrbracket^\pm + \llbracket \tau_2 \rrbracket^\pm \Rightarrow \llbracket \tau_1 \rrbracket^\pm \rightsquigarrow P \triangleright P, \text{lazy}(f(\text{Val}(!P))) : \llbracket \tau_2 \rrbracket} \quad (57)$$

$$\frac{\vdash \Phi_1 \leq \Phi_2 \rightsquigarrow F}{\vdash \llbracket \Phi_1 \rrbracket^+ + \llbracket \Phi_2 \rrbracket^- \Rightarrow \llbracket \Phi_1 \rrbracket^+ \rightsquigarrow P \triangleright P, \text{lazy}(F(!P)) : \Phi_2^\circ} \quad (58) \quad \frac{}{\vdash \llbracket \Phi \rrbracket^- + \llbracket \Phi \rrbracket^- \Rightarrow \llbracket \Phi \rrbracket^- \rightsquigarrow P \triangleright P, P} \quad (59)$$

$$\frac{}{\vdash \Sigma + \{\!\!\} \Rightarrow \Sigma \rightsquigarrow P \triangleright P, \{\!\!\}} \quad (60)$$

$$\frac{\ell \notin \bar{\ell}_2 \quad \vdash \{\!\!\ell_1 : \Sigma_1\!\!\} + \{\!\!\ell_2 : \Sigma_2\!\!\} \Rightarrow \{\!\!\ell_3 : \Sigma_3\!\!\} \rightsquigarrow X \triangleright \{\overline{\ell_1 = M_1}\}, M_2}{\vdash \{\!\!\ell : \Sigma, \bar{\ell}_1 : \Sigma_1\!\!\} + \{\!\!\ell_2 : \Sigma_2\!\!\} \Rightarrow \{\!\!\ell : \Sigma, \bar{\ell}_3 : \Sigma_3\!\!\} \rightsquigarrow P \triangleright \text{let } X = \{\bar{\ell}_3 = P.\bar{\ell}_3\} \text{ in } \{\ell = P.\bar{\ell}, \bar{\ell}_1 = M_1\}, \text{let } X = \{\bar{\ell}_3 = P.\bar{\ell}_3\} \text{ in } M_2} \quad (61)$$

$$\frac{\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma_3 \rightsquigarrow P.\ell \triangleright M_1, M_2 \quad \vdash \{\!\!\bar{\ell}_1 : \Sigma'_1\!\!\} + \{\!\!\bar{\ell}_2 : \Sigma'_2\!\!\} \Rightarrow \{\!\!\bar{\ell}_3 : \Sigma'_3\!\!\} \rightsquigarrow X \triangleright \{\overline{\ell_1 = M'_1}, \overline{\ell_2 = M'_2}\}}{\vdash \{\!\!\ell : \Sigma_1, \bar{\ell}_1 : \Sigma'_1\!\!\} + \{\!\!\ell : \Sigma_2, \bar{\ell}_2 : \Sigma'_2\!\!\} \Rightarrow \{\!\!\ell : \Sigma_3, \bar{\ell}_3 : \Sigma'_3\!\!\} \rightsquigarrow P \triangleright \text{let } X = \{\bar{\ell}_3 = P.\bar{\ell}_3\} \text{ in } \{\ell = M_1, \bar{\ell}_1 = M'_1.\bar{\ell}_1\}, \text{let } X = \{\bar{\ell}_3 = P.\bar{\ell}_3\} \text{ in } \{\ell = M_2, \bar{\ell}_2 = M'_2.\bar{\ell}_2\}} \quad (62)$$

Unit Signature Matching: $\vdash \Phi_1 \leq \Phi_2 \rightsquigarrow F$

$$\frac{\vdash (\mathcal{L}_1^-; \Sigma_1) \rightleftharpoons (\mathcal{L}_2^+; \Sigma_2) \rightsquigarrow \delta \quad \vdash \delta \Sigma_1 + -\delta \Sigma_2 \Rightarrow |\Sigma| \rightsquigarrow X \triangleright M_1, M_2}{\vdash \forall \bar{\alpha}_1. \exists \bar{\beta}_1. (\mathcal{L}_1^-; \mathcal{L}_1^+; \Sigma_1) \leq \forall \bar{\alpha}_2. \exists \bar{\beta}_2. (\mathcal{L}_2^-; \mathcal{L}_2^+; \Sigma_2) \rightsquigarrow \lambda F : (\forall \bar{\alpha}_1. \exists \bar{\beta}_1. (\mathcal{L}_1^-; \mathcal{L}_1^+; \Sigma_1))^\circ . \Lambda \bar{\alpha}_2. \Lambda^\dagger \bar{\beta}_2. \lambda X_2 : \Sigma_2^\circ . \text{new } \bar{\beta}_1 \text{ in def } \bar{\beta}_2 := \delta \bar{\beta}_2 \text{ in let } X = \text{Create}(|\Sigma|) \text{ in let } X'_2 = M_2 \text{ in Copy}(X'_2, X_2 : \Sigma_2); F[\delta \bar{\alpha}_1][\bar{\beta}_1](M_1)} \quad (63)$$

Auxiliary Definitions:

$$\Delta \vdash \mathcal{J} \stackrel{\text{def}}{\iff} \vdash \mathcal{J} \wedge \text{FV}(\mathcal{J}) \subseteq \text{dom}(\Delta) \quad (\text{for } \mathcal{J} \in \{A : K, \Sigma \Downarrow, \Sigma \Uparrow, \Phi \Downarrow, \Phi \Uparrow, \Gamma \Downarrow, \Gamma \Uparrow\})$$

$\llbracket = A \rrbracket^\circ$	$\stackrel{\text{def}}{=} \llbracket = A \rrbracket$	$\text{stat}(\llbracket = A \rrbracket)$	$\stackrel{\text{def}}{=} \llbracket = A \rrbracket$
$\llbracket \tau \rrbracket^\pm$	$\stackrel{\text{def}}{=} \mathcal{S}[\tau]$	$\text{stat}(\llbracket \tau \rrbracket^\pm)$	$\stackrel{\text{def}}{=} \{\!\!\}$
$\llbracket \Phi \rrbracket^\pm$	$\stackrel{\text{def}}{=} \mathcal{S}\Phi^\circ$	$\text{stat}(\llbracket \Phi \rrbracket^\pm)$	$\stackrel{\text{def}}{=} \llbracket \text{stat}(\Phi) \rrbracket^\pm$
$\{\!\!\ell : \Sigma\!\!\}^\circ$	$\stackrel{\text{def}}{=} \{\!\!\ell : \Sigma^\circ\!\!\}$	$\text{stat}(\{\!\!\ell : \Sigma\!\!\})$	$\stackrel{\text{def}}{=} \{\!\!\ell : \text{stat}(\Sigma)\!\!\}$
$(\forall \bar{\alpha}. \exists \bar{\beta}. (\mathcal{L}_1; \mathcal{L}_2; \Sigma))^\circ$	$\stackrel{\text{def}}{=} \forall \bar{\alpha}. \exists \bar{\beta}. \Sigma^\circ \rightarrow \{\!\!\}$	$\text{stat}(\forall \bar{\alpha}. \exists \bar{\beta}. (\mathcal{L}_1; \mathcal{L}_2; \Sigma))$	$\stackrel{\text{def}}{=} \forall \bar{\alpha}. \exists \bar{\beta}. (\mathcal{L}_1; \mathcal{L}_2; \text{stat}(\Sigma))$
$\text{Create}(\llbracket = A \rrbracket)$	$\stackrel{\text{def}}{=} [A]$	$\text{Copy}(P_-, P_+ : \llbracket = A \rrbracket)$	$\stackrel{\text{def}}{=} \{\!\!\}$
$\text{Create}(\llbracket \tau \rrbracket^\pm)$	$\stackrel{\text{def}}{=} [\text{new}(\llbracket \tau \rrbracket)]$	$\text{Copy}(P_-, P_+ : \llbracket \tau \rrbracket^\pm)$	$\stackrel{\text{def}}{=} P_\pm := !P_\mp$
$\text{Create}(\llbracket \Phi \rrbracket^\pm)$	$\stackrel{\text{def}}{=} [\text{new}(\Phi^\circ)]$	$\text{Copy}(P_-, P_+ : \llbracket \Phi \rrbracket^\pm)$	$\stackrel{\text{def}}{=} P_\pm := !P_\mp$
$\text{Create}(\{\!\!\ell : \Sigma\!\!\})$	$\stackrel{\text{def}}{=} \{\ell : \text{Create}(\Sigma)\}$	$\text{Copy}(P_-, P_+ : \{\!\!\ell : \Sigma\!\!\})$	$\stackrel{\text{def}}{=} \overline{\text{Copy}(P_-. \ell, P_+. \ell : \Sigma)}$

Figure 9. Evidence Translation for Signature Mixing and Matching

Synthesis Signatures: $\vdash \Sigma \Uparrow \quad \vdash \Phi \Uparrow$

$$\frac{\vdash A : K}{\vdash \llbracket = A \rrbracket \Uparrow} \quad \frac{\vdash \tau : 0}{\vdash \llbracket \tau \rrbracket^\pm \Uparrow} \quad \frac{\vdash \Phi \Uparrow}{\vdash \llbracket \Phi \rrbracket^+ \Uparrow} \quad \frac{\vdash \Phi \Downarrow}{\vdash \llbracket \Phi \rrbracket^- \Uparrow} \quad \frac{\overline{\vdash \Sigma \Uparrow}}{\vdash \{\!\!\ell : \Sigma\!\!\} \Uparrow} \quad \frac{\vdash \Sigma \Uparrow \quad \vdash \mathcal{L} \text{ locates } \bar{\alpha} \quad \mathcal{L} \subseteq \Sigma}{\vdash \forall \bar{\alpha}. \exists \bar{\beta}. (\mathcal{L}; \Sigma) \Uparrow}$$

Analysis Signatures: $\vdash \Sigma \Downarrow \quad \vdash \Phi \Downarrow$

$$\frac{\vdash A : K}{\vdash \llbracket = A \rrbracket \Downarrow} \quad \frac{\vdash \tau : 0}{\vdash \llbracket \tau \rrbracket^\pm \Downarrow} \quad \frac{\vdash \Phi \Downarrow}{\vdash \llbracket \Phi \rrbracket^\pm \Downarrow} \quad \frac{\overline{\vdash \Sigma \Downarrow}}{\vdash \{\!\!\ell : \Sigma\!\!\} \Downarrow} \quad \frac{\vdash \Sigma \Downarrow \quad \vdash \mathcal{L}^- \text{ locates } \bar{\alpha} \quad \vdash \mathcal{L}^+ \text{ locates } \bar{\beta} \quad \mathcal{L}^- \subseteq \Sigma \quad \mathcal{L}^+ \subseteq \Sigma}{\vdash \forall \bar{\alpha}. \exists \bar{\beta}. (\mathcal{L}^-; \mathcal{L}^+; \Sigma) \Downarrow}$$

Figure 10. Synthesis and Analysis Signatures

Theorem B.1 (Properties of Typechecking and Translation)

Suppose $\Delta \vdash \Gamma \uparrow$ and $\Delta \vdash \mathcal{L}^\circ$ and $\bar{\beta} \subseteq \Delta$. Also, we assume the following about core subtyping:

If $\vdash \tau_1 \leq \tau_2$ and $\Delta \vdash \tau_1 : 0$ and $\Delta \vdash \tau_2 : 0$, then there exists f such that $\vdash \tau_1 \leq \tau_2 \rightsquigarrow f$ and $\Delta; \emptyset \vdash f : \tau_1 \rightarrow \tau_2$.

Then:

1. If $\Gamma \vdash \text{tyc} \rightsquigarrow A$, then there exists a K such that $\vdash A : K$.
2. If $\Gamma \vdash \text{tyc} \rightsquigarrow A : K$, then $\vdash A : K$.
3. If $\Gamma \vdash \text{exp} : \tau$, then there exists e such that $\Gamma \vdash \text{exp} : \tau \rightsquigarrow e$ and $\Delta; \Gamma^\circ \vdash e : \tau$.
4. If $\Gamma \vdash \text{mod} : \Sigma$, then $\Delta \vdash \Sigma \uparrow$ and there exists M such that $\Gamma \vdash \text{mod} : \Sigma \rightsquigarrow M$ and $\Delta; \Gamma^\circ \vdash M : \Sigma^\circ$.
Furthermore, $\Gamma \vdash_{\text{stat}} \text{mod} : \Sigma'$ with $\text{stat}(\Sigma) = \text{stat}(\Sigma')$.
5. If $\Gamma \vdash \text{mod} : \Phi$, then $\Delta \vdash \Phi \uparrow$ and there exists F such that $\Gamma \vdash \text{mod} : \Phi \rightsquigarrow F$ and $\Delta; \Gamma^\circ \vdash F : \Phi^\circ$.
Furthermore, $\Gamma \vdash_{\text{stat}} \text{mod} : \Phi'$ with $\text{stat}(\Phi) = \text{stat}(\Phi')$.
6. If $\Gamma; \mathcal{L}; \bar{\beta} \vdash \text{mod} : \Sigma$, then $\Delta \vdash \Sigma \uparrow$ and $\mathcal{L} \subseteq \Sigma$, and further if $\Delta; \Gamma'; \emptyset \vdash P : \Sigma^\circ$ for some $\Gamma' \supseteq \Gamma^\circ$ with $\Delta \vdash \Gamma'$, then there exists M such that $\Gamma; \mathcal{L}; \bar{\beta} \vdash \text{mod} : \Sigma \rightsquigarrow P \triangleright M$ and $\Delta; \Gamma'; \bar{\beta} \vdash M : \{\!\!\}\}$.
Furthermore, $\Gamma; \mathcal{L}; \bar{\beta} \vdash_{\text{stat}} \text{mod} : \Sigma'$ with $\text{stat}(\Sigma) = \text{stat}(\Sigma')$.
7. If $\Gamma \vdash_{\text{stat}} \text{mod} : \Sigma$, then $\Delta \vdash \Sigma \uparrow$.
8. If $\Gamma \vdash_{\text{stat}} \text{mod} : \Phi$, then $\Delta \vdash \Phi \uparrow$.
9. If $\Gamma; \mathcal{L}; \bar{\beta} \vdash_{\text{stat}} \text{mod} : \Sigma$, then $\Delta \vdash \Sigma \uparrow$ and $\mathcal{L} \subseteq \Sigma$.
10. If $\Gamma \vdash \text{usig} \rightsquigarrow \Phi$, then $\Delta \vdash \Phi \downarrow$.
11. If $\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma$ and $\Delta \vdash \Sigma_1 \uparrow$ and $\Delta \vdash \Sigma_2 \uparrow$, then $\Delta \vdash \Sigma \uparrow$, and further if $\Delta; \Gamma'; \emptyset \vdash P : \Sigma^\circ$ for some Γ' with $\Delta \vdash \Gamma'$, then there exist M_1, M_2 such that $\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma \rightsquigarrow P \triangleright M_1, M_2$, and $\Delta; \Gamma' \vdash M_1 : \Sigma_1^\circ$ and $\Delta; \Gamma' \vdash M_2 : \Sigma_2^\circ$.
Also, if $\mathcal{L}_1 \subseteq \Sigma_1$ and $\mathcal{L}_2 \subseteq \Sigma_2$, then $\mathcal{L}_1 \cup \mathcal{L}_2 \subseteq \Sigma$.
Furthermore, $\vdash \text{stat}(\Sigma_1) + \text{stat}(\Sigma_2) \Rightarrow \text{stat}(\Sigma)$.
12. If $\vdash \Phi_1 \leq \Phi_2$ and $\Delta \vdash \Phi_1 \uparrow$ and $\Delta \vdash \Phi_2 \downarrow$, then there exists F such that $\vdash \Phi_1 \leq \Phi_2 \rightsquigarrow F$ and $\Delta; \emptyset \vdash F : \Phi_1 \rightarrow \Phi_2$.
Furthermore, $\vdash \text{stat}(\Phi_1) \leq \text{stat}(\Phi_2)$.
13. If $\vdash (\mathcal{L}_1; \Sigma_1) \rightleftharpoons (\mathcal{L}_2; \Sigma_2) \rightsquigarrow \delta$ and $\vdash \Sigma_1 \uparrow$ and $\vdash \Sigma_2 \uparrow$ and $\vdash \mathcal{L}_1$ locates $\bar{\alpha}_1$ and $\vdash \mathcal{L}_2$ locates $\bar{\alpha}_2$, then $\Delta, \bar{\alpha}_1, \bar{\alpha}_2 \vdash \delta : \Delta$ and $\delta \mathcal{L}_1 \subseteq \delta \Sigma_2$ and $\delta \mathcal{L}_2 \subseteq \delta \Sigma_1$.
Furthermore, $\vdash (\mathcal{L}_1; \text{stat}(\Sigma_1)) \rightleftharpoons (\mathcal{L}_2; \text{stat}(\Sigma_2)) \rightsquigarrow \delta$.

As an additional property, we have the following:

Theorem B.2 (Substitution)

1. For all derivable judgements of the form $\vdash \mathcal{J}$ and substitutions δ , the judgement $\vdash \delta \mathcal{J}$ is derivable.
2. For all derivable judgements of the form $\Gamma \vdash \mathcal{J}$ and substitutions δ , the judgement $\delta \Gamma \vdash \delta \mathcal{J}$ is derivable.
3. For all derivable judgements of the form $\Gamma; \mathcal{L}; \bar{\beta} \vdash \mathcal{J}$ and substitutions δ with $\bar{\delta} \bar{\beta} = \bar{\beta}'$ for some $\bar{\beta}'$, the judgement $\delta \Gamma; \delta \mathcal{L}; \bar{\delta} \bar{\beta} \vdash \delta \mathcal{J}$ is derivable.

(Here, $\mathcal{C} \vdash \mathcal{J}$ is meant to include static judgements $\mathcal{C} \vdash_{\text{stat}} \mathcal{J}'$.)

This property is not necessary for soundness, but it has the useful implication that in the static pass, Rule 17 does not actually have to check mod_2 twice, because Σ_2' is already known to be derivable, and equal to $\delta \Sigma_2$. This observation decreases the complexity of type checking.

C. Algorithmic Type Checking and Decidability

For practical purposes it is important that type checking can be performed algorithmically. For most part, this is easy to see for our type system, because the typing rules for MixML are syntax-directed. The only relevant source of non-determinism is the need

Template Module Signatures	$S ::= \llbracket K \rrbracket \mid \llbracket U \rrbracket^\pm \mid \{\!\!\bar{\ell} : \bar{S}\!\!\}$
Template Unit Signatures	$U ::= (L; \bar{K}; S)$
Template Type Locators	$L ::= \llbracket K \rrbracket \mid \{\!\!\bar{\ell} : \bar{L}\!\!\}$

$$\begin{aligned}
(\llbracket = A \rrbracket)^\top &= \llbracket K \rrbracket && \text{if } \vdash A : K \\
(\llbracket \tau \rrbracket^\pm)^\top &= \{\!\!\}\} \\
(\llbracket \Phi \rrbracket^\pm)^\top &= \llbracket \Phi^\top \rrbracket^\pm \\
\{\!\!\bar{\ell} : \bar{\Sigma}\!\!\}^\top &= \{\!\!\bar{\ell} : \bar{\Sigma}^\top\!\!\} \\
(\forall \bar{\alpha}. \exists \bar{\beta}. (\mathcal{L}_1; \mathcal{L}_2; \Sigma))^\top &= (\mathcal{L}_1^\top; \bar{K}; \Sigma^\top) && \text{if } \vdash \bar{\beta} : \bar{K} \\
S_1 + S_2 &= S_2 + S_1 \\
S + \{\!\!\}\} &= S \\
\llbracket K \rrbracket + \llbracket K \rrbracket &= \llbracket K \rrbracket \\
\llbracket U_1 \rrbracket^\pm + \llbracket U_2 \rrbracket^\pm &= \llbracket U_1 \rrbracket^\pm \\
\{\!\!\bar{\ell} : \bar{S}_1, \bar{\ell}_1 : \bar{S}'_1\!\!\} + \{\!\!\bar{\ell} : \bar{S}_2, \bar{\ell}_2 : \bar{S}'_2\!\!\} &= \{\!\!\bar{\ell} : \bar{S}_1 + \bar{S}_2, \bar{\ell}_1 : \bar{S}'_1, \bar{\ell}_2 : \bar{S}'_2\!\!\} \\
&&& \text{where } \bar{\ell}_1 \cap \bar{\ell}_2 = \emptyset
\end{aligned}$$

Figure 11. Template Objects and Auxiliary Definitions

to choose appropriate new locators \mathcal{L} and export variables $\bar{\beta}$ in some of the rules.

Figure 12 specifies a straightforward algorithm for computing suitable choices by a simple recursive pass over the module expression being checked. It returns a *template signature* S , which essentially is a semantic signature with all type information erased, and similarly, a *template type locator* L , in the same style. Both are defined in Figure 11, along with a suitable erasure $(\cdot)^\top$ from semantic objects into corresponding template objects. Third, the algorithm returns a list of kinds for the export type names of the module (without choosing actual names).

In the rules, we write $\text{dom}(S)$ to denote all paths defined in a template signature or locator S . Further, we use the notation $L - \bar{\ell}s$ to describe the template locator L with all subcomponents removed whose path is in $\bar{\ell}s$.

The following properties are easy to show for the algorithm, showing that it is complete with respect to the typing judgement – and thereby obtaining decidability of the type system.

Theorem C.1 (Completeness of Template Computation)

Suppose $\vdash \Gamma \uparrow$ and $\vdash \mathcal{L}$ locates $\bar{\alpha}$. Then:

1. If $\Gamma \vdash \text{tyc} \rightsquigarrow A$ and $\vdash A : K$, then $\Gamma^\top \vdash \text{tyc} \Rightarrow K$.
2. If $\Gamma \vdash \text{tyc} \rightsquigarrow A : K$, then $\Gamma^\top \vdash \text{tyc} \Rightarrow K$.
3. If $\Gamma \vdash \text{mod} : \Sigma$, then there exist \bar{K} such that $\Gamma^\top \vdash \text{mod} \Rightarrow \{\!\!\bar{\ell}\!\!\}; \bar{K}; \Sigma^\top$.
4. If $\Gamma; \mathcal{L}; \bar{\beta} \vdash \text{mod} : \Sigma$, then there exist \bar{K} with $\bar{\beta} : \bar{K}$ such that $\Gamma^\top \vdash \text{mod} \Rightarrow \mathcal{L}^\top; \bar{K}; \Sigma^\top$.
5. If $\Gamma \vdash \text{mod} : \Phi$, then $\Gamma^\top \vdash \text{mod} \Rightarrow U^\top$.
6. If $\Gamma \vdash \text{usig} \rightsquigarrow \Phi$, then $\Gamma^\top \vdash \text{usig} \Rightarrow \Phi^\top$.
7. If $\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma$, then $\Sigma_1^\top + \Sigma_2^\top = \Sigma^\top$.

Type Constructor Templates: $\Gamma^T \vdash tyc \Rightarrow K$

$$\frac{\Gamma^T \vdash mod \Rightarrow \{\}; \bar{K}; \llbracket K' \rrbracket}{\Gamma^T \vdash \mathbf{Tyc}(mod) \Rightarrow K'} \quad \frac{}{\Gamma^T \vdash \alpha \Rightarrow 0} \quad \frac{\bar{\alpha} = \alpha_1, \dots, \alpha_n}{\Gamma^T \vdash \lambda(\bar{\alpha}).tyc \Rightarrow n} \quad \frac{}{\Gamma^T \vdash tyc'(\bar{tyc}) \Rightarrow 0}$$

Module Templates: $\Gamma^T \vdash mod \Rightarrow U$

$$\frac{X:S \in \Gamma^T}{\Gamma^T \vdash X \Rightarrow \{\}; \emptyset; S} \quad \frac{}{\Gamma^T \vdash \{\} \Rightarrow \{\}; \emptyset; \{\}} \quad \frac{}{\Gamma^T \vdash [: K] \Rightarrow \llbracket K \rrbracket; \emptyset; \llbracket K \rrbracket} \quad \frac{\Gamma^T \vdash tyc \Rightarrow K}{\Gamma^T \vdash [tyc] \Rightarrow \{\}; \emptyset; \llbracket K \rrbracket}$$

$$\frac{}{\Gamma^T \vdash [: tyc] \Rightarrow \{\}; \emptyset; \{\}} \quad \frac{}{\Gamma^T \vdash [exp] \Rightarrow \{\}; \emptyset; \{\}}$$

$$\frac{\Gamma^T \vdash mod \Rightarrow L; \bar{K}; S}{\Gamma^T \vdash \{\ell = mod\} \Rightarrow \{\ell : L\}; \bar{K}; \{\ell : S\}} \quad \frac{\Gamma^T \vdash mod \Rightarrow \{\ell : L\}; \bar{K}; \{\ell : S, \ell' : S'\}}{\Gamma^T \vdash mod.\ell \Rightarrow L; \bar{K}; S}$$

$$\frac{\Gamma^T \vdash mod_1 \Rightarrow L \uplus L_1; \bar{K}_1; S_1 \quad \Gamma^T, X : S_1 \vdash mod_2 \Rightarrow L \uplus L_2; \bar{K}_2; S_2 \quad \text{dom}(L_1) \cap \text{dom}(L_2) = \emptyset}{\Gamma^T \vdash (X = mod_1) \text{ with } mod_2 \Rightarrow L \uplus (L_1 - \text{dom}(S_2)) \uplus (L_2 - \text{dom}(S_1)); \bar{K}_1, \bar{K}_2; S_1 + S_2}$$

$$\frac{\Gamma^T \vdash mod_1 \Rightarrow L; \bar{K}; S \quad \vdash L \text{ locates } \bar{K}'}{\Gamma^T \vdash (X = mod_1) \text{ seals } mod_2 \Rightarrow \{\}; \bar{K}, \bar{K}'; S} \quad \frac{\Gamma^T \vdash tyc \Rightarrow K}{\Gamma^T \vdash \{ : \ell \approx tyc \} \Rightarrow \{\ell : \llbracket K \rrbracket\}; \emptyset; \{\ell : \llbracket K \rrbracket\}} \quad \frac{\Gamma^T \vdash tyc \Rightarrow K}{\Gamma^T \vdash \{\ell \approx tyc\} \Rightarrow \{\}; K; \{\ell : \llbracket K \rrbracket\}}$$

$$\frac{\Gamma^T \vdash mod \Rightarrow U}{\Gamma^T \vdash [mod] \Rightarrow \{\}; \emptyset; \llbracket U \rrbracket^+} \quad \frac{\Gamma^T \vdash mod \Rightarrow \{\}; \bar{K}; \llbracket U \rrbracket^+}{\Gamma^T \vdash \text{new } mod \Rightarrow U} \quad \frac{\Gamma^T \vdash usig \Rightarrow U}{\Gamma^T \vdash [: usig] \Rightarrow \{\}; \emptyset; \llbracket U \rrbracket^-}$$

Unit Signature Templates: $\Gamma^T \vdash usig \Rightarrow U$

$$\frac{\Gamma^T \vdash mod \Rightarrow L; \emptyset; S \quad \vdash L - \bar{ls}^- \text{ locates } \bar{K} \quad \bar{ls}^- = \{\ell s \mid L.\ell s = \llbracket K \rrbracket \wedge \text{prefix}^?(\bar{ls}, \ell s)\}}{\Gamma^T \vdash mod \text{ import } \bar{ls} \Rightarrow L - \bar{ls}^+; \bar{K}; S} \quad \bar{ls}^+ = \{\ell s \mid L.\ell s = \llbracket K \rrbracket \wedge \neg \text{prefix}^?(\bar{ls}, \ell s)\}$$

$$\frac{\Gamma^T \vdash mod \Rightarrow L; \emptyset; S \quad \vdash L - \bar{ls}^- \text{ locates } \bar{K} \quad \bar{ls}^- = \{\ell s \mid L.\ell s = \llbracket K \rrbracket \wedge \neg \text{prefix}^?(\bar{ls}, \ell s)\}}{\Gamma^T \vdash mod \text{ export } \bar{ls} \Rightarrow L - \bar{ls}^+; \bar{K}; S} \quad \bar{ls}^+ = \{\ell s \mid L.\ell s = \llbracket K \rrbracket \wedge \text{prefix}^?(\bar{ls}, \ell s)\}$$

Type Locator Templates: $\vdash L \text{ locates } \bar{K}$

$$\frac{\vdash \mathcal{L} \text{ locates } \bar{\alpha} \quad \mathcal{L}^T = L \quad \overline{\vdash \alpha : \bar{K}}}{\vdash L \text{ locates } \bar{K}}$$

Figure 12. Template Computation