A

Mixin' Up the ML Module System

ANDREAS ROSSBERG, Google
DEREK DREYER, Max Planck Institute for Software Systems (MPI-SWS)

ML modules provide hierarchical namespace management, as well as fine-grained control over the propagation of type information, but they do not allow modules to be broken up into mutually recursive, separately compilable components. Mixin modules facilitate recursive linking of separately compiled components, but they are not hierarchically composable and typically do not support type abstraction. We synthesize the complementary advantages of these two mechanisms in a novel module system design we call 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 ML-style data abstraction, and mixin-style recursive linking. Moreover, the design of MixML is clean and minimalist; it emphasizes how all the salient, semantically interesting features of the ML module system (and several proposed extensions to it) can be understood simply as stylized uses of a small set of orthogonal underlying constructs, with mixin composition playing a central role.

We provide a declarative type system for MixML, including two important extensions: higher-order modules, and modules as first-class values. We also present a sound and complete, three-pass type-checking algorithm for this system. The operational semantics of MixML is defined by an elaboration translation into an internal core language called LTG—namely, a polymorphic lambda calculus with single-assignment references and recursive type generativity—which employs a linear type and kind system to track definedness of term and type imports.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Recursion, Abstract data types, Modules*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type structure*

General Terms: Languages, Design, Theory

Additional Key Words and Phrases: Type systems, ML modules, mixin modules, abstract data types, recursive modules, hierarchical composability

ACM Reference Format:
Rossberg, A., and Dreyer, D. 2011. Mixin' Up the ML Module System ACM Trans. Program. Lang. Syst. V, N, Article A (January YYYY), 84 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

ML modules and *mixin modules* are two well-known and influential mechanisms for modular programming that have largely complementary advantages and disadvantages. In this article, we show how to synthesize some of the defining aspects of these mechanisms in the design of a novel module system we call MixML.

We begin by reviewing some of the main features and drawbacks of ML modules and mixin modules.

Author's addresses: A. Rossberg, Google Germany, Dienerstr. 12, 80331 München, Germany, rossberg@mpi-sws.org; D. Dreyer, Max Planck Institute for Software Systems (MPI-SWS), Campus E1.5, 66123 Saarbrücken, Germany, dreyer@mpi-sws.org.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1.1. ML Modules

Proposed originally by MacQueen [1984] and developed further by Harper, Leroy, and many others [Harper and Lillibridge 1994; Leroy 1994; Russo 1998; Dreyer et al. 2003], the ML module system offers powerful support for flexible program construction, data abstraction, and code reuse. In ML, *structures* provide namespace management, *signatures* describe module interfaces, *functors* enable the definition of generic modules, and *opaque signature ascription* (a.k.a. *sealing*) lets one hide the implementation details of a module behind an interface.

One important feature of ML modules is that they are *hierarchically composable*. Structures may contain other structures as components, and thus be used to build hierarchical namespaces. Another important feature is that ML 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* [Harper and Lillibridge 1994], *i.e.,* they can specify type components of modules either abstractly or transparently. 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 [Crary et al. 1999; Russo 2001; Leroy 2003; Nakata and Garrigue 2006; Dreyer 2007b]. While the existing proposals address a variety of interesting issues, such as the interaction of recursion and data abstraction [Crary et al. 1999; Dreyer 2007b], none of them provides adequate support for something we view as a central design goal: *separate compilation*, *i.e.,* the ability to break big modules into smaller components that can be type-checked and compiled independently of one another and linked with multiple different implementations of the other components. The desire to do so is half the motivation for recursive modules, 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 hierarchical (*i.e.,* non-recursive) linking—namely, *functors*—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 in general need to refer recursively to abstract type components provided in its exports. Unfortunately, it is not obvious how to generalize the functor mechanism in a simple way in order to permit the argument to depend on the result.

1.2. Mixin Modules

Although the concept of *mixins* originated in work on Common LISP from the mid-1980s [Moon 1986], Bracha and Cook [1990] were the first to propose mixins as an actual language construct (in their case, as an extension to Modula-3). 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* (their terminology), *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 article, is also due to Bracha, in particular his work with Lindstrom on the Jigsaw language [Bracha and Lindstrom 1992]. 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 M such that

(1) $exports(\mathrm{M}) = exports(\mathrm{M}_1) \uplus exports(\mathrm{M}_2)$
(2) $imports(\mathrm{M}) = imports(\mathrm{M}_1) \cup imports(\mathrm{M}_2) - exports(\mathrm{M})$

Here, $\uplus$ denotes that the exports of M_1 and M_2 must be disjoint. In addition, the typing rule for the merge operator checks that any components with the same name in M_1 and M_2 have compatible types (for some suitable definition of "compatible", *e.g.,* the types are equal, or one is a subtype of the other).

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 (and "late binding") is often considered a central feature of mixins. Be that as it may, for the remainder of this article we will use the term *mixin composition* to mean Bracha's merge operator. Following mixin-based languages like Flatt et al.'s *units* [Flatt and Felleisen 1998; Owens and Flatt 2006] and Duggan's *recursive DLLs* [Duggan 2002], 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, 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 [Ancona and Zucca 2002; 1998; Ancona et al. 2003]. While CMS is a pure call-by-name language, it has been extended with support for call-by-value evaluation [Hirschowitz and Leroy 2005] and monadic effects [Ancona et al. 2003].

Compared with ML modules, a key advantage of mixin modules is that the mixin composition of modules M_1 and M_2 is by definition a kind of recursive linking, in which the exports of each module are used to satisfy the imports of the other. Mixin modules thus appear to offer a natural solution to the problems with separate compilation of recursive modules in ML. One major limitation of Bracha/CMS-style mixin modules, however, is that they contain only term components, not type components, which means that they cannot express type genericity or ML-style abstract data types, let alone translucent signatures. This has led a number of researchers to consider ways of combining the support for type abstraction found in ML modules with the support for separate compilation and recursive linking found in mixin modules [Flatt and Felleisen 1998; Duggan 2002; Odersky and Zenger 2005; Owens and Flatt 2006].

1.3. Motivation

The motivation for this article is that the existing proposals for synthesizing ideas from ML modules and mixin modules (which we will discuss in detail in Section 10) are all lacking in one key respect: none of them allows for a simple and direct encoding of all the salient, semantically interesting features of the ML module system. For example, Owens and Flatt [2006] give an encoding of ML-like modules into their *unit* language, but it depends critically on the impractical assumption that the ML programs being

encoded have (redundant) signature annotations on nearly every subexpression. Odersky et al.'s Scala language [Odersky and Zenger 2005; Cremet et al. 2006], while highly expressive in its support for OO-style extensibility, has only limited support for opaque signature ascription—*i.e.,* the ability to seal a module (or class) *ex post facto* behind an abstract interface—which is a central feature of the ML module system. Ideally, we would like a language that seamlessly integrates mixin composition into ML *without* sacrificing any key features of ML modules.

1.4. MixML

In this article, we present a novel foundational module system, called MixML, which incorporates at a deep level the mechanism of mixin module composition, while retaining the full expressive power of ML modules.

The main idea of MixML is simple: the MixML module language unifies ML's structure and signature languages into one. That is, a MixML module may contain both type and term *definitions*, of the kind found in ML structures, as well as type and term *specifications*, of the kind found in ML signatures. It is not required to contain only definitions or only specifications; rather, it may freely mix them. Thus, traditional ML structures and ML signatures may be viewed as endpoints on the spectrum of MixML modules.

Why is MixML's unification of structures and signatures useful? Because it enables us to encode a wide variety of features directly as stylized uses of a small set of orthogonal underlying constructs, thus simplifying and regularizing the design of the language. In particular:

(1) MixML provides a unifying account of several pairs of language constructs that are usually modeled as extensions to *both* the structure and signature languages of ML. Concretely, for each of the following pairs of features, MixML supports both features via a single encoding:
   — hierarchical structures and hierarchical signatures
   — recursive structures and recursively dependent signatures
   — functors and parameterized signatures
(2) A variety of features that are typically supported via distinct mechanisms may be encoded in MixML as idiomatic uses of mixin composition, *i.e.,* (recursive) linking. These include:
   — recursive module definitions
   — structure and signature inheritance (`open` and `include`)
   — signature refinement (`with`/`where`/`sharing`)
   — signature ascription, opaque (`:>`) and transparent (`:`)
   — functor application
   The encodings of these features involve the linking of two MixML modules, one or both of which represents an ML signature. These encodings are made possible by the fact that structures and signatures are just different kinds of MixML modules, and any two modules can be linked together so long as they are *compatible* (in a sense we make precise later in the article).

1.5. Technical Contributions

If the basic design idea of MixML is as simple and powerful as we claim, the reader may wonder why it has not been proposed before. We believe the reason is that the feasibility of the idea is dependent on several novel enhancements to mixin module semantics that we set forth in this article, as well as a generalization of some recent work on handling the "double vision" problem in the context of recursive modules. We briefly summarize these technical contributions here.

Hierarchical Composability. Suppose `M1` and `M2` 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 `M1` and `M2` were *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, in previous CMS-style mixin module systems, hierarchically composing two mixins to form another mixin is not possible. The reason is that CMS-style systems employ a flat namespace for their imports and exports—path names like `A1.x` are disallowed.

Hierarchical composability, which MixML modules support, allows us to use a single *namespace* mechanism to build hierarchies of structures, signatures, or modules that are a mixture of both. More importantly, linking is also hierarchical, *i.e.,* it applies not just to a flat module, but to all modules nested inside it. Without this more general mechanism, we would be unable to provide a unified representation of hierarchical structures and signatures.

Unifying Linking and Binding. In previous systems, the mixin composition of two modules does not provide a way for either of the modules to refer directly to components of the other. In other words, the linking operator is not a variable binder; instead, binding is typically built into other constructs.

In MixML, we take a different tack by making the linking operator the *only* binding construct. This enables us to (1) model all forms of binding in ML modules as stylized uses of linking, and (2) achieve very simple encodings of several features, such as recursive modules and `sharing` specifications. The benefit of unifying linking and binding will be borne out by a number of examples in Section 2.

Cross-Eyed Double Vision. A key problem that arises when extending ML with recursive modules is *double vision* [Crary et al. 1999; Dreyer 2005], *i.e.,* type aliasing through the interaction of recursion and type abstraction: when a recursive module `X` introduces a type name `t`, then inside the definition of `X`, the external type path `X.t` becomes an alias for the local type `t`—double vision arises when the type system fails to equate these two types. As MixML modules subsume the functionality of recursive ML modules, double vision is an issue for MixML as well. In fact, since mixin composition is essentially a bidirectional generalization of ML-style signature matching, the MixML type system must handle a "cross-eyed" version of the double vision problem.

Fortunately, in recent work, Dreyer [2007b; 2007a] has shown how to solve the double vision problem for recursive ML modules, and this solution can be generalized quite easily to handle the cross-eyed double vision problem for MixML modules. We describe the problem and its solution by example in Section 3, and provide full formal details of the solution in Section 4.

1.6. Differences from the Conference Version

This article is an extended version of our ICFP'08 paper of the same title [Dreyer and Rossberg 2008]. The present work offers a number of improvements on the conference version, as well as a wealth of additional material. Most notably:

— The language here supports *higher-kinded* type components.
— We describe how to integrate modules and units *as first-class values*.
— We give the rules and soundness proof for an *elaboration translation*, thereby defining the operational semantics of MixML and establishing type safety.

— The type system of the internal language (named *LTG*) that is targeted by the elaboration translation uses *linear reference types* in order to ensure that all term components of a module are defined once and only once. The type system also employs a novel, analogous notion of *linear kinds* to guarantee definedness of abstract types.
— We describe a sound and complete *algorithm for type-checking* MixML.

1.7. Overview

The rest of the article 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 explore several technical issues that the MixML type system must address, including the double vision problem and the handling of cyclic definitions. In Section 4, we give the formal definition of the MixML type system, in particular the static semantics. In Section 5, we extend MixML with support for higher-order modules, and in Section 6 we describe an extension that allows MixML modules to be packaged into first-class values.

The dynamic semantics of MixML is defined by translation into the internal language LTG. In Section 7 we define this internal language and prove it type-safe, and in Section 8 we give the actual translation of MixML (including the extensions from Sections 5 and 6), thus establishing type safety for it as well. In Section 9 we present a decidable algorithm for type-checking MixML and prove it sound and complete. Finally, in Section 10, we offer a detailed comparision with related work, and conclude with directions for future work.

This article focuses on the technical problem of synthesizing ML-style modules with mixin composition. Thus, for brevity, we assume basic familiarity with ML module programming; for those readers interested in a gentler introduction to ML-style module systems, there is a rich literature on the subject [MacQueen 1984; Harper 2011; Harper and Pierce 2005; Dreyer 2005; Russo 1998; Rossberg et al. 2010].

2. A TOUR OF MIXML

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 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 [2000], we define our module language to be 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 $\texttt{val}(mod)$, 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 $\texttt{typ}(mod)$, which takes an atomic type module mod (*i.e.,* a module containing a single type component) and projects out the type. For brevity, we will typically omit the explicit `typ` and `val` projections in examples when their necessity is clear from context, *e.g.,* we write just `M.t` as a type expression instead of $\texttt{typ}(\texttt{M.t})$.

We also assume that the type language contains type constructors, which take a type argument and return a type as a result. Type constructors can be higher-order, with kinds classifying them. Types classifying terms have kind `type`. Type variables are implicitly annotated with their kind, and we write knd_α to denote α's kind where necessary.

$$
\begin{array}{lrcl}
\textbf{Type Var's} & \alpha, \beta & \in & \mathit{TyVars} \times \mathit{Kinds} \\
\textbf{Module Var's} & \mathrm{X}, \mathrm{Y} & \in & \mathit{ModVars} \\
\textbf{Labels} & \ell & \in & \mathit{Labs} \\
\textbf{Paths} & \ell s & ::= & \epsilon \mid \ell.\ell s \in \mathit{Paths} \\
\textbf{Kinds} & \mathit{knd} & ::= & \texttt{type} \mid \mathit{knd} \to \mathit{knd} \in \mathit{Kinds} \\
\textbf{Types} & \mathit{typ} & ::= & \texttt{typ}(\mathit{mod}) \mid \alpha \mid \lambda\alpha.\mathit{typ} \mid \mathit{typ}_1\ \mathit{typ}_2 \mid \ldots \\
\textbf{Terms} & \mathit{exp} & ::= & \texttt{val}(\mathit{mod}) \mid \ldots \\
\textbf{Modules} & \mathit{mod} & ::= & \mathrm{X} \mid \{\} \mid \texttt{[:}\mathit{typ}\texttt{]} \mid \texttt{[}\mathit{exp}\texttt{]} \mid \texttt{[:}\mathit{knd}\texttt{]} \mid \texttt{[}\mathit{typ}\texttt{]} \mid \\
 & & & \{\ell = \mathit{mod}\} \mid \mathit{mod}.\ell \mid \texttt{[}\mathit{mod}\texttt{]} \mid \texttt{new}\ \mathit{mod} \mid \\
 & & & (\mathrm{X} = \mathit{mod}_1)\ \texttt{with}\ \mathit{mod}_2 \mid (\mathrm{X} = \mathit{mod}_1)\ \texttt{seals}\ \mathit{mod}_2
\end{array}
$$

$$
\begin{array}{rcl}
\overline{\mathit{knd}} \to \mathit{knd} & \stackrel{\text{def}}{=} & \mathit{knd}_1 \to \cdots \to \mathit{knd}_n \to \mathit{knd} \\
\lambda\overline{\alpha}.\mathit{typ} & \stackrel{\text{def}}{=} & \lambda\alpha_1.\cdots\lambda\alpha_n.\mathit{typ} \\
\mathit{typ}\ \overline{\mathit{typ}} & \stackrel{\text{def}}{=} & \mathit{typ}\ \mathit{typ}_1\ \cdots\ \mathit{typ}_n \\
\mathit{mod}_1\ \texttt{with}\ \mathit{mod}_2 & \stackrel{\text{def}}{=} & (\mathrm{X} = \mathit{mod}_1)\ \texttt{with}\ \mathit{mod}_2 \\
\mathit{mod}_1\ \texttt{seals}\ \mathit{mod}_2 & \stackrel{\text{def}}{=} & (\mathrm{X} = \mathit{mod}_1)\ \texttt{seals}\ \mathit{mod}_2
\end{array}
$$

where $\mathrm{X} \notin \mathrm{fv}(\mathit{mod}_2)$

Fig. 1. Basic MixML Syntax

Atomic Modules. Atomic modules are modules containing a single, anonymous 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 $\texttt{[:}\mathit{typ}\texttt{]}$ represents a term specification of type typ (a term import). The module $\texttt{[}\mathit{exp}\texttt{]}$ represents a term component defined to be the value resulting from the evaluation of exp (a term export). The module $\texttt{[:}\mathit{knd}\texttt{]}$ represents an abstract type specification of kind knd (a type import). The module $\texttt{[}\mathit{typ}\texttt{]}$ represents a transparent definition of a type component equal to typ (a type export). 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 Namespaces and Projection. The construct $\{\}$ denotes an empty module, containing no components. The module $\{\ell = \mathit{mod}\}$ introduces a namespace containing a single component named ℓ, whose definition is mod. Any imports (resp. exports) of mod become imports (resp. exports) of $\{\ell = \mathit{mod}\}$ as well, except the path names 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 named type and term definitions and specifications:

$$
\begin{array}{lcl}
\texttt{val v :}\ \mathit{typ} & \stackrel{\text{def}}{=} & \{\texttt{v} = \texttt{[:}\mathit{typ}\texttt{]}\} \\
\texttt{val v =}\ \mathit{exp} & \stackrel{\text{def}}{=} & \{\texttt{v} = \texttt{[}\mathit{exp}\texttt{]}\} \\
\texttt{type t}\ \overline{\alpha} & \stackrel{\text{def}}{=} & \{\texttt{t} = \texttt{[:}\overline{\mathit{knd}_\alpha} \to \texttt{type]}\} \\
\texttt{type t}\ \overline{\alpha}\ \texttt{=}\ \mathit{typ} & \stackrel{\text{def}}{=} & \{\texttt{t} = \texttt{[}\lambda\overline{\alpha}.\mathit{typ}\texttt{]}\}
\end{array}
$$

Here, $\overline{\alpha} = \alpha_1 \ldots \alpha_n$ denotes a (possibly empty) vector of type variables, and $\overline{knd_\alpha}$ is the vector of their respective kinds. We write $\overline{knd_\alpha} \to \mathtt{type}$ to abbreviate the kind $knd_{\alpha_1} \to \cdots \to knd_{\alpha_n} \to \mathtt{type}$.

Dual to $\{\ell = mod\}$ is the construct $mod.\ell$, which projects the ℓ component from the module mod. The typing rule for $mod.\ell$ insists that any imports mod 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 namespaces. 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 $(\mathrm{X} = mod_1)$ `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 (assuming they are compatible).
— 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 only means of variable binding in the language. It binds X as a representative of mod_1 inside mod_2. (We allow dropping the "X=" if X does not occur in mod_2.)

Compatibility of mod_1 and mod_2 reduces to compatibility of their atomic components. For each component of the same path name in both modules, compatibility is defined informally as follows:

— If the component is an import in mod_1 and an export in mod_2, then mod_2's export must match the import specification from mod_1. (And vice versa, if the component is an export in mod_1 and an import in mod_2.)
— If the component is a type import in both modules, they must both specify it to have the same kind.
— If the component is a type export in both modules, they must both define it to be the same type.
— If the component is a term import in both modules, and has specification typ_1 in mod_1 and typ_2 in mod_2, then either typ_1 must be a subtype of typ_2 (for some notion of core subtyping, *e.g.,* polymorphic instantiation) or vice versa, and whichever type is stronger is the one propagated as the specification of the component in the linked module.
— The component must not be a term export in both modules.[1]

One may wonder why the linking construct is asymmetric. There are two main reasons. First, as we will soon see, the asymmetric form of linking turns out to be sufficient for encoding recursive modules, with mod_1 acting as a kind of "forward declaration" for mod_2. Second, it is not clear how we could generalize this design to a symmetric linking construct while keeping type-checking decidable. (See further discussion at the end of the section on "double vision" in Section 3.)

Before exploring the recursive aspects of linking, we first show how it may be used to express n-ary non-recursive structures and signatures, as well as several other non-recursive features.

n-ary 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* [Harper and Lillibridge

[1] Allowing term exports to be mixed would give rise to OO-style *overriding*, which we exclude deliberately.

1994] (or the equivalent [Leroy 1994]). This divides the name of each component into a *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 expressed as

$$\{\ell_1 \triangleright \mathrm{X}_1 = mod_1, \ldots, \ell_n \triangleright \mathrm{X}_n = mod_n\}$$

where each X_i is bound in the subsequent mod_j's to the result of evaluating mod_i. In ML, ℓ_i and X_i must be the same identifier. We will also often omit the variable when it can be the same identifier as the label.

The encoding of an n-ary structure defines it as the linking of n disjoint unary structures (we assume here that X is suitably fresh):

$$\{\ell_1 \triangleright \mathrm{X}_1 = mod_1, \ldots\} \stackrel{\text{def}}{=} (\mathrm{X} = \{\ell_1 = mod_1\})\ \texttt{with}\ \{\ldots\}[\mathrm{X}.\ell_1/\mathrm{X}_1]$$

Inside the linking, X stands for the unary structure containing just the ℓ_1 component. In the encoding of the remainder of the components ($\ldots$), we must therefore replace references to X_1 with $\mathrm{X}.\ell_1$. We assume here for simplicity that all the ℓ_i are distinct labels. There are well-known ways of allowing for shadowing [Harper and Stone 2000; Rossberg et al. 2010], essentially by rewriting $\{\ell_1 \triangleright \mathrm{X}_1 = mod_1, \ldots\}$ to $\texttt{let}\ \mathrm{X}_1 = mod_1\ \texttt{in}\ \{\ldots\}$ if ℓ_1 is rebound in "$\ldots$" (see below for a definition of `let`).

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

$$\{\ell_1 \triangleright \mathrm{X}_1 : sig_1, \ldots, \ell_n \triangleright \mathrm{X}_n : sig_n\}$$

However, since ML signatures are encoded in MixML as modules (*i.e.,* a sig is just a module with no term exports), the encoding of n-ary signatures is exactly the same as for n-ary structures:

$$\{\ell_1 \triangleright \mathrm{X}_1 : sig_1, \ldots\} \stackrel{\text{def}}{=} (\mathrm{X} = \{\ell_1 = sig_1\})\ \texttt{with}\ \{\ldots\}[\mathrm{X}.\ell_1/\mathrm{X}_1]$$

We just *change the colons to equal signs*: wherever you see $\mathrm{X} : sig$ in ML code, expect to see $\mathrm{X} = sig$ in its MixML encoding. As we will show, this maxim applies to all instances of structure specification in ML, not just substructure specifications.

Type Signatures. Since n-ary structures and signatures expand to the same construct, definitions and specifications can be mixed freely. One benefit of this is the possibility to provide separate type signatures for the definitions in a structure (a feature that is frequently requested for ML). For example:

$$\begin{array}{l}\{\texttt{val f : int} \to \texttt{int},\\ \ \texttt{val f =}\ \lambda x.\, \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * \texttt{f}\,(x-1),\ \ldots\}\end{array}$$

The first binding only declares the type of `f`, *i.e.,* provides a type signature for it, while the second gives the actual definition. The first is an import and the second an export, so they will be linked and appear as one field to the remainder of the program (provided the type of the definition matches the declared type).

Note that, according to the expansions we have defined above, structures and their fields are not, by themselves, recursive. Consequently, in the example, the occurrence of `f` inside the function *definition* refers to the binding provided by the previous *specification* (which then get merged via recursive linking). Its type hence is determined by the explicit type signature. Thus, if the core language provides ML-style polymorphism, explicit type signatures can be used to directly introduce polymorphic recursion, like in Haskell [Peyton Jones et al. 2003]. (Unlike in Haskell, however, the ex-

ternal type of `f` will be the—potentially more general—type derived for the second binding, *i.e.,* the type signature does not restrict the final type.)

Due to the general nature of linking, there is no reason that the type signature and the definition of a value have to be given as consecutive bindings—a type signature can also act as a *forward declaration* for a definition occurring much later in the same structure.

Local Module Definitions. Using the above encoding of n-ary namespaces, we can easily encode a `let` construct that enables the definition of *local* modules. The encoding makes use of two labels, ℓ_1 and ℓ_2, which are arbitrary:

$$\texttt{let X} = mod_1 \texttt{ in } mod_2 \stackrel{\text{def}}{=} \{\ell_1 \triangleright \mathrm{X} = mod_1, \ell_2 = mod_2\}.\ell_2$$

The two modules mod_1 and mod_2 are combined through hierarchical composition into a pair module, from which the second component ℓ_2 is then projected out. Since this has the effect of hiding mod_1, the MixML type system will insist that mod_1 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:

$$\texttt{complete } mod \stackrel{\text{def}}{=} \texttt{let X} = mod \texttt{ in X}$$

Signature Inheritance. In ML, one may define a signature that inherits specifications from an existing signature sig and adds new specifications to it. This is supported by the `include` mechanism:

$$\{\texttt{include } sig;\ newspecs\}$$

MixML supports signature inheritance through linking. To add $newspecs$ to sig, we can write

$$(\mathrm{X} = sig) \texttt{ with } \{newspecs\}$$

(Note that in our encoding, in order for $newspecs$ to refer to the components specified in sig, it must project them from X.)

In fact, linking is more flexible than `include` because `include` does not permit multiple inheritance from overlapping signatures. For instance, if sig_1 and sig_2 both contain specifications of a type component (named `t` in both signatures), together with several operations over values of that type, it is prohibited in ML to write

$$\{\texttt{include } sig_1;\ \texttt{include } sig_2\}$$

due to the overlapping `t` specs. In MixML, though, we can write

$$sig_1 \texttt{ with } sig_2$$

and mixin composition will permit overlapping specs in sig_1 and sig_2 so long as they are compatible (which in this case they are). A similar approach to multiple signature inheritance is offered by Ramsey et al.'s `andalso` signature combinator [Ramsey et al. 2005], but in our case the added functionality falls out directly from the semantics of mixin linking.

ML also provides an `include` mechanism for *structure* inheritance.[2] If that `include` were a non-shadowing operation like signature `include`, the encoding of `include` would double as an encoding of `include` for structures (replacing the sig's above with mod's). However, unlike signature inclusion, structure inclusion is permitted to (1) shadow

[2]In Standard ML, it is called `open`.

earlier bindings, and (2) be shadowed by later bindings. As mentioned earlier, the former is easy to handle using known techniques [Harper and Stone 2000; Rossberg et al. 2010]. The latter, however, cannot be expressed concisely in MixML, where we can only rewrite $\{\texttt{include}\ \ mod\texttt{;}\ \ newdecs\}$ to $\texttt{let X} = mod\ \texttt{in}\ \{\ell_1 = \mathrm{X}.\ell_1, \ldots, \ell_n = \mathrm{X}.\ell_n, newdecs\}$, enumerating all labels ℓ_i from mod that are not shadowed by $newdecs$.

Signature Refinement. Mixin linking can also be used to define a very simple encoding of ML's `with type` (or `where type`) mechanism for adding type definitions to signatures. The ML construct

$$sig\ \texttt{with type t}\ \overline{\alpha}\ \texttt{=}\ typ$$

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

$$sig\ \texttt{with (type t}\ \overline{\alpha}\ \texttt{=}\ typ\texttt{)}$$

where "$\texttt{type t}\ \overline{\alpha}\ \texttt{=}\ typ$" is encoded as defined earlier. Mixin linking will use the definition for `t` on the right side of the `with` (*i.e.,* $\lambda\overline{\alpha}.typ$) to fill in the abstract specification for `t` in sig. It is also easy to encode the more general form of `with type` in which `t` can be a path $\ell_1.\cdots.\ell_n$, using the following inductive definition:

$$\texttt{type}\ \ell_1.\ell s\ \overline{\alpha}\ \texttt{=}\ typ \stackrel{\text{def}}{=} \{\ell_1 = (\texttt{type}\ \ell s\ \overline{\alpha}\ \texttt{=}\ typ)\}$$

In fact, though, the above encoding is not entirely faithful to the original ML semantics of `with type`: if the type component `t` does not appear in sig at all, then the encoding will not report a type error (as ML semantics would dictate it should), but rather simply add "$\texttt{type t}\ \overline{\alpha}\ \texttt{=}\ typ$" to sig. If we want to match ML semantics more precisely, we need to first check that sig contains a specification for the type `t`.

Enforcing Signature Matching. Such a check can be achieved by replacing sig in the above encoding with $sig\ \texttt{matches (type t}\ \overline{\alpha}\texttt{)}$, where the `matches` mechanism is defined as follows:

$$sig_1\ \texttt{matches}\ sig_2 \stackrel{\text{def}}{=} \{\ell_1 \triangleright \mathrm{X} = sig_1, \ell_2 = \mathrm{X}\ \texttt{with}\ sig_2\}.\ell_1$$

This expression enforces that sig_1 *matches* the signature sig_2. More precisely, the projection of ℓ_1 here means that: (1) if the encoding is well-typed, then $sig_1\ \texttt{matches}\ sig_2$ is indistinguishable from sig_1, and (2) the encoding will only be well-typed if the hidden module labeled ℓ_2 is complete (*i.e.,* has no imports). This second condition implies that the imports of sig_2 (its value and abstract type specs) must all be provided by X (as either imports or exports)—*i.e.,* X's signature sig_1 must actually match sig_2.

Type Sharing Constraints. If sig contains two abstract type components `u` and `t`, and we wish to refine the signature so that `t` is transparently equal to `u`, the traditional ML `with type` construct does not permit us to do so because `u` is not a valid type outside the signature. Standard ML retains a second signature refinement operator, `sharing type`, precisely to make up for this deficiency.

In MixML, we can encode `sharing type` very easily by exploiting the ability to bind sig to a variable while we refine it. That is, in order to refine sig so that `t` equals `u`, we can write

$$(\mathrm{X} = sig)\ \texttt{with type t = X.u}$$

We use X here to provide `t`'s definition with a way of referring to the `u` component from sig. This is similar to a proposal of Ramsey et al. [2005], but in our case the added functionality again falls out directly from our unification of linking and binding.

Recursive Structures. Another feature for which the unification of linking and binding facilitates a very simple encoding is the recursive structure definition. Recursive structure extensions to ML typically have the form:

$$\texttt{rec}\,(\mathrm{X} : sig)\; mod$$

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 type-checking of mod.

The encoding of this construct in MixML is extremely simple:

$$(\mathrm{X} = sig)\ \texttt{with}\ mod$$

Mixin linking is in fact a generalization of traditional recursive structure definitions! It 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. The binding of X inside mod gives mod a way of referring to its own components (at least those specified in sig) recursively. Note also that this is another instance of our rule: Just *change the colons to equal signs*—$\mathrm{X} : sig$ becomes $\mathrm{X} = sig$.

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 could use the `complete` combinator, 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* [Crary et al. 1999]. In Russo's extension to Moscow ML [Russo 2001], it takes the form:

$$\texttt{rec}\,(\mathrm{X} : sig_1)\; sig_2$$

This construct allows the signatures of mutually recursive modules (in sig_2) to refer recursively to each other's type components through the variable X. Of course, since structures and signatures are both encoded in MixML as modules, this construct is encoded in the exact same way as the recursive structure construct:

$$(\mathrm{X} = sig_1)\ \texttt{with}\ sig_2$$

One point of note is that not all recursive module extensions to ML require the programmer to write down sig_1. Instead, they infer it from sig_2. We view such an inference step as a separable convenience. In any case, this encoding demonstrates that recursive structures and recursively dependent signatures can be understood as one and the same feature.

Opaque Signature Ascription as 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—$(\mathrm{X} = mod_1)$ `seals` mod_2—which we call *opaque linking* (as opposed to the original form, which we view as *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 mod_1—no information about mod_2 may be revealed.

This informal property implies several things. First, mod_2 must define *all* of mod_1's imports. If it only defined some of them, we would have no way of knowing which ones

it defined without looking at it (and thus violating data abstraction). Second, the type *imports* of mod_1 will become type *exports* of the linked module; furthermore, these will be *abstract type exports*, meaning that the type-checker will ensure that their identity is not revealed outside of the linked module. Third, the exports of the linked module are limited to those components either specified (imports) or defined (exports) in mod_1. Thus, since all of mod_1's imports are fulfilled by mod_2, the result of opaque linking is always a complete module.

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

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

Functors as 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 10, our units are inspired by Flatt et al.'s units [Flatt and Felleisen 1998; Owens and Flatt 2006], but are different in many respects.)

A unit, written $\texttt{[}mod\texttt{]}$, 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:

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

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. It is probably no coincidence that this approach is very similar to Abadi and Cardelli's encoding of functions in their object calculus [Abadi and Cardelli 1996]. (Note how even here the argument binding $\mathrm{X} : sig$ is encoded as $\mathrm{X} = sig$.)

Unlike ML functors, though, units need not be so rigidly structured. Any module can be suspended into a unit, and this added generality is useful, as we will see shortly.

The elimination construct for units is written $\texttt{new } mod$. Here, mod is assumed to be a unit, and $\texttt{new } mod$ has the effect of *instantiating* that unit by producing a fresh copy of its constituent module, which can then be linked with other modules that satisfy its imports. 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:

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

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

Every instantiation of a unit F generates a distinct instance of the module expression contained within F. In particular, each occurrence of $\texttt{new F}$ 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 [Leroy 1995], which we leave to future work.

Transparent Signature Ascription. In addition to opaque signature ascription, Standard ML includes a mechanism for transparent signature ascription, written $mod : sig$, which narrows the exports of mod to those specified in sig but does not perform any type abstraction. It is well-known that transparent ascription with signature sig can be encoded as an application of the identity functor at sig (cf. Rossberg et al. [2010]):

$$mod : sig \quad \stackrel{\text{def}}{=} \quad (\lambda(\mathrm{X}{:}sig).\mathrm{X})(mod)$$

```
TREE = λ(forest : [:type]).
       [ { t    = [:type]
         , leaf = [:int → t]
         , fork = [:forest → t]
         , push = [:t → t → t]
         , has  = [:int → t → bool] } ]

FOREST = λ(tree : [:type]).
         [ { t     = [:type]
           , empty = [:t]
           , add   = [:tree → t → t]
           , has   = [:int → t → bool] } ]

TREE_FOREST = [ rec (X : {Forest = {t = [:type]}})
              [ { Tree   = new TREE(X.Forest.t)
                , Forest = new FOREST(Tree.t) } ] ]

TreeForest = rec (X : new TREE_FOREST)
             { Tree = new TREE(X.Forest.t) seals
                      { t    = [Leaf of int | Fork of X.Forest.t]
                      , leaf = [Leaf]
                      , fork = [Fork]
                      , push = [ λ(x:t). λ(y:t). case y of
                                 | Leaf i ⇒ Fork (X.Forest.add x
                                                 (X.Forest.add y
                                                    X.Forest.empty))
                                 | Fork f ⇒ Fork (X.Forest.add x f) ]
                      , has  = [ λ(i:int). λ(x:t). case x of
                                 | Leaf j ⇒ i = j
                                 | Fork f ⇒ X.Forest.has i f ] }
             , Forest = new FOREST(X.Tree.t) seals
                        { t     = [list X.Tree.t]
                        , empty = [List.nil]
                        , add   = [List.cons]
                        , has   = [λi. λf. exists (X.Tree.has i) f] } }
```

Fig. 2. Example: Mutually Recursive Modules

Parameterized Signatures. Jones [1996] proposed the idea of parameterized signatures, *i.e.,* signatures parameterized over module arguments. Although it has been argued that ordinary ML signatures subsume the expressiveness of parameterized signatures, we merely wish to point out here that parameterized signatures are directly encodable in MixML via the *exact same encoding* as for functors—functors and parameterized signatures are one and the same thing.

Signature Bindings. In addition to modeling parameterized structures and signatures, 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). It would be incorrect to bind S to

$$\begin{array}{lcl}
\texttt{MkTree} & = & \begin{bmatrix}(\texttt{X} = \texttt{new TREE_FOREST}) \texttt{ with} \\ \{\texttt{Tree} = \ldots \texttt{(* as before *)} \ldots\}\end{bmatrix} \\
\texttt{MkForest} & = & \begin{bmatrix}(\texttt{X} = \texttt{new TREE_FOREST}) \texttt{ with} \\ \{\texttt{Forest} = \ldots \texttt{(* as before *)} \ldots\}\end{bmatrix} \\
\texttt{TreeForest} & = & \texttt{new MkTree with new MkForest}
\end{array}$$

Fig. 3. Example: Separate Compilation of Mutually Recursive Modules

sig directly: $\mathrm{S} = sig$ is the MixML encoding of the ML *structure* specification $\mathrm{S} : sig$, so if S were defined that way, references to it would be references to a particular structure of signature sig (and to particular, yet undefined, instances of the abstract types specificed in sig). More concretely, assume we want to bind the signature $\{\texttt{t} = \texttt{[:type]}, \ldots\}$ as S and use it for transparent ascription. However, if we simply wrote

$$\left\{ \begin{array}{l} \mathrm{S} = \{\texttt{t} = \texttt{[:type]}, \ldots\}, \\ \texttt{A} = \{\texttt{t} = \texttt{[bool]}, \ldots\} : \mathrm{S}, \\ \texttt{B} = \{\texttt{t} = \texttt{[int]}, \ldots\} : \mathrm{S} \end{array} \right\}$$

then S would just be a name for *one* given structure with a yet undefined type component `t`. The $mod : \mathrm{S}$ operator links this structure against mod. In the above example, we hence try to link S against two different structures, with two different implementations of type `t`, therefore defining `S.t` twice with incompatible types. Clearly, that cannot type-check.

In ML, a signature binding implicitly quantifies over all abstract types declared in the signature. Inversely, when the signature name is used, the quantifiers are instantiated with "fresh" types. In MixML, we can simulate this with units. Hence, in order to define S to *be* the *signature* sig (instead of just a *structure* shaped like sig), we bind S to the *unit* $[sig]$ and replace all subsequent uses of S with `new` S. This works because each reference to `new` S will produce a fresh copy of sig, whose imports (including abstract types) may then be instantiated via linking independently:

$$\left\{ \begin{array}{l} \mathrm{S} = [\{\texttt{t} = \texttt{[:type]}, \ldots\}], \\ \texttt{A} = \{\texttt{t} = \texttt{[bool]}, \ldots\} : \texttt{new } \mathrm{S}, \\ \texttt{B} = \{\texttt{t} = \texttt{[int]}, \ldots\} : \texttt{new } \mathrm{S} \end{array} \right\}$$

Since ML signatures do not contain term definitions, performing `new` on a signature variable will never have any computational effects.

Separate Compilation of Recursive Modules. At the start of the article, 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 $sig_{\texttt{A}}$ and $sig_{\texttt{B}}$, and definitions $mod_{\texttt{A}}$ and $mod_{\texttt{B}}$, which refer recursively to themselves and to each other through the module variable X. Let the signature variable S be bound to the unit

$$[(\mathrm{X} = \ldots) \texttt{ with } \{\texttt{A} = sig_{\texttt{A}}, \texttt{B} = sig_{\texttt{B}}\}]$$

where $\ldots$ is a signature specifying the type components of `A` and `B` that $sig_{\texttt{A}}$ and $sig_{\texttt{B}}$ need to refer to recursively. Were we to write `A`'s and `B`'s definitions together, the MixML code would be:

$$(\mathrm{X} = \texttt{new } \mathrm{S}) \texttt{ with } \{\texttt{A} = mod_{\texttt{A}}, \texttt{B} = mod_{\texttt{B}}\}$$

But there is no need to define A and B together. We can, separately, bind $\mathtt{U_A}$ to

$$[(\mathrm{X} = \mathtt{new}\ \mathrm{S})\ \mathtt{with}\ \{\mathtt{A} = \mathit{mod}_{\mathtt{A}}\}]$$

and $\mathtt{U_B}$ to

$$[(\mathrm{X} = \mathtt{new}\ \mathrm{S})\ \mathtt{with}\ \{\mathtt{B} = \mathit{mod}_{\mathtt{B}}\}]$$

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

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

Of course, there is nothing requiring us to link $\mathtt{U_A}$ and $\mathtt{U_B}$ in this order or with each other. They are completely independent program units that can link with any other compatible units.

Figure 3 shows a more concrete example of two mutually recursive modules defining abstract data types for (unordered) trees and forests (cf. Nakata and Garrigue [2006] for a related example, but without the use of type abstraction). It first defines the signatures TREE and FOREST for those two modules. In order to factor out the recursion between them, we have chosen here to define them as parameterized signatures. (However, we could as well have made their parameters into *components* of the signatures, which is the technique often used in ML.)

We then define a third, recursive signature TREE_FOREST that describes a single structure containing the Tree and Forest modules, tying the knot between the two individual signatures. It is needed as a forward declaration for the actual definition of those modules in the structure TreeForest. Each module is sealed, independent of the other one, with its respective signature. Consequently, the implementation of neither abstract data type can see internal details of the other.

Figure 3 then demonstrates how TreeForest can be split into two separate units. Each is defined in recursion with the *combined* signature TREE_FOREST, but defines only half of it. Subsequently, the two units can be instantiated and linked together.

The recursive nature of units is instrumental to making this decomposition work. Suppose we were to try expressing, say, MkTree as a plain *functor*:

$$\lambda(\mathtt{X} : \mathtt{new}\ \mathtt{TREE\_FOREST}).\{\mathtt{Tree} = \ldots\ \mathtt{(*\ as\ before\ *)}\ \ldots\}$$

Then the type X.Tree.t would be fully abstract inside the functor body, and type-checking the push function would fail at the call to X.Forest.add, which is in scope with abstract type $\mathtt{X.Tree.t} \to \mathtt{X.Forest.t} \to \mathtt{X.Forest.t}$, but which is passed y with concrete internal type $\mathtt{t} = \mathtt{Leaf\ of\ int} \mid \mathtt{Fork\ of\ X.Forest.t}$. There is no way to recover the necessary type equivalence, because functors simply do not have the ability to express that, in fact, X.Tree.t is supposed to be the same type as the type Tree.t *returned* by the functor. Compare this to the unit definition we use:

$$[(\mathtt{X} = \mathtt{new}\ \mathtt{TREE\_FOREST})\ \mathtt{with}\ \{\mathtt{Tree} = \ldots\ \mathtt{(*\ as\ before\ *)}\ \ldots\}]$$

Here, it is readily apparent to the type system that X.Tree.t and Tree.t end up being the same type (for more details, see the discussion of *double vision* in the next section).

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 (functors that take functors as arguments—a feature in many dialects of ML), then we must also allow unit *imports*. In order to encode unit imports, it is necessary to extend MixML with a notion of *unit signature*, analogous to a *functor signature* in higher-order module extensions to ML.

Units as First-Class Values. Like in most existing dialects of ML, we have so far assumed that the module language is separate from the core language: the constructs described provide no means for passing modules to, or returning them from, core-level functions, or for storing them in data structures. However, it is sometimes useful to pick or compute modules at run-time. This functionality can be provided by extending the language with the ability to package up units as first-class values.

As the latter two extensions are completely orthogonal to the other features of MixML, and since we feel the rest of the MixML type system is perhaps easier to understand in the absence of these extensions, we will continue in the next two sections by presenting the basic MixML type system, and then present higher-order units and units as first-class values as optional extensions in Sections 5 and 6.

3. CHALLENGES IN TYPING MIXML

The combination of recursive mixin composition with abstract type components and sealing raises a number of technical challenges. In this section we informally discuss the central problems that arise in typing MixML. The solutions we employ are mostly generalizations of the techniques developed by Dreyer for typing recursive modules [Dreyer 2007b].

Bidirectional Type Lookup. In ML, matching a structure mod against a signature sig is a two-step procedure. First, for each type component specified abstractly in sig, we look up its definition in mod, and refine its specification in sig appropriately (*i.e.,* make its specification transparently equal to its definition in mod). We then check that the specification of each (type or value) component in the refined sig is matched by its corresponding definition in mod.

To see why this two-step process is necessary, consider:

```
struct type t = int; val x = 3 end
                  :>
    sig type t; val x : t end
```

Checking whether 3 has type `t` will fail unless we first refine the specification of `t` to its underlying definition, `type t = int`.

In MixML, we no longer explicitly distinguish structures from signatures. Linking effectively generalizes unidirectional matching to bidirectional *merging* of two modules, which may both contain abstract type components. Consequently, type lookup and refinement must be performed in both directions simultaneously. For example, consider:

$$\left\{\begin{array}{l}\texttt{t} = \texttt{[int]},\\ \texttt{u} = \texttt{[:type]},\\ \texttt{f} = \texttt{[:int} \to \texttt{u]}\end{array}\right\} \texttt{ with } \left\{\begin{array}{l}\texttt{t} = \texttt{[:type]},\\ \texttt{u} = \texttt{[bool]},\\ \texttt{f} = \texttt{[}\lambda x\texttt{:t.true]}\end{array}\right\}$$

The definition for `f` in the second module will only match its specification in the first module if we first refine `u` to `bool` and `t` to `int` in both modules. This involves *bidirectional type lookup*, which, as we will see, is a straightforward generalization of ML's unidirectional type lookup.

Cyclic Type Definitions. Type lookup in MixML can easily introduce cyclic type definitions. For example,

$$(\texttt{X} = \{\texttt{t} = \texttt{[:type]}\})\ \texttt{with}\ \{\texttt{t} = \texttt{[X.t} \to \texttt{int]}\}$$

or

$$\left\{\begin{array}{l}\texttt{t} = \texttt{[:type]},\\ \texttt{u} = \texttt{[t]}\end{array}\right\} \texttt{ with } \left\{\begin{array}{l}\texttt{u} = \texttt{[:type]},\\ \texttt{t} = \texttt{[u]}\end{array}\right\}$$

Supporting such definitions would require the introduction of higher-kinded *equi-recursive* types [Crary et al. 1999] into the type system, for which there is no known effective type-checking algorithm in the general case. Hence, during type lookup, we check that the definitions of the abstract type components being looked up do not have cyclic dependencies. In particular, we would reject both of the above examples.

The prohibition on type cycles during lookup prevents one from defining *transparently* recursive types. However, as we explain at the end of this section, we do allow the definition of *opaquely* recursive types, which generalize ML-style iso-recursive datatypes.

Cyclic Term Definitions. Linking can also introduce cycles between the definitions of term components. We adopt a sequential call-by-value semantics for the evaluation of term components in MixML, where recursion is implemented by `letrec`-style backpatching (Section 8). This is similar to the approach taken by several other recursive module systems [Flatt and Felleisen 1998; Russo 2001; Leroy 2003; Nakata and Garrigue 2006; Dreyer 2007b].

Under the backpatching semantics, cyclic linking can cause a run time exception if a term component is accessed before its definition has been evaluated. Static detection of such errors is a problem that is orthogonal to our work and has been addressed by Hirschowitz and Leroy [2005] and Dreyer [2004] among others. (In general, if separate compilation is still desired, static checking for illegal cycles is only possible if the programmer provides sufficient dependency annotations on imports. We have chosen not to open that particular can of worms for MixML.)

Double Vision. An important problem that arises in extending ML with recursive modules is the *double vision problem* [Dreyer 2005]. Consider the following simple example:

$$(\texttt{X} = \{\texttt{t} = \texttt{[:type]}, \ldots\})\ \texttt{seals}\ \{\texttt{t} = \texttt{[int]}, \ldots\}$$

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 alias for `t`) equals `int` as well, but the signature bound to `X` does not reflect this. As a result, the programmer may be forced to expose the definition of `t` as `int` in the l.h.s. module, thus losing type abstraction.

For this particular example, the problem can be worked around by making `t` transparently equal to `int` in the l.h.s. module, and then applying sealing "after the fact." However, it is not always possible to seal after the fact. For instance, if a recursive module contains sealed substructures that wish to hide type information from one another, then there is no way to hoist out the sealing without exposing the substructures' implementations to each other.

Fortunately, Dreyer has developed a general solution to the double vision problem in his RMC type system [Dreyer 2007b], and we can readily adopt his solution. The central ideas of RMC are as follows.

First is the idea of *forward-declaring* abstract types. In RMC, the typing judgment for a module mod assumes that the names of mod's abstract types have already been forward-declared (*i.e.,* created ahead of time, potentially in an earlier scope), and that they will be passed in as input to the typing judgment. For example, in typing the sealed recursive module above, RMC would assume that in the context there already exists a type variable, say α, which was forward-declared to represent the abstract type component `t`.

Secondly, when type-checking a recursive or sealed module, RMC employs a *two-pass* algorithm. The first pass is a "static" pass, which computes the type components

of the module (*e.g.,* discovering that `t` in our example is defined internally as `int`). The information from the static pass is then incorporated into the typing context during the second "main" pass, which fully type-checks the module. In our example, this would mean that the body of the sealed module will be fully type-checked in a context where (1) `X.t` is transparently equal to `int`, and (2) any occurrence of α (the forward-declared representative of `t`) in the typing context is replaced by `int`. This approach successfully avoids double vision by ensuring that all forward references to the abstract type `t` in the typing context are "up-to-date" with the most precise information available about `t` in the current scope.

We adopt the same cure for double vision in MixML, forward-declaring abstract types and performing two passes on the r.h.s. of every linking operation. (Note that, if we had used a symmetric linking construct, this cure would not work. If both sides of the linking construct could refer to each other, then the static passes for both sides would become mutually dependent.) Unfortunately, since linking involves bidirectional merging instead of unidirectional matching, the RMC solution *per se* is not quite enough.

Cross-Eyed Double Vision. Consider the following example:

$$\left(\texttt{X} = \begin{Bmatrix} \texttt{t} \triangleright \texttt{t}_1 \;=\; \texttt{[:type]}, \\ \texttt{u} \triangleright \texttt{u}_1 \;=\; \texttt{[int]}, \\ \texttt{f} \;=\; \texttt{[}\lambda\texttt{x}\texttt{:t}_1\texttt{.\ x]} \end{Bmatrix}\right) \texttt{ with } \begin{Bmatrix} \texttt{t} \triangleright \texttt{t}_2 \;=\; \texttt{[bool]}, \\ \texttt{u} \triangleright \texttt{u}_2 \;=\; \texttt{[:type]}, \\ \texttt{g} \;=\; \texttt{[}\lambda\texttt{y}\texttt{:u}_2\texttt{.\ X.f(y > 1)]} \end{Bmatrix}$$

Inside the definition of `g`, both `t` and `u` are accessible under two distinct paths ($\texttt{t}_2$ vs. `X.t` and $\texttt{u}_2$ vs. `X.u`, respectively). Thus, to type-check the definition, two instances of double vision have to be handled: checking `y > 1` requires knowing that $\texttt{u}_2 = \texttt{X.u}$ (and thus $\texttt{u}_2 = \texttt{int}$), and checking the application of `X.f` to the resulting boolean requires knowing that $\texttt{X.t} = \texttt{t}_2$ (and thus $\texttt{X.t} = \texttt{bool}$).

In short, this example suffers from *cross-eyed double vision*. The RMC solution takes care of one direction ($\texttt{X.t} = \texttt{bool}$) but not the other. In order to inform the type-checker that $\texttt{u}_2 = \texttt{int}$, we generalize the RMC approach as follows. In addition to taking as input a list of type variables corresponding to the abstract export types of a module, the MixML module typing judgment takes as input an *import realizer*, which maps the type imports of the module to the concrete types that will instantiate them. In the above example, when performing the main pass on the r.h.s. module, the type-checker will pass in a realizer mapping `u` to `int`, and this information will get propagated to the r.h.s. definition of `u`. In order to compute this realizer, we perform bidirectional type lookup in between the static and main passes of type-checking.

Information about import types is only propagated rightward. For instance, in the above example, the l.h.s. module does not get to know that $\texttt{t}_1$ = `bool`. This does not incur double vision because the l.h.s. module does not have a name by which to refer to the r.h.s. module. If a module's type imports are not instantiated by (the l.h.s. of) any enclosing linking operation, the type-checker will pass in a realizer that maps them to abstract type variables. For the details of how those import type variables are managed, see Section 4.

While double vision is a problem with no easy workarounds, the seriousness of cross-eyed double vision is somewhat debatable. For instance, in our example, we could easily avoid double vision for the `u` component by making its definition in the r.h.s. module manifestly equal to `X.u`. However, such a workaround is quite brittle. It only works if `u` is an export in the l.h.s. module; otherwise, the definition of $\texttt{u}_2$ as `X.u` is indistinguishable from a transparent type cycle. It is simpler for the programmer to be able to rely on the type system to avoid double vision in both directions.

Opaquely Recursive Types. We conclude this section by briefly explaining how to encode ML-style iso-recursive (or "opaquely recursive") `datatype`'s. The encoding is interesting because it demonstrates an instance where we *want* to incur double vision! Luckily, the MixML type system provides a very simple way of manually overriding the built-in solution to double vision.

First, consider the following encodings of specifications and definitions for *non-recursive* `datatype`'s, respectively:

$$\{:\ell \approx typ\} \stackrel{\text{def}}{=} \{\ell \triangleright \mathrm{X} = [:\mathtt{type}], \ell\_\mathtt{in} = [:typ \to \mathrm{X}], \ell\_\mathtt{out} = [:\mathrm{X} \to typ]\}$$
$$\{\ell \approx typ\} \stackrel{\text{def}}{=} \{:\ell \approx typ\}\ \mathtt{seals}\ \{\ell \triangleright \mathrm{X} = [typ], \ell\_\mathtt{in} = [\lambda x{:}\mathrm{X}.x], \ell\_\mathtt{out} = [\lambda x{:}\mathrm{X}.x]\}$$

Similar to the interpretations given by Harper and Stone [2000] and Dreyer [2007b], these encodings model the `datatype` definition $\{\ell \approx typ\}$ as an ADT providing an abstract type ℓ, together with ℓ`_in` (fold) and ℓ`_out` (unfold) functions to coerce between ℓ and its underlying representation typ. For brevity, we have only shown the encoding of monomorphic `datatype`'s (of kind `type`) here; it can be easily generalized to the polymorphic, higher-kinded case.

Given these definitions, it would seem straightforward to encode a recursive `datatype` by enclosing a non-recursive `datatype` in a recursive module. For example, integer lists:

$$\mathtt{rec}\,(\mathrm{X}:\{\mathtt{intlist} = [:\mathtt{type}]\})\ \{\mathtt{intlist} \approx \mathtt{unit} + \mathtt{int} \times \mathrm{X}.\mathtt{intlist}\}$$

Unfortunately, this encoding does not type-check. The reason is that, when the type-checker descends into the body of the sealed `datatype` module, it will (1) discover that `intlist` is defined to be $\tau = \mathtt{unit} + \mathtt{int} \times \mathrm{X}.\mathtt{intlist}$, (2) try to update the typing context so that `X.intlist` is transparently equal to τ, and (3) report the presence of a transparent type cycle.

What we want, then, is to be able to switch off the type-checker's double vision avoidance mechanism. We can achieve this by inserting a computationally irrelevant *unit* suspension/instantiation β-redex (underlined here):

$$\mathtt{rec}\,(\mathrm{X}:\{\mathtt{intlist} = [:\mathtt{type}]\})\ \underline{\mathtt{new}[\{\mathtt{intlist} \approx \mathtt{unit} + \mathtt{int} \times \mathrm{X}.\mathtt{intlist}\}]}$$

Inserting "`new[·]`" has the effect of dislocating the `datatype` module from its surrounding scope. Units in MixML were designed for encapsulation and separate compilation, and thus the MixML type-checker does not make any attempt to connect the abstract types defined inside a unit (in this case, `intlist`) with any forward-declared types in the typing context (`X.intlist`). One could make the case for an alternative design in which the type-checker does attempt to connect the types, but the benefit of being able to switch off double vision avoidance is evident from the above encoding.

4. THE MIXML TYPE SYSTEM

4.1. Semantic Objects

The MixML type system is based to a large extent on Dreyer's RMC type system for recursive modules [Dreyer 2007b]. RMC in turn inherits many aspects from the Definition of Standard ML [Milner et al. 1997]. In particular, it represents the types of modules by *semantic objects*. As in RMC, our semantic objects—shown in Figure 4—are actually types from a simpler "internal" type system (which will be defined in Section 7), enriched with annotations that guide type-checking.

Semantic signatures (Σ) include structure signatures ($\{\!|\overline{\ell : \Sigma}|\!\}$), as well as atomic signatures for type modules ($[\![= \mathrm{A}]\!]$), term modules ($[\![\mathrm{A}]\!]^{\pm}$, where $\pm$ stands for either $+$ or $-$), and units ($[\![\Phi]\!]^{+}$). Our semantic objects differ from RMC's in that atomic term and

$$
\begin{array}{lrl}
\textbf{Type Constructors} & \mathrm{A} & ::= \alpha \mid \lambda\alpha.\mathrm{A} \mid \mathrm{A}_1\ \mathrm{A}_2 \mid \mathrm{A}_1 \to \mathrm{A}_2 \mid \ldots \\
\textbf{Module Signatures} & \Sigma & ::= [\![= \mathrm{A}]\!] \mid [\![\mathrm{A}]\!]^{\pm} \mid [\![\Phi]\!]^{+} \mid \{\!|\overline{\ell : \Sigma}|\!\} \\
\textbf{Unit Signatures} & \Phi & ::= \forall\overline{\alpha}.\, \exists\overline{\beta}.\, (\mathcal{L}; \Sigma) \\
\textbf{Type Substitutions} & \delta & ::= \{\overline{\alpha \mapsto \mathrm{A}}\} \\
\textbf{Type Locators} & \mathcal{L} & ::= [\![= \alpha]\!] \mid \{\!|\overline{\ell : \mathcal{L}}|\!\} \\
\textbf{Import Realizers} & \mathcal{R} & ::= [\![= \mathrm{A}]\!] \mid \{\!|\overline{\ell : \mathcal{R}}|\!\} \\
\textbf{Module Contexts} & \Gamma & ::= \epsilon \mid \Gamma, \mathrm{X} : |\Sigma|
\end{array}
$$

$$
\begin{array}{rcl}
\Sigma.\ell s & \stackrel{\text{def}}{=} & \begin{cases} \Sigma & \text{if } \ell s = \epsilon \\ \Sigma' & \text{if } \ell s = \ell s'.\ell \text{ and } \Sigma.\ell s' = \{\!|\ell : \Sigma', \ldots|\!\} \end{cases} \\
\Sigma(\ell s) & \stackrel{\text{def}}{=} & \mathrm{A} \ \text{ if } \Sigma.\ell s = [\![= \mathrm{A}]\!] \\
\mathrm{dom}(\Sigma) & \stackrel{\text{def}}{=} & \{\ell s \mid \Sigma(\ell s) = \mathrm{A}\} \\
\mathrm{rng}(\mathcal{L}) & \stackrel{\text{def}}{=} & \{\alpha \mid \mathcal{L}(\ell s) = \alpha\} \\
\mathcal{L}^{-1}(\alpha) & \stackrel{\text{def}}{=} & \ell s \ \text{ if } \mathcal{L}(\ell s) = \alpha \text{ and } \nexists\, \ell s' \text{ such that } \mathcal{L}(\ell s') = \alpha \\
\mathcal{R} \subseteq \Sigma & \stackrel{\text{def}}{\Leftrightarrow} & \forall\, \ell s \in \mathrm{dom}(\mathcal{R}).\ \mathcal{R}(\ell s) = \Sigma(\ell s) \\
\mathcal{R} \,\#\, \Sigma & \stackrel{\text{def}}{\Leftrightarrow} & \mathrm{dom}(\mathcal{R}) \cap \mathrm{dom}(\Sigma) = \emptyset \\
\mathcal{R}_1 \uplus \mathcal{R}_2 & \stackrel{\text{def}}{=} & \mathcal{R} \text{ such that } \mathrm{dom}(\mathcal{R}) = \mathrm{dom}(\mathcal{R}_1) \uplus \mathrm{dom}(\mathcal{R}_2) \\
 & & \text{and } \forall \ell s \in \mathrm{dom}(\mathcal{R}).\ \mathcal{R}(\ell s) = \mathcal{R}_1(\ell s) \vee \mathcal{R}(\ell s) = \mathcal{R}_2(\ell s)
\end{array}
$$

$$
\begin{array}{rcl}
|[\![= \mathrm{A}]\!]| & \stackrel{\text{def}}{=} & [\![= \mathrm{A}]\!] \\
|[\![\mathrm{A}]\!]^{\pm}| & \stackrel{\text{def}}{=} & [\![\mathrm{A}]\!]^{+} \\
|[\![\Phi]\!]^{+}| & \stackrel{\text{def}}{=} & [\![\Phi]\!]^{+} \\
|\{\!|\overline{\ell : \Sigma}|\!\}| & \stackrel{\text{def}}{=} & \{\!|\overline{\ell : |\Sigma|}|\!\}
\end{array}
$$

Fig. 4. Semantic Objects and Auxiliary Definitions

unit signatures are annotated with variances, in order to denote whether they are imports ($-$) or exports ($+$).[3] The import/export distinction for type components is handled differently, as we explain below. Signatures in a module context Γ never have imports (a restriction that is enforced by the notation $|\Sigma|$, which turns all imports into exports).

A unit signature (Φ) is a module signature that has been universally quantified over the module's import types and existentially quantified over its abstract export types. (We write $\overline{\alpha}$ or $\overline{\beta}$ for an ordered sequence of type variables, but where appropriate, we also use the notation as shorthand for the sets $\{\overline{\alpha}\}$ or $\{\overline{\beta}\}$.) Unit signatures also contain a *type locator* $\mathcal{L}$ that maps the import names $\overline{\alpha}$ to label paths in Σ. Type locators are used to implement type lookup.

Locators are a syntactic subcategory of *import realizers* $\mathcal{R}$, described in Section 3, which are in turn a subcategory of module signatures Σ. This conveniently allows the sharing of meta-notation for all three kinds of objects, as shown in Figure 4. In particular, all three may be viewed as functions mapping the path names of type components to the type components themselves. Well-formed locators have the additional property that all their type components are distinct type variables, and thus they can be viewed as bijective functions between those type variables and their corresponding paths. As a matter of simplicity, we implicitly identify all realizers that represent the same map-

[3]Here we present only the fragment of MixML without unit imports $[\![\Phi]\!]^{-}$. Higher-order units are discussed in Section 5.

ping from paths to types, in effect ignoring syntactic differences with respect to empty substructures.

As a concrete example, consider the following unit:

$$\left[\begin{Bmatrix} \texttt{A} &=& \{\texttt{t} = \texttt{[:type]}\}, \\ \texttt{R} &=& \begin{Bmatrix}\texttt{u} = \texttt{[:type]}, \\ \texttt{f} = \texttt{[:A.t} \to \texttt{u]}\end{Bmatrix} \end{Bmatrix} \texttt{ seals } \begin{Bmatrix}\texttt{u} = \texttt{[int]}, \\ \texttt{f} = \texttt{[}\lambda\texttt{x:(A.t).7]}\end{Bmatrix}\right]$$

The following semantic signature describes this unit:

$$\forall\alpha.\, \exists\beta.\, (\mathcal{L}; \{\!|\texttt{A} : \{\!|\texttt{t} : [\![= \alpha]\!]|\!\}, \texttt{R} : \{\!|\texttt{u} : [\![= \beta]\!], \texttt{f} : [\![\alpha \to \beta]\!]^+|\!\}|\!\})$$

where $\mathcal{L}$ is the locator $\{\!|\texttt{A} : \{\!|\texttt{t} : [\![= \alpha]\!]|\!\}|\!\}$, mapping α to `A.t`. It imports the type `A.t`, represented by α, and exports the type `R.u`, represented by β. The exported function `R.f` is the only term component.

Assumptions about the core language. Semantic core-level types include standard type functions and application, plus an unspecified set of additional base types. We assume that there is a (decidable) judgment $\Gamma \vdash \mathit{typ} \rightsquigarrow \mathrm{A}$ that elaborates syntactic core language types into semantic core-level types. Likewise, there has to be a judgment $\Gamma \vdash \mathit{exp} : \mathrm{A}$ that type-checks a core language expression exp and assigns a semantic type. Finally, we need a subtyping judgment $\vdash \mathrm{A}_1 \leq \mathrm{A}_2$ of which we require that it defines a partial order on semantic core-level types, and is stable under substitution. We list additional assumptions regarding translation and algorithmic type-checking at the beginning of Sections 8 and 9, respectively.

Further assumptions. 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 $\vdash \mathrm{A} \Uparrow \mathit{knd}$ to assert that a constructor A is well-formed with kind knd. For type substitutions δ we demand implicitly that they be kind-preserving.

We also assume and maintain the invariant that types are kept in β-normal η-long form (this is relevant where we assume that types are syntactically equal, or when we take the set of free type variables of a type). We assume that substitutions are implicitly normalizing, *i.e.,* $\delta\mathrm{A}$ denotes the normal form of the application of δ to A. And we employ the convention that a type variable is synonymous with its η-long form, which allows us to treat locators as normalized signatures.

Finally, we assume the existence of a strict total ordering $<_{Paths}$ on label paths as a technical device for ensuring unique types.

4.2. Typing Rules

Figures 5 and 6 show the typing rules for MixML.

The main typing judgment for MixML modules has the form $\Gamma; \mathcal{R}; \overline{\beta} \vdash \mathit{mod} : \Sigma$. Here, $\overline{\beta}$ is a list of type variables representing mod's abstract type *exports*, while the realizer $\mathcal{R}$ captures mod's type *imports*. We implicitly require the variables in $\overline{\beta}$ to be distinct. Note that, due to forward declarations of mod's abstract types arising from linking, the variables $\overline{\beta}$ may also appear free in Γ.

As explained in Section 3, the use of realizers generalizes RMC's typing judgment. Another minor difference from RMC is that we choose to write $\overline{\beta}$ to the left of the turnstile instead of writing "with $\overline{\beta}\!\downarrow$" at the right end of the judgment. Just as in RMC, the MixML type system tracks type definitions linearly, ensuring that each β in $\overline{\beta}$ gets defined by mod exactly once. Thus, although $\overline{\beta}$ in the present judgment is treated like a linear list of capabilities (as opposed to the formulation in RMC, in which "with $\overline{\beta}\!\downarrow$" was treated as a *type effect*), the underlying semantics is morally the same, and we consider the difference to be cosmetic.

Modules: $\Gamma; \mathcal{R}; \overline{\beta} \vdash \mathit{mod} : \Sigma$

$$\frac{\mathrm{X} : |\Sigma| \in \Gamma}{\Gamma; \{\!|\}; \emptyset \vdash \mathrm{X} : |\Sigma|}\ (\textsf{VAR}) \qquad \frac{}{\Gamma; \{\!|\}; \emptyset \vdash \{\} : \{\!|\}}\ (\textsf{EMP})$$

$$\frac{\vdash \mathrm{A} \Uparrow \mathit{knd}}{\Gamma; [\![= \mathrm{A}]\!]; \emptyset \vdash \texttt{[:}\mathit{knd}\texttt{]} : [\![= \mathrm{A}]\!]}\ (\textsf{ITYP}) \qquad \frac{\Gamma \vdash \mathit{typ} \rightsquigarrow \mathrm{A}}{\Gamma; \{\!|\}; \emptyset \vdash \texttt{[}\mathit{typ}\texttt{]} : [\![= \mathrm{A}]\!]}\ (\textsf{ETYP})$$

$$\frac{\Gamma \vdash \mathit{typ} \rightsquigarrow \mathrm{A} \qquad \vdash \mathrm{A} \Uparrow \texttt{type}}{\Gamma; \{\!|\}; \emptyset \vdash \texttt{[:}\mathit{typ}\texttt{]} : [\![\mathrm{A}]\!]^-}\ (\textsf{IVAL}) \qquad \frac{\Gamma \vdash \mathit{exp} : \mathrm{A} \qquad \vdash \mathrm{A} \Uparrow \texttt{type}}{\Gamma; \{\!|\}; \emptyset \vdash \texttt{[}\mathit{exp}\texttt{]} : [\![\mathrm{A}]\!]^+}\ (\textsf{EVAL})$$

$$\frac{\Gamma; \mathcal{R}; \overline{\beta} \vdash \mathit{mod} : \Sigma}{\Gamma; \{\!|\ell : \mathcal{R}|\!\}; \overline{\beta} \vdash \{\ell = \mathit{mod}\} : \{\!|\ell : \Sigma|\!\}}\ (\textsf{STR}) \qquad \frac{\Gamma; \{\!|\ell : \mathcal{R}|\!\}; \overline{\beta} \vdash \mathit{mod} : \{\!|\ell : \Sigma, \overline{\ell' : |\Sigma'|}|\!\}}{\Gamma; \mathcal{R}; \overline{\beta} \vdash \mathit{mod}.\ell : \Sigma}\ (\textsf{DOT})$$

$$\frac{\begin{array}{rr} \vdash \mathcal{L}_1 \text{ locates } \overline{\alpha_1} & \mathcal{R}_1 \# \Sigma_2 \\ \vdash \mathcal{L}_2 \text{ locates } \overline{\alpha_2} & \mathcal{R}_2 \# \Sigma_1 \\ \vdash (\mathcal{L}_1; \Sigma_1) \rightleftarrows (\mathcal{L}_2; \Sigma_2') \rightsquigarrow \delta & \\ \overline{\alpha_1}, \overline{\alpha_2} \text{ fresh} & \end{array} \quad \begin{array}{r} \Gamma; \mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{L}_1; \overline{\beta_1} \vdash \mathit{mod}_1 : \Sigma_1 \\ \Gamma, \mathrm{X} : |\Sigma_1|; \mathcal{R} \uplus \mathcal{R}_2 \uplus \mathcal{L}_2; \overline{\beta_2} \vdash_{\mathrm{stat}} \mathit{mod}_2 : \Sigma_2' \\ \Gamma, \mathrm{X} : |\delta\Sigma_1|; \mathcal{R} \uplus \mathcal{R}_2 \uplus \delta\mathcal{L}_2; \overline{\beta_2} \vdash \mathit{mod}_2 : \Sigma_2 \\ \vdash \delta\Sigma_1 + \Sigma_2 \Rightarrow \Sigma \end{array}}{\Gamma; \mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{R}_2; \overline{\beta_1}, \overline{\beta_2} \vdash (\mathrm{X} = \mathit{mod}_1)\ \texttt{with}\ \mathit{mod}_2 : \Sigma}\ (\textsf{LINK})$$

$$\frac{\begin{array}{rr} \vdash \mathcal{L}_1 \text{ locates } \overline{\alpha_1} & \Gamma; \mathcal{L}_1; \overline{\beta_1} \vdash \mathit{mod}_1 : \Sigma_1 \\ \vdash \mathcal{L}_2 \text{ locates } \overline{\alpha_2} & \Gamma, \mathrm{X} : |\Sigma_1|; \mathcal{L}_2; \overline{\beta_2} \vdash_{\mathrm{stat}} \mathit{mod}_2 : \Sigma_2' \\ \vdash (\mathcal{L}_1; \Sigma_1) \rightleftarrows (\mathcal{L}_2; \Sigma_2') \rightsquigarrow \delta & \delta\Gamma, \mathrm{X} : |\delta\Sigma_1|; \delta\mathcal{L}_2; \overline{\beta_2} \vdash \mathit{mod}_2 : \Sigma_2 \\ \overline{\beta_2}, \overline{\alpha_2} \text{ fresh} & \vdash \delta\Sigma_1 + \Sigma_2 \Rightarrow |\Sigma| \end{array}}{\Gamma; \{\!|\}; \overline{\beta_1}, \overline{\alpha_1} \vdash (\mathrm{X} = \mathit{mod}_1)\ \texttt{seals}\ \mathit{mod}_2 : |\Sigma_1|}\ (\textsf{SEAL})$$

$$\frac{\Gamma \vdash \mathit{mod} : \Phi}{\Gamma; \{\!|\}; \emptyset \vdash \texttt{[}\mathit{mod}\texttt{]} : [\![\Phi]\!]^+}\ (\textsf{EUN})$$

$$\frac{\Gamma \vdash \mathit{mod} : [\![\forall \overline{\alpha}.\, \exists \overline{\beta}.\, (\mathcal{L}; \Sigma)]\!]^+ \qquad \mathrm{dom}(\delta) = \{\overline{\alpha}, \overline{\beta}\}}{\Gamma; \delta\mathcal{L}; \delta\overline{\beta} \vdash \texttt{new}\ \mathit{mod} : \delta\Sigma}\ (\textsf{NEW})$$

Complete Modules: $\Gamma \vdash \mathit{mod} : \Sigma$

$$\frac{\Gamma; \{\!|\}; \overline{\beta} \vdash \mathit{mod} : |\Sigma| \qquad \overline{\beta} \text{ fresh} \qquad \overline{\beta} \notin \mathrm{fv}(\Sigma)}{\Gamma \vdash \mathit{mod} : |\Sigma|}\ (\textsf{COMPL})$$

Units: $\Gamma \vdash \mathit{mod} : \Phi$

$$\frac{\Gamma; \mathcal{L}; \overline{\beta} \vdash \mathit{mod} : \Sigma \qquad \vdash \mathcal{L} \text{ locates } \overline{\alpha} \qquad \overline{\alpha}, \overline{\beta} \text{ fresh}}{\Gamma \vdash \mathit{mod} : \forall \overline{\alpha}.\, \exists \overline{\beta}.\, (\mathcal{L}; \Sigma)}\ (\textsf{UNIT})$$

Fig. 5. Typing Rules for MixML

Core-Language Types and Terms: $\Gamma \vdash typ \rightsquigarrow \mathrm{A} \quad \Gamma \vdash exp : \mathrm{A}$

$$\frac{\Gamma \vdash mod : [\![= \mathrm{A}]\!]}{\Gamma \vdash \mathtt{typ}(mod) \rightsquigarrow \mathrm{A}}\ (\textsf{PTYP}) \qquad \frac{\Gamma \vdash mod : [\![\mathrm{A}]\!]^+}{\Gamma \vdash \mathtt{val}(mod) : \mathrm{A}}\ (\textsf{PVAL})$$

Rules for α, $\lambda\alpha.typ$, $typ_1\ typ_2$ are standard.

Type Locators: $\vdash \mathcal{L}\ \mathsf{locates}\ \overline{\alpha}$

$$\frac{\overline{\alpha} = \alpha_1, \ldots, \alpha_n \qquad \mathrm{rng}(\mathcal{L}) = \{\overline{\alpha}\} \qquad \mathcal{L}\ \text{is bijective} \qquad \forall i, j \in 1..n :\ i < j \Leftrightarrow \mathcal{L}^{-1}(\alpha_i) <_{Paths} \mathcal{L}^{-1}(\alpha_j)}{\vdash \mathcal{L}\ \mathsf{locates}\ \overline{\alpha}}\ (\textsf{LOC})$$

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

$$\frac{(\Sigma_2 \circ \mathcal{L}_1^{-1}) \uplus (\Sigma_1 \circ \mathcal{L}_2^{-1}) = \{\alpha_1 \mapsto \mathrm{A}_1, \ldots, \alpha_n \mapsto \mathrm{A}_n\} \qquad \delta_0 = \{\} \qquad \forall i, j \in 1..n \text{ s.t. } i \le j :\quad \alpha_j \notin \mathrm{fv}(\mathrm{A}_i) \quad \delta_i = \delta_{i-1} \uplus \{\alpha_i \mapsto \delta_{i-1}\mathrm{A}_i\}}{\vdash (\mathcal{L}_1; \Sigma_1) \rightleftarrows (\mathcal{L}_2; \Sigma_2) \rightsquigarrow \delta_n}\ (\textsf{LOOKUP})$$

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

$$\frac{\vdash \Sigma_2 + \Sigma_1 \Rightarrow \Sigma}{\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma}\ (\textsf{MSYM})$$

$$\frac{}{\vdash [\![= \mathrm{A}]\!] + [\![= \mathrm{A}]\!] \Rightarrow [\![= \mathrm{A}]\!]}\ (\textsf{MTYP}) \qquad \frac{\vdash \mathrm{A}_1 \le \mathrm{A}_2}{\vdash [\![\mathrm{A}_1]\!]^{\pm} + [\![\mathrm{A}_2]\!]^{-} \Rightarrow [\![\mathrm{A}_1]\!]^{\pm}}\ (\textsf{MVAL})$$

$$\frac{}{\vdash \Sigma + \{\!|\}\!| \Rightarrow \Sigma}\ (\textsf{MEMP}) \qquad \frac{\ell \notin \overline{\ell_2} \qquad \vdash \{\!|\overline{\ell_1 : \Sigma_1}|\!\} + \{\!|\overline{\ell_2 : \Sigma_2}|\!\} \Rightarrow \{\!|\overline{\ell_3 : \Sigma_3}|\!\}}{\vdash \{\!|\ell : \Sigma, \overline{\ell_1 : \Sigma_1}|\!\} + \{\!|\overline{\ell_2 : \Sigma_2}|\!\} \Rightarrow \{\!|\ell : \Sigma, \overline{\ell_3 : \Sigma_3}|\!\}}\ (\textsf{MSTR1})$$

$$\frac{\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma_3 \qquad \vdash \{\!|\overline{\ell_1 : \Sigma_1'}|\!\} + \{\!|\overline{\ell_2 : \Sigma_2'}|\!\} \Rightarrow \{\!|\overline{\ell_3 : \Sigma_3'}|\!\}}{\vdash \{\!|\ell : \Sigma_1, \overline{\ell_1 : \Sigma_1'}|\!\} + \{\!|\ell : \Sigma_2, \overline{\ell_2 : \Sigma_2'}|\!\} \Rightarrow \{\!|\ell : \Sigma_3, \overline{\ell_3 : \Sigma_3'}|\!\}}\ (\textsf{MSTR2})$$

Fig. 6. Typing Rules for MixML (continued)

Thanks to 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 omit the Δ's in the presentation of our rules. We write $\overline{\alpha}$ fresh in the premise of a rule to mean that none of $\overline{\alpha}$ occurs in the context (Γ or $\Gamma; \mathcal{R}; \overline{\beta}$, respectively) of the conclusion of the rule.

Following 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 ($\vdash_{\mathrm{stat}}$). The static judgment is used to implement the static pass of recursive linking, as described in Section 3. To obtain the static version of any rule, simply remove all shaded premises, and replace all $\vdash$'s with $\vdash_{\mathrm{stat}}$'s.

Most of the rules for basic modules are fairly straightforward. Notably, rule VAR always returns a signature $|\Sigma|$ from the context, which only has exports. The reason is

that, regardless of whether the module that a variable X is bound to—call it mod—is fully defined, the module expression X is a *definite* reference to mod and is therefore itself fully defined. To take a concrete example, consider the structure $\{\mathsf{A} = [:\tau], \mathsf{B} = \mathsf{A}\}$, in which A is an import and B is an export. The atomic term module $[:\tau]$, to which A is bound, has signature $[\![\tau]\!]^-$. However, in the environment under which the binding of B is type-checked, it is given signature $[\![\tau]\!]^+$. If we did not switch the polarity for the signatures of module variables (which happens in rules LINK and SEAL), then B would have signature $[\![\tau]\!]^-$, too, and itself be considered an import, which is clearly wrong. Also, recall the encoding of sig_1 `matches` sig_2 in Section 2. In the encoding, we bind sig_1 to a variable X, and then check that the linking of X with sig_2 results in a complete module. This only works because X will *export* a component corresponding to each of sig_1's imports, and those exports will be used to satisfy all the imports of sig_2.

Rule ITYP for type imports $[:knd]$ uses the import realizer to look up the definition of its type import (which is A). The other rules for type and term modules are straightforward. Note that rule EVAL has a shaded premises because terms are ignored during the static pass. Instead, an arbitrary well-formed type A may be "guessed"—but as we will see in Section 9.1, this guess can be made fairly deterministic.

Typing namespaces (rule STR) and projection (rule DOT) is also straightforward. The latter rule requires the signatures of all components other than the one being projected out to be of the form $|\Sigma|$. This enforces that no term imports (of signature $[\![\mathrm{A}]\!]^-$) are being hidden. (The reasons for enforcing this are discussed in Section 2.) The hidden components cannot contain type imports either, as the import realizer in the premise contains only the projected label ℓ.

Rule LINK handles recursive linking. It is the central rule of our system, 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 $\mathcal{L}$'s and $\mathcal{R}$'s. 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 bidirectional lookup (rule LOOKUP) to look up mod_1's type imports in mod_2 and vice versa. (RMC only employs unidirectional lookup.)

The bidirectional lookup judgment will fail if it detects any transparent type cycles (manifest in the side condition $\alpha_j \notin \mathrm{fv}(A_i)$ for all $i \leq j$). 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 type-check 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 a straightforward auxiliary judgment (rules MSYM–MSTR2); it assumes the existence of a core type subsumption $\vdash \mathrm{A}_1 \leq \mathrm{A}_2$, forming a partial order on types. (Note that rule MTYP relies on our assumption that all core types are implicitly normalized.)

Now about those $\mathcal{L}$'s and $\mathcal{R}$'s: To deal with type imports and cross-eyed double vision, the linking rule has to properly adjust locators and realizers as it proceeds. The input realizer is first split into $\mathcal{R}$, $\mathcal{R}_1$ and $\mathcal{R}_2$, such that $\mathcal{R} \uplus \mathcal{R}_1$ contains the imports (of the linked module) stemming from mod_1, and $\mathcal{R} \uplus \mathcal{R}_2$ those from mod_2. (That is, $\mathcal{R}$ contains the imports shared by both sides. The side conditions on $\mathcal{R}_1$ and $\mathcal{R}_2$ ensure that they do not overlap with any components from the respective other side, so that the partitioning $\mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{R}_2$ is always uniquely determined.) In addition, each module may have additional *local* imports, *i.e.,* imports that the other module will satisfy by providing as exports. These are handled by locally extending the realizer with fresh locators $\mathcal{L}_1$ and $\mathcal{L}_2$, which are later used for type lookup. For the final pass on mod_2, δ

is applied to $\mathcal{L}_2$, turning it into a realizer that allows mod_2 to see all type definitions from mod_1. (Consequently, we do not need to apply δ to Σ_2 again when we merge the signatures.)

Let us walk through the cross-eyed double vision example from Section 3. Because the linked module has neither imports nor abstract type exports, rule LINK will be invoked with the following instantiation: $\mathcal{R} = \mathcal{R}_1 = \mathcal{R}_2 = \{\!|\}$ and $\overline{\beta}_1 = \overline{\beta}_2 = \emptyset$. However, it will introduce fresh variables α_1 and α_2 for the local imports of both sides and define $\mathcal{L}_1 = \{\!|\mathtt{t} : [\![= \alpha_1]\!]|\!\}$ and $\mathcal{L}_2 = \{\!|\mathtt{u} : [\![= \alpha_2]\!]|\!\}$ as local realizers for the l.h.s. and r.h.s., respectively. Traversing into the l.h.s. delivers $\Sigma_1 = \{\!|\mathtt{t} : [\![= \alpha_1]\!], \mathtt{u} : [\![= \mathtt{int}]\!], \mathtt{f} : [\![\alpha_1 \to \alpha_1]\!]^+|\!\}$. Next, the static pass on the r.h.s. is performed, with a guess for the type of g, yielding $\Sigma_2' = \{\!|\mathtt{t} : [\![= \mathtt{bool}]\!], \mathtt{u} : [\![= \alpha_2]\!], \mathtt{g} : [\![\forall \alpha.\alpha]\!]^+|\!\}$. Note that we just picked the type $\forall \alpha.\alpha$ for g, assuming for a minute that the core language provides ML-style polymorphism—in this particular example, any type would work, but in general we may need to be slightly more careful with picking a suitable type (see Section 9.1).

Using Σ_1, Σ_2', $\mathcal{L}_1$, and $\mathcal{L}_2$, bidirectional type lookup returns the substitution $\delta = \{\alpha_1 \mapsto \mathtt{bool}, \alpha_2 \mapsto \mathtt{int}\}$. For the main pass on the r.h.s., δ is applied to Σ_1 and to the locator $\mathcal{L}_2$, turning the latter into the realizer $\{\!|\mathtt{u} : [\![= \mathtt{int}]\!]|\!\}$. Thus, in this pass, the type-checker will see that `X.t` is `bool`, and it will know to implement `u` as `int`, thus avoiding cross-eyed double vision. Finally, it will return the signature $\Sigma_2 = \{\!|\mathtt{t} : [\![= \mathtt{bool}]\!], \mathtt{u} : [\![= \mathtt{int}]\!], \mathtt{g} : [\![\mathtt{int} \to \mathtt{bool}]\!]^+|\!\}$, which can be merged successfully with $\delta\Sigma_1$.

Rule SEAL for opaque linking is very similar to rule LINK, but slightly simpler because the result of opaque linking is not permitted to have any residual imports. (This is enforced by requiring the incoming realizer to be empty, and requiring that the merged signature have the form $|\Sigma|$.) In addition, the type imports of mod_1 (the $\overline{\alpha_1}$), which mod_2 must satisfy, become abstract type *exports* for the whole module, while the abstract type exports of mod_2 (the $\overline{\beta_2}$), are fresh variables only introduced into scope locally (since mod_2 is hidden). The final signature of the module is derived solely from the signature of mod_1—all information about mod_2 is kept secret. Finally, note that the $\overline{\alpha_1}$ are not just local and may in fact occur in Γ. This is the case, *e.g.,* if there is a recursive binding for the sealed module. Applying δ to Γ locally replaces those abstract types by their implementations, thereby preventing double vision.

Consider the following example of opaque linking:

$$\left\{\begin{array}{l}\mathtt{t} = \mathtt{[:type]},\\ \mathtt{u} = \mathtt{[int]},\\ \mathtt{f} = \mathtt{[:u \to t]}\end{array}\right\} \mathtt{seals} \left\{\begin{array}{l}\mathtt{t} = \mathtt{[bool]},\\ \mathtt{u} = \mathtt{[:type]},\\ \mathtt{f} = \mathtt{[\lambda x{:}u.true]}\end{array}\right\}$$

The linked module creates a single abstract export type, say α. Neither constituent module creates any abstract types independent of the sealing, so rule SEAL is applied with $\overline{\beta_1} = \overline{\beta_2} = \emptyset$ and $\overline{\alpha_1} = \alpha$. Then, $\Sigma_1 = \{\!|\mathtt{t} : [\![= \alpha]\!], \mathtt{u} : [\![= \mathtt{int}]\!], \mathtt{f} : [\![\mathtt{int} \to \alpha]\!]^-|\!\}$ is derived for the l.h.s. The r.h.s. locally imports `u`, so we choose a single fresh α_2, and $\Sigma_2' = \{\!|\mathtt{t} : [\![= \mathtt{bool}]\!], \mathtt{u} : [\![= \alpha_2]\!], \mathtt{f} : [\![\forall \beta.\beta]\!]^+|\!\}$ will be returned by the static pass (where $\forall \beta.\beta$ is just a guess at the type of `f`). With lookup returning $\delta = \{\alpha \mapsto \mathtt{bool}, \alpha_2 \mapsto \mathtt{int}\}$, the main pass proceeds under context $\delta\Gamma$, where any forward references to α are replaced by `bool`. The main pass yields $\Sigma_2 = \{\!|\mathtt{t} : [\![= \mathtt{bool}]\!], \mathtt{u} : [\![= \mathtt{int}]\!], \mathtt{f} : [\![\mathtt{int} \to \mathtt{bool}]\!]^+|\!\}$. Merging $\delta\Sigma_1$ and Σ_2 produces $\Sigma = \Sigma_2$, and since Σ has no residual imports, we have $\Sigma = |\Sigma|$. In contrast to rule LINK, Σ is not taken as the final signature. Instead, the signature $|\Sigma_1| = \{\!|\mathtt{t} : [\![= \alpha]\!], \mathtt{u} : [\![= \mathtt{int}]\!], \mathtt{f} : [\![\mathtt{int} \to \alpha]\!]^+|\!\}$ is returned. Note how it keeps `t` abstract (using the type name α passed in from the context), while marking `f` as an export.

Rules EUN and NEW, dealing with unit introduction and elimination, are very simple. The former invokes the unit typing judgment described below. The latter instantiates the given unit by choosing an appropriate substitution δ for the unit's import and export type names, and then applying δ to the signature Σ of the unit's constituent mod-

ule. Although the rule appears nondeterministic, the choice of δ is in fact completely determined by the actual context, which is an input to type-checking. Concretely, what really is happening is this: we want to type-check `new` mod under some given context $\Gamma; \mathcal{R}; \overline{\beta}'$. To do so, we type-check mod, yielding a unit signature $\forall\overline{\alpha}.\, \exists\overline{\beta}.\, (\mathcal{L}; \Sigma)$. Now δ is simply the (unique) substitution mapping $\mathcal{L}$ to $\mathcal{R}$ and $\overline{\beta}$ to $\overline{\beta}'$. In other words, δ's role is both renaming $\overline{\beta}$ and matching $\mathcal{L}$ against the actual realizer imposed by the program context (*e.g.,* through enclosing uses of linking).

Projection of (core) types and terms (rules PTYP and PVAL) requires the module being projected from to be *complete* (*i.e.,* without imports). This prevents meaningless examples like `typ([:type])` or `val([:int])`. Note that projections of the form $\texttt{typ}(\mathrm{X}.\ell s)$ or $\texttt{val}(\mathrm{X}.\ell s)$ are always acceptable, because variables are definite references (see above). The latter may, however, raise a runtime "blackhole" exception at runtime if the component $\mathrm{X}.\ell s$ refers to is as yet undefined (Section 8).

Completeness is ensured by rule COMPL, which checks that mod neither has type imports (by passing in an empty realizer) nor term imports (the signature is of the form $|\Sigma|$). Local abstract types $\overline{\beta}$ may not escape their scope by appearing in Σ.

Finally, rule UNIT type-checks a module as a self-contained unit. It introduces fresh names for import types ($\overline{\alpha}$) and export types ($\overline{\beta}$), which become quantified in the resulting unit signature. While the rule appears to have to guess the structure of the type locator $\mathcal{L}$, as well as the number, order, and kinds of $\overline{\alpha}$ and $\overline{\beta}$, out of nowhere, there is in fact only one way to choose them, which is easy to compute algorithmically by a simple pre-pass over mod (Section 9). Intuitively, that is because both $\mathcal{L}$ and $\overline{\beta}$ are treated in a linear fashion by the rules. The content of the former is only consumed in rules ITYP or NEW, and the latter by rules SEAL or NEW.

4.3. Differences from the Conference Version of the Type System

The current presentation of MixML's basic type system does not deviate much from the conference version of this article [Dreyer and Rossberg 2008]. Besides the straightforward generalization to higher-order kinds, and a couple of minor stylistic changes, we have primarily cleaned up the following details:

(1) We now enforce that all bindings in the environment Γ have signatures of the form $|\Sigma|$, whereas in the previous version, application of the $|\_|$-operator was deferred to the variable rule. This is merely a technical change that eases the correctness proof for our translation in Section 8.3.
(2) In the static pass, we no longer erase atomic term export signatures $[\![\mathrm{A}]\!]^+$ to $\{\!\}$. Consequently, the static version of rule EVAL now requires a non-deterministic guess of a proper type A. This change was necessary in order to support units as first-class values (see Section 6). Without the change, the new rule PACKAGE concerning package types could produce results in the static pass that differ from those in the regular pass, because unit signatures Φ would be structurally different in the two passes. Moreover, we feel our present approach is cleaner.
(3) The conclusion of rule LINK specifies the partitioning of its input realizer more strictly. In the conference version, we allowed $\mathcal{R}_1$ and $\mathcal{R}_2$ to overlap (written as $\mathcal{R}_1 \cup \mathcal{R}_2$) instead of making the common $\mathcal{R}$ explicit. The underspecification of $\mathcal{R}_1$ and $\mathcal{R}_2$ makes the completeness proof for our type-checking algorithm non-obvious (specifically, completeness of "template" computation in Section 9.2). Now we require a partition $\mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{R}_2$, and the additional side conditions $\mathcal{R}_1 \# \Sigma_2$ and $\mathcal{R}_2 \# \Sigma_1$ uniquely determine the domain of each part.

$$\begin{array}{lrcl} \text{Modules} & \mathit{mod} & ::= & \ldots \mid \texttt{[:}\mathit{usig}\texttt{]} \\ \text{Unit Signatures} & \mathit{usig} & ::= & \mathit{mod}\,\texttt{import}\,\overline{\ell s} \mid \mathit{mod}\,\texttt{export}\,\overline{\ell s} \end{array}$$

Fig. 7. Higher-Order MixML Syntax Extensions

$$\begin{array}{lrcl} \text{Module Signatures} & \Sigma & ::= & \ldots \mid [\![\Phi]\!]^- \\ \text{Unit Signatures} & \Phi & ::= & \forall\overline{\alpha}.\,\exists\overline{\beta}.\,(\mathcal{L}_1;\mathcal{L}_2;\Sigma) \end{array}$$

$$\forall\overline{\alpha}.\,\exists\overline{\beta}.\,(\mathcal{L};\Sigma) \stackrel{\text{def}}{=} \forall\overline{\alpha}.\,\exists\overline{\beta}.\,(\mathcal{L};\mathcal{L}';\Sigma) \text{ for some } \mathcal{L}'$$

$$\begin{array}{lcllcl} |[\![=\mathrm{A}]\!]| & \stackrel{\text{def}}{=} & [\![=\mathrm{A}]\!] & -[\![=\mathrm{A}]\!] & \stackrel{\text{def}}{=} & [\![=\mathrm{A}]\!] \\ |[\![\mathrm{A}]\!^{\pm}| & \stackrel{\text{def}}{=} & [\![\mathrm{A}]\!]^{+} & -[\![\mathrm{A}]\!^{\pm} & \stackrel{\text{def}}{=} & [\![\mathrm{A}]\!]^{\mp} \\ |[\![\Phi]\!]^{\pm}| & \stackrel{\text{def}}{=} & [\![\Phi]\!]^{+} & -[\![\Phi]\!]^{\pm} & \stackrel{\text{def}}{=} & [\![\Phi]\!]^{\mp} \\ |\{\!|\overline{\ell:\Sigma}|\!\}| & \stackrel{\text{def}}{=} & \{\!|\overline{\ell:|\Sigma|}|\!\} & -\{\!|\overline{\ell:\Sigma}|\!\} & \stackrel{\text{def}}{=} & \{\!|\overline{\ell:-\Sigma}|\!\} \end{array}$$

$$\begin{array}{lcllcl} |\Sigma|_{\epsilon} & \stackrel{\text{def}}{=} & |\Sigma| & \mathcal{L}\setminus\epsilon & \stackrel{\text{def}}{=} & \{\!|\}\!\} \\ |\{\!|\ell:\Sigma,\overline{\ell':\Sigma'}|\!\}|_{\ell.\ell s} & \stackrel{\text{def}}{=} & \{\!|\ell:|\Sigma|_{\ell s},\overline{\ell':\Sigma'}|\!\} & \{\!|\ell:\mathcal{L},\overline{\ell':\mathcal{L}'}|\!\}\setminus\ell.\ell s & \stackrel{\text{def}}{=} & \{\!|\ell:\mathcal{L}\setminus\ell s,\overline{\ell':\mathcal{L}'}|\!\} \\ |\Sigma|_{\ell s_1,\ldots,\ell s_n} & \stackrel{\text{def}}{=} & |\ldots|\Sigma|_{\ell s_1}\ldots|_{\ell s_n} & \mathcal{L}\setminus\ell s_1,\ldots,\ell s_n & \stackrel{\text{def}}{=} & \mathcal{L}\setminus\ell s_1\ldots\setminus\ell s_n \end{array}$$

Fig. 8. Higher-Order MixML Semantic Object Extensions and Notation

5. HIGHER-ORDER MIXML

The language presented in the previous section provides only first-order units. In this section we extend it with *higher-order units*, which enable units to be parameterized over other units. Higher-order units thus subsume the functionality of higher-order functors in traditional ML-style module systems (albeit with an SML-style generative semantics, not an OCaml-style applicative semantics).

5.1. Syntax

Figure 7 shows the syntactic extensions necessary for supporting higher-order units, relative to the "basic" language from Figure 1. Essentially, all that is needed is to add atomic unit imports `[:`usig`]`. However, to describe a unit import, it is necessary to introduce a new syntactic class usig of *unit signatures*.

A unit signature takes one of two symmetric forms, written $\mathit{mod}\,\texttt{import}\,\overline{\ell s}$ and $\mathit{mod}\,\texttt{export}\,\overline{\ell s}$. 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 $\mathit{mod}\,\texttt{import}\,\overline{\ell s}$, the list $\overline{\ell s}$ of paths enumerates all components of mod that are to be considered imports, treating all others as exports. Conversely, $\mathit{mod}\,\texttt{export}\,\overline{\ell s}$ lists the exports, and treats all other components as imports. A path $\ell s \in \overline{\ell s}$ may point to an entire structure in mod, in which case the annotation applies to all its subcomponents.

For example, recall the unit $\mathtt{U_A}$ from the end of Section 2.

$$[(\mathrm{X} = \texttt{new}\ \mathrm{S})\ \texttt{with}\ \{\mathtt{A} = \mathit{mod}_{\mathtt{A}}\}]$$

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

$$\frac{\Gamma \vdash usig \rightsquigarrow \Phi}{\Gamma; \{\!|\}; \emptyset \vdash \texttt{[:}\, usig \,\texttt{]} : [\![\Phi]\!]^-} \ (\textsf{IUN})$$

Unit Signatures: $\Gamma \vdash usig \rightsquigarrow \Phi$

$$\frac{\begin{array}{c}\Gamma \vdash mod : \forall\overline{\alpha}.\, \exists\emptyset.\, (\mathcal{L}; \Sigma) \qquad |\Sigma| = -\Sigma \\ \mathcal{L} = \mathcal{L}_1 \uplus \mathcal{L}_2 \qquad \vdash \mathcal{L}_1 \text{ locates } \overline{\alpha_1} \qquad \vdash \mathcal{L}_2 \text{ locates } \overline{\alpha_2} \qquad \mathcal{L}_1 = \mathcal{L} \setminus \overline{\ell s}\end{array}}{\Gamma \vdash mod\ \texttt{export}\ \overline{\ell s} \rightsquigarrow \forall\overline{\alpha_1}.\, \exists\overline{\alpha_2}.\, (\mathcal{L}_1; \mathcal{L}_2; |\Sigma|_{\overline{\ell s}})} \ (\textsf{EXPORT})$$

$$\frac{\Gamma \vdash mod\ \texttt{export}\ \overline{\ell s} \rightsquigarrow \forall\overline{\alpha_1}.\, \exists\overline{\alpha_2}.\, (\mathcal{L}_1; \mathcal{L}_2; \Sigma)}{\Gamma \vdash mod\ \texttt{import}\ \overline{\ell s} \rightsquigarrow \forall\overline{\alpha_2}.\, \exists\overline{\alpha_1}.\, (\mathcal{L}_2; \mathcal{L}_1; -\Sigma)} \ (\textsf{IMPORT})$$

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

$$\frac{}{\vdash [\![\Phi]\!]^- + [\![\Phi]\!]^- \Rightarrow [\![\Phi]\!]^-} \ (\textsf{MUN1}) \qquad \frac{\vdash \Phi_1 \leq \Phi_2}{\vdash [\![\Phi_1]\!]^+ + [\![\Phi_2]\!]^- \Rightarrow [\![\Phi_1]\!]^+} \ (\textsf{MUN2})$$

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

$$\frac{\vdash (\mathcal{L}_{11}; \Sigma_1) \rightleftarrows (\mathcal{L}_{22}; \Sigma_2) \rightsquigarrow \delta \qquad \vdash \delta\Sigma_1 + -\delta\Sigma_2 \Rightarrow |\Sigma|}{\vdash \forall\overline{\alpha_1}.\, \exists\overline{\beta_1}.\, (\mathcal{L}_{11}; \mathcal{L}_{12}; \Sigma_1) \leq \forall\overline{\alpha_2}.\, \exists\overline{\beta_2}.\, (\mathcal{L}_{21}; \mathcal{L}_{22}; \Sigma_2)} \ (\textsf{MATCH})$$

Fig. 9. Higher-Order MixML Type System Extensions

We can assign it the following unit signature:

$$(\texttt{new}\ \mathrm{S})\ \texttt{export A}$$

Or alternatively:

$$(\texttt{new}\ \mathrm{S})\ \texttt{import B}$$

We provide both forms merely as a convenience.

The encoding of the functor F given in Section 2,

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

can be classified with a unit signature as follows (assuming that sig' is a suitable specification of its body mod):

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

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

But unit signatures are more flexible than that. For example, the following variation of the above unit signature has no counterpart in traditional ML, assuming sig contains occurrences of X:

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

This would be the signature for a "functor" whose argument signature depends on the functor's own result! Imports and exports can also be specified in mutual recursion, by making suitable use of linking or the `rec` form we defined earlier—a use case would be describing the unit signatures of the units `MkTree` and `MkForest` from Figure 3. (Moreover, we are obviously not limited, as functors are, to grouping imports and exports into separate structures.)

Unit signature bindings (*i.e.,* bindings of unit signatures $usig$ to signature variables S for convenience) are easy to encode as well. Just as with regular signature bindings, we simply suspend the unit signature using a unit. That is, we bind $\mathrm{S} = \texttt{[[:}usig\texttt{]]}$. Then, whenever we wish later in the program to create a unit import U of signature $usig$, we simply bind $\mathrm{U} = \texttt{new}\ \mathrm{S}$. As units subsume functors, this demonstrates how MixML encodes functor signature bindings, of the kind that exist in higher-order dialects of the ML module system.

5.2. Semantic Objects

In order to be able to express unit imports, the definition of semantic objects has to be extended in two respects, shown in Figure 8:

(1) Atomic unit signatures can be marked as imports with a negative polarity as in $[\![\Phi]\!]^-$, analogously to atomic term signatures.
(2) Unit signatures Φ contain two type locators $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively mapping the import types $\overline{\alpha}$ and abstract export types $\overline{\beta}$. The export locator $\mathcal{L}_2$ is used for higher-order unit signature matching and thus only is needed when representing the translation of MixML-level $usig$'s; we omit it in other places.

The extensions to semantic signatures come with an adapted definition of absolute signatures $|\Sigma|$, and a new meta-operator $-\Sigma$ that switches the polarities of all atomic term and unit signatures projectible from Σ. The other meta-operations are explained below.

5.3. Typing Rules

Figure 9 shows the additional rules (and changes to existing rules) that are necessary to incorporate unit imports.

Rule IUN is the obvious rule for unit imports. Note that rule NEW is left unchanged: it still requires that mod be a unit export module, *i.e.,* that it actually contains a unit definition. Again, the unit it *contains* is free to have imports, which will become imports of `new` mod itself. More concretely, the premise prevents examples like $\texttt{new [:}usig\texttt{]}$ from type-checking, which would instantiate a non-existent unit, but permits $\texttt{new [[:}usig\texttt{]]}$, which creates a module consisting of a single unit import (as seen in the encoding of unit signature bindings given earlier).

The two rules for elaborating unit signatures are simpler than they might look. Rule EXPORT checks that mod is like an ML signature in that it does not export any fresh abstract types or term components ($|\Sigma| = -\Sigma$). It then partitions the components according to the paths listed in $\overline{\ell s}$: the notation $|\Sigma|_{\overline{\ell s}}$ (defined in Figure 8) turns those components reachable from $\overline{\ell s}$ into exports and leaves the others unchanged as imports. The import type variables and the respective locator are partitioned in a similar way. The second form of unit signature (with an `import` clause) is handled in the dual manner. Note that both the notations $|\Sigma|_{\overline{\ell s}}$ and $\mathcal{L} \setminus \overline{\ell s}$ require that all paths from $\overline{\ell s}$ actually exist in the respective signature or locator.

One interesting restriction is that the merging of two unit imports (rule MUN1) does not allow their signatures to differ. This restriction is in place in order to ensure principal types, as we will explain in Section 5.4 below. In contrast, when matching unit

exports against unit imports (rule MUN2), merging allows subtyping in the form of unit signature matching. The restriction on merging unit imports does not place any limitations, though, on the MixML encoding of ML-style higher-order functors, since that encoding will never attempt to merge two unit imports—ML signature matching always amounts to merging an export with an import, either in co- or in contravariant direction.

Rule MATCH defines unit 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. If this succeeds without any remaining imports, we know that the exports of subsignature Φ_1 in fact match the exports of Φ_2, and conversely for the (contravariant) imports of the two signatures. 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. If the lookup succeeds, we know that the type exports of Φ_1 subsume those of Φ_2, and contravariantly, the type imports of Φ_2 subsume those of Φ_1. 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 (discussed in Section 9) ensure that Φ_2 must be the translation of some MixML $usig$.

5.4. Characteristics of Unit Signature Matching

We feel that rule MATCH is remarkably elegant—certainly it is the most concise formulation of higher-order signature matching that we have ever seen. At the same time, it is more general than standard functor signature matching, because it is based on *symmetric* first-order *merging*, which is more general than the *directed* first-order *matching* seen in conventional ML modules. However, this generality leads to characteristics that are slightly different than what the reader may expect.

Higher-order signature matching is typically understood as a form of co/contravariant subtyping [Harper et al. 1990; Dreyer et al. 2003; Rossberg et al. 2010]. Although unit signature matching in higher-order MixML plays a similar role, and we use a suggestive notation, it is not actually a subtype ordering on unit signatures: while it is easy to see that the relation is reflexive, it is neither transitive nor antisymmetric.

Let us abbreviate $\Phi(\Sigma) = \forall\emptyset.\,\exists\emptyset.\,(\{\!|\}; \{\!|\}; \Sigma)$. Then from rule MATCH we can derive the matching $\Phi(\{\!|\ell : [\![= \mathtt{int}]\!]\ell|\}) \leq \Phi(\{\!|\}\!\})$, as one would expect. It may be somewhat more surprising that the inverse $\Phi(\{\!|\}\!\}) \leq \Phi(\{\!|\ell : [\![= \mathtt{int}]\!]\ell|\})$ can also be derived, but this behavior follows naturally from our account of signature matching in terms of signature merging. Similarly, we can derive $\Phi(\{\!|\}\!\}) \leq \Phi(\{\!|\ell : [\![= \mathtt{bool}]\!]\ell|\})$, of course. The transitive relation $\Phi(\{\!|\ell : [\![= \mathtt{int}]\!]\ell|\}) \leq \Phi(\{\!|\ell : [\![= \mathtt{bool}]\!]\ell|\})$, however, does *not* hold, because `int` and `bool` are incompatible type definitions when merged directly.

Moreover, despite being in mutual matching relation, $\Phi(\{\!|\ell : [\![= \mathtt{int}]\!]\ell|\})$ and $\Phi(\{\!|\}\!\})$, are not equivalent unit signatures, as there are contexts in which only one of them is usable. For example, if $\mathrm{X} : [\![\Phi]\!]^+$ with Φ being one of the two signatures, then $(\mathtt{new}\ \mathrm{X}).\ell$ would only be well-typed given the former, while conversely, $\mathrm{X}\ \mathtt{with}\ [\mathtt{:}\{\ell = [\mathtt{bool}]\}]$ would demand the latter to avoid a type clash.

A consequence of this lack of anti-symmetry is that we had to opt for the rather conservative formulation of the higher-order merging rule MUN1 mentioned earlier. Consider the "obvious" relaxation of that rule, namely:

$$\frac{\vdash \Phi_1 \leq \Phi_2}{\vdash [\![\Phi_1]\!]^- + [\![\Phi_2]\!]^- \Rightarrow [\![\Phi_1]\!]^-}\ (\mathsf{MUN1'})$$

$$\begin{array}{lrcl} \textbf{Types} & \mathit{typ} & ::= & \ldots \mid \mathtt{pack}(\mathit{usig}) \\ \textbf{Terms} & \mathit{exp} & ::= & \ldots \mid \mathtt{pack}(\mathit{mod}) \\ \textbf{Modules} & \mathit{mod} & ::= & \ldots \mid \mathtt{unpack}(\mathit{exp}\ \mathtt{as}\ \mathit{usig}) \end{array}$$

Fig. 10. Syntax Extensions for Units as First-Class Values

$$\text{Type Constructors}\ \ \mathrm{A} ::= \ldots \mid \langle\!|\Phi|\!\rangle$$

Core-Language Terms: $\Gamma \vdash \mathit{exp} : \mathrm{A}$

$$\frac{\Gamma \vdash \mathit{mod} : \Phi}{\Gamma \vdash \mathtt{pack}(\mathit{mod}) : \langle\!|\Phi|\!\rangle}\ (\textsf{PACK})$$

Core-Language Types: $\Gamma \vdash \mathit{typ} \rightsquigarrow \mathrm{A}$

$$\frac{\Gamma \vdash \mathit{usig} \rightsquigarrow \Phi}{\Gamma \vdash \mathtt{pack}(\mathit{usig}) \rightsquigarrow \langle\!|\Phi|\!\rangle}\ (\textsf{PACKAGE})$$

Modules: $\Gamma; \mathcal{L}; \overline{\beta} \vdash \mathit{mod} : \Sigma$

$$\frac{\Gamma \vdash \mathit{exp} : \langle\!|\forall\overline{\alpha}.\, \exists\overline{\beta}.\, (\mathcal{L}; \Sigma)|\!\rangle \qquad \Gamma \vdash \mathit{usig} \rightsquigarrow \forall\overline{\alpha}.\, \exists\overline{\beta}.\, (\mathcal{L}; \Sigma) \qquad \mathrm{dom}(\delta) = \{\overline{\alpha}, \overline{\beta}\}}{\Gamma; \delta\mathcal{L}; \delta\overline{\beta} \vdash \mathtt{unpack}(\mathit{exp}\ \mathtt{as}\ \mathit{usig}) : \delta\Sigma}\ (\textsf{UNPACK})$$

Fig. 11. Type System Extensions for Units as First-Class Values

Along with the symmetry rule MSYM, this more permissive version would allow us picking either $[\![\Phi_1]\!]^-$ or $[\![\Phi_2]\!]^-$ as the resulting signature in the case that Φ_1 and Φ_2 mutually match each other. This is fine as long both choices are equivalent, but short of anti-symmetry, that is not generally the case. Hence, rule MUN1' would destroy principal types. For example,

$$\mathtt{[:\{}\ell\mathtt{=[int]\}]\ with\ [:\{\}]}$$

could be given either of the two incompatible signatures $[\![\Phi(\{\!|\ell : [\![= \mathtt{int}]\!]|\!\})]\!]^-$ or $[\![\Phi(\{\!|\}\!)]\!]^-$ from above.

It is worth noting that the same problem would also arise for term imports (rule MVAL) if we hadn't assumed that core subtyping is a partial order. Fortunately, this assumption holds true for many interesting languages (including ML, up to appropriate normalization of polymorphic types). For other languages, we would need to treat merging of term imports in a manner analogous to that of unit imports.

6. UNITS AS FIRST-CLASS VALUES

So far, the language we have presented provides modules and units as second-class objects, defined over a (mostly arbitrary) core language. However, it is sometimes desirable to choose or compute modules within core terms based on information that is only available at run time. In this section, we extend MixML with the ability to create *packages*, *i.e.,* units that are packaged up as first-class core-language values.

6.1. Syntax

Figure 10 gives the syntax for the package extension. The core-language expression $\texttt{pack}(\mathit{mod})$ creates a package from unit mod. The type of a package value can be denoted by $\texttt{pack}(\mathit{usig})$, where usig describes the signature of the embedded unit (Section 5). The module expression $\texttt{unpack}(\mathit{exp}\ \texttt{as}\ \mathit{usig})$ extracts the module from a package exp. Again, the unit signature usig describes the package's signature.

For example, assume there are two different modules, `TreeMap` and `HashMap`, providing implementations of an ADT for finite maps, with the same signature `MAP`. We can pack them up as first-class values:

$$\begin{array}{l}\texttt{let treeMap} = \texttt{pack(TreeMap) in}\\ \texttt{let hashMap} = \texttt{pack(HashMap) in} \ldots\end{array}$$

Somewhere else, we can pick one of these maps depending on the expected number n of elements (computed elsewhere) that have to be stored:

$$\texttt{unpack((if } n \leq 100 \texttt{ then treeMap else hashMap) as MAP import } \emptyset)$$

Using two packages interchangeably requires that they have the same type—in this case, $\texttt{pack(MAP import } \emptyset)$. If the units to be packaged have *different* signatures, but match a common "super "signature, like `MAP` in this example, then sealing can be used to explicitly coerce them to the common signature beforehand:

$$\begin{array}{l}\texttt{let treeMap} = \texttt{pack(TreeMap :> MAP) in}\\ \texttt{let hashMap} = \texttt{pack(HashMap :> MAP) in} \ldots\end{array}$$

The notion of first-class unit we define here differs slightly from previous formulations [Russo 1999a; Dreyer et al. 2003; Rossberg et al. 2010] in that it always embeds the argument module as a (suspended) unit instead of an (already evaluated) module. In the presence of higher-order modules, both formulations are equivalent in expressive power. The reason for our design is mainly technical simplicity: it allows us to reuse most of the typing rules for higher-order units and unit signatures. The more conventional form of first-class modules can easily be recovered through the following syntactic definitions:

$$\begin{array}{lcl}\texttt{pack}(\mathit{mod}\ \texttt{as}\ \mathit{sig}) & \stackrel{\text{def}}{=} & \texttt{let X} = \mathit{mod}\ \texttt{in pack(X :>}\ \mathit{sig})\\ \texttt{pack}(\mathit{sig}) & \stackrel{\text{def}}{=} & \texttt{pack}(\mathit{sig}\ \texttt{import}\ \emptyset)\\ \texttt{unpack}(\mathit{exp}\ \texttt{as}\ \mathit{sig}) & \stackrel{\text{def}}{=} & \texttt{unpack}(\mathit{exp}\ \texttt{as}\ \mathit{sig}\ \texttt{import}\ \emptyset)\end{array}$$

The effect of this encoding is that (1) packed modules are evaluated prior to suspension, *i.e.,* the suspension will be a 'constant' unit, and (2) packed modules may only contain exports, not imports, *i.e.,* they have to be complete. (Of course, because units are higher-order, they can nevertheless *export* a proper unit.)

6.2. Semantic Objects and Typing Rules

To represent packages internally, we extend the language of type constructors to include package types: a type $\langle\!|\Phi|\!\rangle$ classifies a package containing a unit with the signature Φ. We assume that the type well-formedness judgment $\vdash \mathrm{A} \Uparrow \mathit{knd}$ asserts that each semantic signature Φ in a package type is well-formed, written $\vdash \Phi \Uparrow$, a notion that will be made precise in Section 8.3.

Figure 11 presents the straightforward typing rules for packages. Rules PACK and PACKAGE mirror the rules EUN and IUN for (higher-order) units, except that the expressions form core terms and types instead of modules. Rule UNPACK in turn mirrors rule NEW for the `new` construct. It uses the signature annotation to derive the

package signature in the static pass of type-checking recursive linking, where we cannot derive it from the expression exp. In the main pass, the rule requires the actual package signature to coincide with the annotation. A more liberal rule only requiring the package signature to *match* the annotation would be feasible, but potentially interferes with type inference in the core language. (It should be mentioned that, apart from this choice, we ignore the issue of core-language type inference for this presentation. Consequently, unlike in previous work [Russo 1999a; Dreyer et al. 2003; Rossberg et al. 2010], we do not require a signature annotation on the `pack` construct. Should type inference be desired, this is straightforward to add.)

6.3. Signature Normalization

A somewhat subtle technical detail with packages is signature equivalence: the core language does not necessarily support coercive subtyping, so package types are only compatible when they embed *equivalent* signatures. In order to avoid over-restrictive typing, we at least want to make sure that seemingly equivalent *syntactic* unit signatures generate equivalent *semantic* signatures. For example, we want the syntactic types $\texttt{pack}(\{\texttt{type t}, \texttt{type u}\}\ \texttt{import}\ \emptyset)$ and $\texttt{pack}(\{\texttt{type u}, \texttt{type t}\}\ \texttt{import}\ \emptyset)$ to be represented by equivalent semantic types. In general, this requires *normalizing* semantic signatures [Rossberg et al. 2010]—specifically, imposing a canonical ordering on quantifiers binding abstract types like `t` and `u`.

Fortunately, in our type system, *explicit* unit signatures stemming from $usig$'s (and thus explicit package types) are normalized by construction. All such unit signatures are formed from modules that have only imports, and rule LOC for locators prescribes an ordering on import variables $\overline{\alpha}$ that depends only on the global ordering relation on paths. Furthermore, the type system ensures that all import variables of unit signature actually occur in the signature (unused import variables would correspond to "dropped" imports, which are not allowed). So both the above package types will be represented by the same semantic type $\langle\!|\forall\emptyset.\,\exists\beta_1\beta_2.\,(\{\!|\};\{\!|\texttt{t} : [\![= \beta_1]\!], \texttt{u} : [\![= \beta_2]\!]|\!\})|\!\rangle$ (assuming $\texttt{t} <_{Paths} \texttt{u}$).

However, the story is somewhat different for *implicit* unit signatures as assigned to module expressions by the type system: exports can freely be dropped from modules, *e.g.,* via projection. So derived signatures are not necessarily normalized with respect to their export variables. For example, the package

$$\texttt{pack}((mod\ \texttt{:>}\ \{\texttt{type t}, \texttt{type u}\}).\texttt{t})$$

has the semantic type $\langle\!|\forall\emptyset.\,\exists\beta_1\beta_2.\,(\{\!|\};[\![=\beta_1]\!])|\!\rangle$, which includes a spurious binding for the export type variable β_2 representing the local type `u`. Consequently, this package is incompatible with the package type $\texttt{pack}(\texttt{[:type]}\ \texttt{import}\ \emptyset)$, which is represented as $\forall\emptyset.\,\exists\beta.\,(\{\!|\};[\![=\beta]\!])$.

The problem could be avoided in the type system by normalizing the unit signature in rule PACK. We refer to Rossberg et al. [2010] for the details of that approach. Alternatively, it is always possible for the programmer to explicitly canonicalize a package by re-sealing with an explicit signature before packaging:

$$\texttt{pack}((mod\ \texttt{:>}\ \{\texttt{type t}, \texttt{type u}\}).\texttt{t}\ \texttt{:>}\ \texttt{[:type]})$$

This amounts to putting a type annotation on the `pack` construct, as is mandatory in most previous systems with first-class modules.

6.4. First-Class vs. Higher-Order Units

As a final remark, it should come as no surprise that the addition of units as first-class values *almost* subsumes higher-order units. That is, given packages, we can define

units via the following syntactic sugar:

$$\begin{array}{rcl} [mod] & \stackrel{\text{def}}{=} & [\texttt{pack}(mod)] \\ [\texttt{:}usig] & \stackrel{\text{def}}{=} & [\texttt{:pack}(usig)] \\ \texttt{new}\ mod\ \texttt{as}\ usig & \stackrel{\text{def}}{=} & \texttt{unpack}(\texttt{val}(mod)\ \texttt{as}\ usig) \end{array}$$

It should be intuitively clear from looking at the respective typing rules that this encoding yields the same typings. The only limitation is that we would have to require a signature annotation on every `new`. This limitation is due to the encoding of units as values, which implies that no type information about them is available during the static pass—yet this information is required for the type-checking of `new`.

In principle, it should be possible to refine the rules for the static pass such that it can derive type information at least for a limited subset of expressions—in particular, for simple cases like $\texttt{val}(mod)$ as needed for the unit encoding. We did not pursue this path, however, since the added complexity for such a refinement seems to far outweigh the simplification gained by eliminating the three typing rules for the unit constructs.

7. INTERNAL LANGUAGE

Giving a type-preserving direct-style operational semantics for a language with the type-theoretic complexity of ordinary ML is known to be very difficult, and doing the same for MixML is even more difficult. Instead, we will follow the "elaboration" approach of Harper and Stone [2000] and define the dynamic semantics of MixML by translation into a simpler and more standard internal language. Soundness of the translation (that is, preservation of well-typedness) together with soundness of the internal language is then sufficient to establish type soundness for MixML.

The internal language we employ is named *LTG* (standing for *linear type generativity*) and is a variant of Dreyer's RTG calculus for recursive type generativity [Dreyer 2007a]. However, it incorporates several simplifications and generalizations relative to RTG:

— Instead of tracking definedness of type names using an effect system, LTG employs a more general and uniform substructural type system [Walker 2005; Ahmed et al. 2005] that treats undefined type variables as linear capabilities.
— LTG allows cyclic definitions for abstract type variables. As a result, certain aspects of the type system become simpler—for example, we do not need to track what Dreyer [2007a] called "stability", nor do we need to treat recursive type definitions in some special way. LTG's type system remains sound, but becomes difficult to type-check in general, because type normalization might diverge and consequently, type equivalence is probably undecidable (at least we do not know any algorithm for deciding it). However, this has no effect on decidability of MixML's external type system, because that does not allow any transparent type cycles (see Section 9).
— LTG provides single-assignment references to express backpatching for terms. Unlike in the conference version of this article [Dreyer and Rossberg 2008], LTG's linear type system, together with soundness of elaboration, enables us to track that all components of a complete module are defined exactly once. (LTG's type system still does *not* check whether references are assigned before being accessed, a deliberate choice we made for the treatment of cyclic term definitions, see Section 3.)

7.1. LTG Basics

Figure 12 shows the syntax of LTG, along with some notational abbreviations that we will use throughout this article. As usual, we identify terms up to the renaming of bound variables (besides the usual binders, $\mathsf{new}\,\alpha{:}\kappa\,\mathsf{in}\,e$ binds α in e) and adopt Barendregt's hygiene convention that bound variables are assumed distinct from any free

$$
\begin{array}{lrcl}
\textbf{Modes} & \iota & ::= & \mathrm{U} \mid \mathrm{L} \\
& \hat{\kappa} & ::= & \kappa^\iota \\
& \hat{\tau} & ::= & \tau^\iota \\
\textbf{Kinds} & \kappa & ::= & \mathsf{type} \mid \kappa_1 \to \kappa_2 \\
\textbf{Types} & \tau & ::= & \hat{\tau}_1 \to \hat{\tau}_2 \mid \{\overline{\ell : \hat{\tau}}\} \mid \forall\alpha{:}\hat{\kappa}.\hat{\tau} \mid \exists\alpha{:}\hat{\kappa}.\hat{\tau} \mid ?\tau \mid \alpha \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\ \tau_2 \\
\textbf{Values} & v & ::= & \lambda x{:}\hat{\tau}.e \mid \{\overline{\ell{=}v}\} \mid \lambda\alpha{:}\hat{\kappa}.e \mid \langle \tau, v\rangle_{\tau'} \mid x \\
\textbf{Terms} & e & ::= & v \mid e_1\ e_2 \mid \{\overline{\ell{=}e}\} \mid \mathsf{let}\ \{\overline{\ell{=}x}\}{=}e_1\ \mathsf{in}\ e_2 \mid \mathsf{new}\ \tau \mid \mathsf{def}\ e_1{:=}e_2 \mid !\,e \\
& & & \mid e_1\ \tau_2 \mid \langle \tau, e\rangle_\tau \mid \mathsf{let}\ \langle\alpha, x\rangle{=}e_1\ \mathsf{in}\ e_2 \mid \mathsf{new}\ \alpha{:}\kappa\ \mathsf{in}\ e \mid \mathsf{def}\ \tau_1{:=}\tau_2\ \mathsf{in}\ (e{:}\hat{\tau})
\end{array}
$$

$$
\begin{array}{lrcl}
\textbf{Type environments} & \Delta & ::= & \epsilon \mid \Delta, \alpha{:}\hat{\kappa} \\
\textbf{Equivalence environments} & \Psi & ::= & \epsilon \mid \Psi, \alpha{:=}\tau \\
\textbf{Value environments} & \Gamma & ::= & \epsilon \mid \Gamma, x{:}\hat{\tau} \\
\textbf{Environments} & \Xi & ::= & \Delta; \Psi; \Gamma
\end{array}
$$

$$
\begin{array}{lcll}
\forall^\iota\overline{\alpha{:}\kappa}.\tau & \stackrel{\text{def}}{=} & \forall\alpha_1{:}\kappa_1^\iota.(\cdots\forall\alpha_n{:}\kappa_n^\iota.(\tau)^\iota\cdots)^\iota & \\
e_1; e_2 & \stackrel{\text{def}}{=} & \mathsf{let}\ \{\}{=}e_1\ \mathsf{in}\ e_2 & \\
e.\ell & \stackrel{\text{def}}{=} & \mathsf{let}\ \{\ell{=}x, \overline{\ell'{=}x'}\}{=}e\ \mathsf{in}\ x & \text{where } e : \{\ell{:}\hat{\tau}, \overline{\ell'{:}\hat{\tau}'}\} \\
e|_{\overline{\ell}} & \stackrel{\text{def}}{=} & \mathsf{let}\ \{\overline{\ell{=}x}, \overline{\ell'{=}x'}\}{=}e\ \mathsf{in}\ \{\overline{\ell{=}x}\} & \text{where } e : \{\overline{\ell{:}\hat{\tau}}, \overline{\ell'{:}\hat{\tau}'}\} \\
\{e, \ell{=}e'\} & \stackrel{\text{def}}{=} & \mathsf{let}\ \{\overline{\ell'{=}x'}\}{=}e\ \mathsf{in}\ \{\overline{\ell'{=}x'}, \ell{=}e'\} & \text{where } e : \{\overline{\ell'{:}\hat{\tau}'}\} \\
\mathsf{let}\ \overline{x{=}e}\ \mathsf{in}\ e' & \stackrel{\text{def}}{=} & \mathsf{let}\ \{\overline{\ell{=}x}\} = \{\overline{\ell{=}e}\}\ \mathsf{in}\ e' & \\
\mathsf{new}\ \overline{\alpha{:}\kappa}\ \mathsf{in}\ e & \stackrel{\text{def}}{=} & \mathsf{new}\ \alpha_1{:}\kappa_1\ \mathsf{in}\cdots\mathsf{new}\ \alpha_n{:}\kappa_n\ \mathsf{in}\ e & \\
\mathsf{def}\ \overline{\alpha{:=}\tau}\ \mathsf{in}\ e & \stackrel{\text{def}}{=} & \mathsf{def}\ \alpha_1{:=}\tau_1\ \mathsf{in}\cdots\mathsf{def}\ \alpha_n{:=}\tau_n\ \mathsf{in}\ e & \\
\Xi, \alpha{:}\hat{\kappa} & \stackrel{\text{def}}{=} & \Delta, \alpha{:}\hat{\kappa}; \Psi; \Gamma & \text{where } \Xi = \Delta; \Psi; \Gamma \text{ and } \alpha \notin \mathrm{fv}(\Psi, \Gamma) \\
\Xi, \alpha{:=}\tau & \stackrel{\text{def}}{=} & \Delta; \Psi, \alpha{:=}\tau; \Gamma & \text{where } \Xi = \Delta; \Psi; \Gamma \\
\Xi, x{:}\hat{\tau} & \stackrel{\text{def}}{=} & \Delta; \Psi; \Gamma, x{:}\hat{\tau} & \text{where } \Xi = \Delta; \Psi; \Gamma
\end{array}
$$

Fig. 12. LTG Syntax

variables appearing in the context. We write $\mathrm{fv}(\_)$ for the set of free (type and term) variables of a syntactic objects, and $\mathrm{dom}(\_)$ for the set of (type or term) variables in the domain of an environment or substitution.

If we ignore everything involving *mode* qualifiers ι and *environment splitting* $\Xi_1 * \Xi_2$ for a second, then the language is a relatively standard extension of System F_ω. Products take the form of labeled records, and we identify record types $\{\overline{\ell : \hat{\tau}}\}$ up to reordering of labels. Values of existential type are written $\langle \tau, v\rangle_{\exists\alpha:\hat{\kappa}.\hat{\tau}}$, where τ is the witness type and $\exists\alpha{:}\hat{\kappa}.\hat{\tau}$ an annotation determining a unique type for the value. In order to ease some of the developments in succeeding sections, we assume that type variables are implicitly kinded for LTG as well, and thus impose the syntactic restriction that in any occurrence of "$\alpha{:}\kappa$" it holds that $\kappa_\alpha = \kappa$.

Types of the form $?\tau$ classify *single-assignment references* (or just *references* hereafter) that—eventually—contain values of type τ. Single-assignment references enable the creation of "names" for values before the values are actually known, which is useful in modeling recursive linking. A fresh, uninitialized reference is created with the expression $\mathsf{new}\,\tau$. Only after defining it through assignment ($\mathsf{def}\ e_1{:=}e_2$) can it be successfully read ($!e$). References are non-strict: assignment does not evaluate e_2. Instead,

it stores the computation e_2 in the reference denoted by e_1, and this computation will be re-executed whenever e_1 is dereferenced. This semantics only really makes sense if e_2 is restricted to be a "pure" expression (where purity may include dereferencing of single-assignment references), and indeed this will always be the case in our elaboration translation.[4]

An analogous feature exists for types: the expression form $\mathsf{new}\,\alpha{:}\kappa\ \mathsf{in}\ e$ introduces a new abstract type name α that can be used within e. Later, within e, α can be defined via the construct $\mathsf{def}\,\alpha{:=}\tau\ \mathsf{in}\ (e' : \hat{\tau})$.[5] The type annotation $\hat{\tau}$ in the def form ensures unique types. The definition of α is only visible within e', with the knowledge about that equivalence recorded in the *equivalence environment* Ψ.[6] It can be utilized in the conversion rule (econv at the bottom of Figure 13), which invokes the type equivalence judgment $\Psi \vdash \tau_1 \equiv \tau_2$. (Unlike in the MixML rules, type equivalence is fully explicit in LTG, *i.e.,* we do not assume implicit normalization in this section.) Most of the rules for this judgment (Figure 13) are the standard (inductive) rules of System F_ω (β and η-reduction for type constructors, plus the necessary congruence rules); the only new rule, qdelta, allows invoking assumptions from the environment in the obvious manner.

Generative type names with a separate definition scope are the central feature taken from RTG [Dreyer 2007a]. Both references and type names together allow us to deal with the recursive nature of linking in MixML by "forward declaring" values and types and then using "backpatching" to define them. The details of this technique will be explained in Section 8.

However, when doing so, we want to make sure that every forward declaration has a corresponding definition. In RTG (and in the technical appendix to the conference version of this article [Dreyer and Rossberg 2008; Rossberg and Dreyer 2008]) the type system ensured that for type names by applying a simple, *ad hoc* notion of linearity (disguised as effects in RTG). No such guarantee was present for references, though. In the present work, we address this by moving to a more general form of linearity that uniformly covers type and value definitions.

7.2. Linearity

Our approach to linearity is to beef up the language with *substructural* mode qualifiers, ranged over by ι, that annotate every type [Walker 2005; Ahmed et al. 2005]. The only two modes are U (*unrestricted*) and L (*linear*).[7] A type that only contains U annotations has the same meaning as a plain old System F_ω type—and we sometimes take the liberty of dropping U annotations to avoid clutter. In contrast, a linear mode enforces that a respective value (or type name, see below) is consumed eventually. In the case of reference types, linearity only restricts *assignment*, not *reading*, which is always possible (see below). In that sense, our linear references differ fundamentally from previous work.

The typing judgment $\Xi \vdash e : \hat{\tau}$ classifies terms by moded types. Perhaps more surprisingly, the kinding judgment $\Delta \vdash \tau : \hat{\kappa}$ analogously classifies types by *moded kinds*, in order to deal with linearity of undefined type names. Environments Ξ are triples

[4] In the Technical Appendix to the conference version of this article [Dreyer and Rossberg 2008], we actually used *lazy* references, where e_2 would only be evaluated once and then memoized. But memoization was inessential to the semantics, so we dropped it for the sake of simplicity.

[5] In order to make it closed under substitution, the actual syntax of this construct is $\mathsf{def}\,\tau_1{:=}\tau_2\ \mathsf{in}\ (e : \hat{\tau})$. In any well-typed term, τ_1 will be a variable, and any well-formed type substitution will maintain this property.

[6] In the original RTG presentation, Ψ was folded into Δ.

[7] As a matter of terminology, we refer to an unannotated τ as a "type", and call τ^ι a *moded* type (which we range over by the meta-variable $\hat{\tau}$). Other authors often speak of τ as a "pre-type", and only consider τ^ι a type. In the higher-order setting we are dealing with here, the former terminology is somewhat more convenient.

Types: $\Delta \vdash \tau : \hat{\kappa}$

$$\frac{\Delta_1, \Delta_2 \preceq \mathrm{U}}{\Delta_1, \alpha{:}\hat{\kappa}, \Delta_2 \vdash \alpha : \hat{\kappa}}(\mathsf{tvar})$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{type}^{\mathrm{U}} \qquad \Delta \vdash \tau_2 : \mathsf{type}^{\mathrm{U}}}{\Delta \vdash \tau_1^{\iota_1} \to \tau_2^{\iota_2} : \mathsf{type}^{\mathrm{U}}}(\mathsf{tarr}) \qquad \frac{\Delta \vdash \tau_1 : \mathsf{type}^{\mathrm{U}} \quad \cdots \quad \Delta \vdash \tau_n : \mathsf{type}^{\mathrm{U}} \quad \Delta \preceq \mathrm{U}}{\Delta \vdash \{\overline{\ell : \tau^{\iota}}\} : \mathsf{type}^{\mathrm{U}}}(\mathsf{trec})$$

$$\frac{\Delta, \alpha{:}\kappa^{\mathrm{U}} \vdash \tau : \mathsf{type}^{\mathrm{U}}}{\Delta \vdash \forall \alpha{:}\kappa^{\iota_1}.\tau^{\iota_2} : \mathsf{type}^{\mathrm{U}}}(\mathsf{tall}) \qquad \frac{\Delta, \alpha{:}\kappa^{\mathrm{U}} \vdash \tau : \mathsf{type}^{\mathrm{U}}}{\Delta \vdash \exists \alpha{:}\kappa^{\iota_1}.\tau^{\iota_2} : \mathsf{type}^{\mathrm{U}}}(\mathsf{tex}) \qquad \frac{\Delta \vdash \tau : \mathsf{type}^{\mathrm{U}}}{\Delta \vdash ?\tau : \mathsf{type}^{\mathrm{U}}}(\mathsf{tref})$$

$$\frac{\Delta, \alpha{:}\kappa_1^{\mathrm{U}} \vdash \tau : \kappa_2^{\mathrm{U}}}{\Delta \vdash \lambda \alpha{:}\kappa_1.\tau : (\kappa_1 \to \kappa_2)^{\mathrm{U}}}(\mathsf{tfun}) \qquad \frac{\Delta \vdash \tau_1 : (\kappa_2 \to \kappa)^{\mathrm{U}} \qquad \Delta \vdash \tau_2 : \kappa_2^{\mathrm{U}}}{\Delta \vdash \tau_1\ \tau_2 : \kappa^{\mathrm{U}}}(\mathsf{tapp})$$

Type Equivalence: $\Psi \vdash \tau_1 \equiv \tau_2$

$$\frac{}{\Psi \vdash \alpha \equiv \Psi(\alpha)}(\mathsf{qdelta}) \qquad \frac{}{\Psi \vdash (\lambda \alpha{:}\kappa.\tau_1)\ \tau_2 \equiv \tau_1[\tau_2/\alpha]}(\mathsf{qbeta}) \qquad \frac{\alpha \notin \mathrm{fv}(\tau)}{\Psi \vdash \lambda \alpha{:}\kappa.\tau\ \alpha \equiv \tau}(\mathsf{qeta})$$

...and standard congruence rules...

Terms: $\Xi \vdash e : \hat{\tau}$

$$\frac{\Xi_1, \Xi_2 \preceq \mathrm{U}}{\Xi_1, x{:}\hat{\tau}, \Xi_2 \vdash x : \hat{\tau}}(\mathsf{evar})$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{type}^{\mathrm{U}} \qquad \Xi, x{:}\tau_1^{\iota_1} \vdash e : \hat{\tau}_2 \qquad \Xi \preceq \iota}{\Delta * \Xi \vdash \lambda x{:}\tau_1^{\iota_1}.e : (\tau_1^{\iota_1} \to \hat{\tau}_2)^{\iota}}(\mathsf{efun}) \qquad \frac{\Xi_1 \vdash e_1 : (\hat{\tau}_2 \to \hat{\tau})^{\iota} \qquad \Xi_2 \vdash e_2 : \hat{\tau}_2}{\Xi_1 * \Xi_2 \vdash e_1\ e_2 : \hat{\tau}}(\mathsf{eapp})$$

$$\frac{\Xi_1 \vdash e_1 : \hat{\tau}_1 \ \cdots\ \Xi_n \vdash e_n : \hat{\tau}_n \quad \overline{\hat{\tau} \preceq \iota} \quad \Xi_0 \preceq \mathrm{U}}{\Xi_0 * \Xi_1 * \cdots * \Xi_n \vdash \{\overline{\ell{=}e}\} : \{\overline{\ell : \hat{\tau}}\}^{\iota}}(\mathsf{erec}) \qquad \frac{\Xi_1 \vdash e_1 : \{\overline{\ell : \hat{\tau}}\}^{\iota} \quad \Xi_2, \overline{x{:}\hat{\tau}} \vdash e_2 : \hat{\tau}'}{\Xi_1 * \Xi_2 \vdash \mathsf{let}\ \{\overline{\ell{=}x}\}{=}e_1\ \mathsf{in}\ e_2 : \hat{\tau}'}(\mathsf{eprj})$$

$$\frac{\Xi, \alpha{:}\hat{\kappa} \vdash e : \hat{\tau} \qquad \Xi \preceq \iota}{\Xi \vdash \lambda \alpha{:}\hat{\kappa}.e : (\forall \alpha{:}\hat{\kappa}.\hat{\tau})^{\iota}}(\mathsf{egen}) \qquad \frac{\Xi \vdash e : (\forall \alpha{:}\hat{\kappa}.\hat{\tau})^{\iota} \qquad \Delta \vdash \tau_2 : \hat{\kappa}}{\Xi * \Delta \vdash e\ \tau_2 : \hat{\tau}[\tau_2/\alpha]}(\mathsf{einst})$$

$$\frac{\Delta_1 \vdash \tau_1 : \hat{\kappa} \qquad \Xi \vdash e : \hat{\tau}[\tau_1/\alpha] \qquad \Delta_2 \vdash \exists \alpha{:}\hat{\kappa}.\hat{\tau} : \mathsf{type}^{\mathrm{U}} \qquad \hat{\kappa} \preceq \iota \qquad \hat{\tau} \preceq \iota}{\Delta_1 * \Xi * \Delta_2 \vdash \langle \tau_1, e \rangle_{\exists \alpha:\hat{\kappa}.\hat{\tau}} : (\exists \alpha{:}\hat{\kappa}.\hat{\tau})^{\iota}}(\mathsf{epack})$$

$$\frac{\Xi_1 \vdash e_1 : (\exists \alpha{:}\hat{\kappa}_1.\hat{\tau}_2)^{\iota} \qquad \Xi_2, \alpha{:}\hat{\kappa}_1, x{:}\hat{\tau}_2 \vdash e_2 : \hat{\tau} \qquad \alpha \notin \mathrm{fv}(\hat{\tau})}{\Xi_1 * \Xi_2 \vdash \mathsf{let}\ \langle \alpha, x \rangle{=}e_1\ \mathsf{in}\ e_2 : \hat{\tau}}(\mathsf{eopen})$$

$$\frac{\Delta \vdash \tau : \mathsf{type}^{\mathrm{U}} \qquad \Xi \preceq \mathrm{U}}{\Delta * \Xi \vdash \mathsf{new}\ \tau : (?\tau)^{\mathrm{L}}}(\mathsf{enewe}) \qquad \frac{\Xi_1 \vdash e_1 : (?\tau)^{\mathrm{L}} \qquad \Xi_2 \vdash e_2 : \tau^{\mathrm{U}} \qquad \Xi_2 \preceq \mathrm{U}}{\Xi_1 * \Xi_2 \vdash \mathsf{def}\ e_1{:=}e_2 : \{\}^{\mathrm{U}}}(\mathsf{edefe})$$

$$\frac{\Xi, \alpha{:}\kappa^{\mathrm{L}} \vdash e : \hat{\tau} \qquad \alpha \notin \mathrm{fv}(\hat{\tau})}{\Xi \vdash \mathsf{new}\ \alpha{:}\kappa\ \mathsf{in}\ e : \hat{\tau}}(\mathsf{enewt}) \qquad \frac{\Delta_1 \vdash \alpha : \kappa^{\mathrm{L}} \qquad \Delta_2 \vdash \tau_2 : \kappa^{\mathrm{U}} \qquad \Xi, \alpha{:=}\tau_2 \vdash e : \hat{\tau}}{\Delta_1 * \Delta_2 * \Xi \vdash \mathsf{def}\ \alpha{:=}\tau_2\ \mathsf{in}\ (e : \hat{\tau}) : \hat{\tau}}(\mathsf{edeft})$$

$$\frac{\Xi \vdash e : (?\tau)^{\mathrm{U}}}{\Xi \vdash\ !e : \tau^{\mathrm{U}}}(\mathsf{eget}) \qquad \frac{\Xi \vdash e : \tau_1^{\iota} \qquad \Psi\ \mathbf{of}\ \Xi \vdash \tau_1 \equiv \tau_2 \qquad \Delta \vdash \tau_2 : \mathsf{type}^{\mathrm{U}}}{\Xi * \Delta \vdash e : \tau_2^{\iota}}(\mathsf{econv})$$

Fig. 13. LTG Static Semantics (Main Judgments)

Environments: $\vdash \Delta \quad \Delta \vdash \Psi \quad \Delta \vdash \Gamma \quad \vdash \Xi$

$$\frac{}{\vdash \epsilon}(\mathsf{tenil}) \qquad \frac{\vdash \Delta \qquad \alpha \notin \mathrm{dom}(\Delta)}{\vdash \Delta, \alpha{:}\hat{\kappa}}(\mathsf{tecons})$$

$$\frac{}{\Delta \vdash \epsilon}(\mathsf{qenil}) \qquad \frac{\Delta_1 \vdash \Psi \qquad \Delta_2 \vdash \tau : \kappa^{\mathrm{U}} \qquad \alpha{:}\kappa^{\mathrm{U}} \in \Delta_1 \qquad \alpha \notin \mathrm{dom}(\Psi)}{\Delta_1 * \Delta_2 \vdash \Psi, \alpha{:=}\tau}(\mathsf{qecons})$$

$$\frac{}{\Delta \vdash \epsilon}(\mathsf{eenil}) \qquad \frac{\Delta_1 \vdash \Gamma \qquad \Delta_2 \vdash \tau : \mathsf{type}^{\mathrm{U}} \qquad x \notin \mathrm{dom}(\Gamma)}{\Delta_1 * \Delta_2 \vdash \Gamma, x{:}\tau^{\iota}}(\mathsf{eecons})$$

$$\frac{\vdash \Delta \qquad \Delta \vdash \Psi \qquad \Delta \vdash \Gamma}{\vdash \Delta; \Psi; \Gamma}(\mathsf{env})$$

Mode Bounds: $\iota \preceq \iota \quad \hat{\kappa} \preceq \iota \quad \hat{\tau} \preceq \iota \quad \Delta \preceq \iota \quad \Gamma \preceq \iota \quad \Xi \preceq \iota$

$$\mathrm{U} \preceq \mathrm{L} \qquad \iota \preceq \iota \qquad \frac{\iota \preceq \iota'}{\kappa^{\iota} \preceq \iota'} \qquad \frac{\iota \preceq \iota'}{\tau^{\iota} \preceq \iota'}$$

$$\frac{}{\epsilon \preceq \iota} \qquad \frac{\Delta \preceq \iota \qquad \hat{\kappa} \preceq \iota}{\Delta, \alpha{:}\hat{\kappa} \preceq \iota} \qquad \frac{}{\epsilon \preceq \iota} \qquad \frac{\Gamma \preceq \iota \qquad \hat{\tau} \preceq \iota}{\Gamma, x{:}\hat{\tau} \preceq \iota} \qquad \frac{\Delta \preceq \iota \qquad \Gamma \preceq \iota}{\Delta; \Psi; \Gamma \preceq \iota}$$

Splitting: $\hat{\kappa}_1 * \hat{\kappa}_2 \quad \hat{\tau}_1 * \hat{\tau}_2 \quad \Delta_1 * \Delta_2 \quad \Gamma_1 * \Gamma_2 \quad \Xi_1 * \Xi_2$

$$\begin{array}{rcll}
\hat{\kappa}_1 * \hat{\kappa}_2 & = & \hat{\kappa}_2 * \hat{\kappa}_1 & \\
\kappa^{\mathrm{U}} * \kappa^{\mathrm{U}} & = & \kappa^{\mathrm{U}} & \\
\kappa^{\mathrm{L}} * \kappa^{\mathrm{U}} & = & \kappa^{\mathrm{L}} & \\
\\
\hat{\tau}_1 * \hat{\tau}_2 & = & \hat{\tau}_2 * \hat{\tau}_1 & \\
\tau^{\mathrm{U}} * \tau^{\mathrm{U}} & = & \tau^{\mathrm{U}} & \\
(?\tau)^{\mathrm{L}} * (?\tau)^{\mathrm{U}} & = & (?\tau)^{\mathrm{L}} & \\
\{\overline{\ell : \hat{\tau}_1}\}^{\mathrm{L}} * \{\overline{\ell : \hat{\tau}_2}\}^{\iota} & = & \{\overline{\ell : \hat{\tau}_1 * \hat{\tau}_2}\}^{\mathrm{L}} & \text{if } \overline{\hat{\tau}_2 \preceq \iota} \\
\\
\epsilon * \epsilon & = & \epsilon & \\
\Delta_1 * \Delta_2 & = & \Delta_2 * \Delta_1 & \\
\Delta_1, \alpha{:}\hat{\kappa}_1 * \Delta_2, \alpha{:}\hat{\kappa}_2 & = & (\Delta_1 * \Delta_2), \alpha{:}(\hat{\kappa}_1 * \hat{\kappa}_2) & \\
\\
\epsilon * \epsilon & = & \epsilon & \\
\Gamma_1 * \Gamma_2 & = & \Gamma_2 * \Gamma_1 & \\
\Gamma_1, x{:}\tau^{\mathrm{L}} * \Gamma_2 & = & (\Gamma_1 * \Gamma_2), x{:}\tau^{\mathrm{L}} & \text{if } x \notin \mathrm{dom}(\Gamma_2) \\
\Gamma_1, x{:}\hat{\tau}_1 * \Gamma_2, x{:}\hat{\tau}_2 & = & (\Gamma_1 * \Gamma_2), x{:}(\hat{\tau}_1 * \hat{\tau}_2) & \\
(\Delta_1; \Psi; \Gamma_1) * (\Delta_2; \Psi; \Gamma_2) & = & (\Delta_1 * \Delta_2); \Psi; (\Gamma_1 * \Gamma_2) & \\
\Xi * \Delta & = & \Xi * (\Delta; \Psi \text{ of } \Xi; \epsilon) &
\end{array}$$

Liberation: $\Delta^{\mathrm{U!}} \quad \Gamma^{\mathrm{U!}}$

$$\begin{array}{rcl}
\epsilon^{\mathrm{U!}} & = & \epsilon \\
(\Delta, \alpha{:}\kappa^{\iota})^{\mathrm{U!}} & = & \Delta^{\mathrm{U!}}, \alpha{:}\kappa^{\mathrm{U}}
\end{array} \qquad \begin{array}{rcl}
\epsilon^{\mathrm{U!}} & = & \epsilon \\
(\Gamma, x{:}(?\tau)^{\iota})^{\mathrm{U!}} & = & \Gamma^{\mathrm{U!}}, x{:}(?\tau)^{\mathrm{U}}
\end{array}$$

Fig. 14. LTG Static Semantics (Auxiliary Judgments and Definitions)

of type environments Δ, term environments Γ, and type equivalence environments Ψ, as defined in Figure 12. Instead of having separate environments for unrestricted and linear variables (as is common in many formulations of linear type systems), our system simply assigns moded types and kinds to the term and type variables in Γ and Δ, respectively. We say that an environment is unrestricted if all its variables have unrestricted mode. This is expressed by the notation $\Xi \preceq \mathrm{U}$ that is defined in Figure 14.

An environment can be *split* pointwise, written $\Xi_1 * \Xi_2$ according to the definition given in Figure 14. (Despite the name, the equations, when read left to right, actually define a deterministic *merging* operation. However, since inference rules are typically read backwards, which amounts to applying the definitions right to left, it can as well be interpreted as "splitting" the right-hand side.) Unrestricted variables will simply be copied (*i.e.,* an unrestricted environment can be freely copied). For a linear variable, the standard case is to put it in only one of the resulting environments. However, there are two exceptions: (1) a linear type name or a linear reference can be split into a linear and a non-linear copy, for reasons described below, and (2) a linear record can be split into two copies if each component can be split recursively. The latter allows forming records of linear references while still maintaining the ability to split them implicitly.

Linear references. When initially created with $\mathsf{new}\,\tau$, a reference is given linear type $(?\tau)^{\mathrm{L}}$ (rule enewe). This type represents the *capability* to define the associated value. Defining a linear reference x via $\mathsf{def}\,x{:=}e$ consumes the capability (rule edefe). Consequently, only one definition can take effect at runtime. Moreover, the capability *must* be consumed at some point, so a linear reference also represents an *obligation* to define its contents.

Reading (dereferencing, $!\,e$) can only be performed from an unrestricted reference (rule eget). How do we get one? By binding a linear one to a variable and splitting the resulting environment into a linear and an unrestricted copy of itself. While other linear type systems often provide explicit constructs for this purpose (*e.g.,* let! and friends [Wadler 1990]), this form of copying is entirely implicit in LTG. For example, the function $\lambda x{:}(?\mathsf{int})^{\mathrm{L}}.\,\mathsf{def}\,x{:=}5; !x$ is well-formed because, when type-checking its body, the environment $x{:}(?\mathsf{int})^{\mathrm{L}}$ can be split into $x{:}(?\mathsf{int})^{\mathrm{L}} * x{:}(?\mathsf{int})^{\mathrm{U}}$, such that the first occurrence of x in the function body has linear type, while the second is unrestricted.

Note that linear splitting does *not* ensure that a reference is only read after being defined—the function $\lambda x{:}(?\mathsf{int})^{\mathrm{L}}.\,!x; \mathsf{def}\,x{:=}5$ is well-typed in our system as well, but will raise a runtime error. As we explained in Section 3, this is a deliberate design choice.

The *contents* of a reference must always be unrestricted (which is why no explicit mode annotation appears on τ in $?\tau$) and may not consume any linear variable (rule edefe)—this is because a reference is non-strict and may be read multiple times. The side condition $\Xi \preceq \mathrm{U}$ in rule enewe ensures, as usual, that no linear variable from the environment can be ignored at the leaves of a typing derivation.

Type names and linear kinds. In a manner similar to references, $\mathsf{new}\,\alpha{:}\kappa\,\mathsf{in}\,e$ introduces a linear *type name*, which is locally bound as α in the environment (rule enewt). This involves a more esoteric feature of LTG: α is classified as an undefined abstract type by assigning it *linear kind* κ^{L}. Like a linear reference, a linear type name is ultimately consumed by defining it via def (rule edeft), and the environment splitting rules allow separating it into a linear and an unrestricted copy implicitly. The type system thereby ensures that all abstract types get defined, in a way almost entirely analogous to references. That is, undefined type names are linear capabilities as well. (This subsumes the use of effects for tracking type definitions in RTG.)

There is no explicit equivalent to "dereferencing" for a type name—a type name can simply be used as a type, provided it has unrestricted kind. Within the scope of

its definition a type name is always unrestricted: in rule edeft, the Δ_1 containing the linear α is split off from Ξ, so that α must be unrestricted in Ξ.

As an example, consider the following term:

$$\begin{array}{l}\mathsf{new}\ \alpha{:}\mathsf{type}\ \mathsf{in}\\ \mathsf{let}\ m = \mathsf{def}\ \alpha{:=}\mathsf{int}\ \mathsf{in}\ (\{\mathsf{f} = \lambda x{:}\alpha.x, \mathsf{v} = 77\} : \{\mathsf{f} : \alpha \to \mathsf{int}, \mathsf{v} : \alpha\})\\ \mathsf{in}\ m.\mathsf{f}\ ((\lambda x{:}\alpha.x)\ m.\mathsf{v})\end{array}$$

It introduces the type name α, scoping over the rest of the expression. However, α's definition is only known inside the r.h.s. of the binding for the "module" m, thereby effectively making α an ADT implemented by m. In the remainder of the term, α can freely be used (as witnessed by the little identity function), but its definition is not available. Hence, applying $m.\mathsf{f}$ to 33 would be ill-typed.

Note at this point that LTG only supports abstraction over plain types (with moded kind), not over moded types. Consequently, the witnesses for existential introduction and universal elimination are un-moded, and so are type constructor abstraction and application. Abstraction over moded types would be a natural extension to LTG, but we had no use for it in the context of MixML. (Arguably, that makes our linear kind system rather degenerate.)

The use of linear kinds and types facilitates a more elegant account of Dreyer's "destination-passing style (DPS) universal" types than was possible in the original RTG system. DPS universal types—which, as we shall see, are useful in modeling separate compilation of recursive modules—have the form $\forall\alpha{\uparrow}\kappa.\,\tau_1 \to \tau_2$; such a type describes a function that takes as arguments an undefined abstract type name α and a value of type τ_1 (which may mention α), and returns a value of type τ_2, while defining *en passant* the name α. In RTG, the parameterization over the type name α and the value of type τ_1 had to be hard-wired together, because it was necessary to ensure that the function returned after instantiating α was called exactly once, but the type system did not build in support for reasoning about linearity. Here, we can use linear kinds and types to encode the DPS universal type $\forall\alpha{\uparrow}\kappa.\,\tau_1 \to \tau_2$ as a composition of the existing universal and arrow type constructors: $\forall\alpha{:}\kappa^{\mathrm{L}}.\,(\tau_1^{\mathrm{U}} \to \tau_2^{\mathrm{U}})^{\mathrm{L}}$. The linear kind ascribed to α means that it is treated as a type name that must be defined, and the linear mode on the arrow type ensures that the function defining α must be applied exactly once by the program context.

Other terms. Given linear references, linearity is lifted to other types in standard ways. For example, the type system ensures that a record of linear type $\{\ell_1 : \tau_1^{\mathrm{U}}, \ell_2 : \tau_2^{\mathrm{L}}\}^{\mathrm{L}}$ will (eventually) be deconstructed by the program context, as will its linear ℓ_2 component. The unrestricted ℓ_1 component, however, may be ignored by the context. In a similar manner, a function of type $(\tau_1^{\mathrm{L}} \to \tau_2^{\mathrm{L}})^{\mathrm{L}}$ must be applied exactly once; it will consume its argument, and return a result value that must then be consumed by the context.

Most of the rules for terms closely follow Ahmed et al. [2005]. As should be expected, the central invariant is that a term is only well-formed under a given environment Ξ if it consumes all linear variables (term and type variables in our case) bound in Ξ. In particular, the variable rule has to require that all of the environment except for the variable binding in question is unrestricted (rule evar).

Functions (rule efun) have to be linear whenever their body consumes a linear variable from the environment: the side condition $\Xi \preceq \iota$ ensures that $\iota = \mathrm{U}$ only if Ξ is unrestricted, and otherwise, the type system will ensure that the function gets applied by the surrounding program context. Function application (rule eapp) involves two expressions. Therefore, the environment has to be split such that each linear capability is given to either e_1 or e_2. An unusual aspect of our system is that splitting is also

used in the function rule, where we need a suitable (unrestricted) type environment Δ under which the type annotation is well-formed (as described below).

Records require iterated splitting (rule erec). The side conditions $\hat{\tau}_i \preceq \iota$ ensure that the record is considered linear ($\iota = \text{L}$) if at least one of its components is linear; the extra Ξ_0 is needed to handle the empty case. To deconstruct a record, pattern matching has to be used (rule eprj). Figure 12 defines the familiar dot notation for projection as a derived form. However, this notation will only desugar to a well-typed term if all unused components of the record are unrestricted.

The treatment of linear kinds in LTG's type system is very similar to that of linear types. The rules for polymorphic functions (egen and einst) and existential packages (epack and eopen) hence mirror those of ordinary functions and records, except that they involve (potentially linear) *type* expressions, not just term expressions. Accordingly, they introduce and eliminate *type* variables of possibly linear *kind*.

Type well-formedness. The kinding judgment $\Delta \vdash \tau : \hat{\kappa}$ checks well-formedness of types. Most rules are standard, except for the additional mode annotation. All but rule tvar involve only unrestricted mode. Types of linear kind can only be introduced and consumed on the term level—only new expressions, term-level type lambdas, and unpacking for existentials can bind linear type variables. Consequently, the only possible types of linear kind are type variables. As with term variables, the variable rule has to require that the remaining environment is unrestricted.

Perhaps surprisingly, our kinding rules do not employ splitting. Splitting is not necessary because we can show that Δ will always be unrestricted in any derivation except one that only consists of the variable rule (cf. Lemma 7.2 below).

Environments. Figure 14 defines well-formed environments. For type and term environments the rules are standard, except for the additional requirement that types classifying term variables in Γ obviously need to have unrestricted kind type^{U} (rule eecons).

An equivalence environment Ψ is only deemed well-formed if (1) all variables it defines are bound in the type environment with unrestricted kind, (2) their definitions are well-formed with unrestricted kind, and (3) no variable is defined twice (rule qecons). It is deliberately not required that type definitions in Ψ are acyclic. Finally, a combined environment $\Xi = \Delta; \Psi; \Gamma$ is well-formed if its individual components are.

Environment splitting. The definition of environment splitting implies that type variables always have to be kept on both sides, even if they are linear (albeit with a possible mode change). This is no real limitation, but yields the following useful property about well-formed environments, which allows us to derive the well-formedness of the environments used at the leaves of a derivation from those appearing at its root:

LEMMA 7.1 (ENVIRONMENT SPLITTING).

(*1*) *Let* $\Delta = \Delta_1 * \Delta_2$. *Then,* $\vdash \Delta$ *if and only if* $\vdash \Delta_1$ *and* $\vdash \Delta_2$.
(*2*) *Let* $\Gamma = \Gamma_1 * \Gamma_2$. *Then,* $\Delta \vdash \Gamma$ *if and only if* $\Delta \vdash \Gamma_1$ *and* $\Delta \vdash \Gamma_2$.
(*3*) *Let* $\Xi = \Xi_1 * \Xi_2$. *Then,* $\vdash \Xi$ *if and only if* $\vdash \Xi_1$ *and* $\vdash \Xi_2$.

In particular, the last part would not hold if we were to allow parts of the Δ component of either Ξ_1 or Ξ_2 to be dropped—the respective type variables might occur free in the same side's Γ.

We can show that most derivations with unrestricted mode on the right-hand side require an unrestricted environment:

LEMMA 7.2 (UNRESTRICTED DERIVATIONS).

(*1*) *If* $\Delta \vdash \tau : \kappa^{\text{U}}$, *then* $\Delta \preceq \text{U}$.

$$
\begin{array}{lll}
\text{Type stores} & \sigma & ::= \epsilon \mid \sigma, \alpha{:}?\kappa \mid \sigma, \alpha{:}{=}\tau{:}\kappa \\
\text{Value stores} & s & ::= \epsilon \mid s, x{:}?\tau \mid s, x{:}{=}e{:}\tau \\
\text{Configurations} & \xi & ::= \sigma; s; e \mid \bullet \\
\text{Contexts} & E & ::= [] \mid E\ e \mid v\ E \mid \{\overline{\ell{=}v}, \ell{=}E, \overline{\ell{=}e}\} \mid \mathsf{let}\ \{\overline{\ell{=}x}\}{=}E\ \mathsf{in}\ e \mid \mathsf{def}\ E{:}{=}e \mid\ !E \\
& & \quad \mid E\ \tau \quad \mid \langle \tau, E \rangle_{\tau'} \quad \mid \mathsf{let}\ \langle \alpha, x \rangle{=}E\ \mathsf{in}\ e
\end{array}
$$

Reduction: $\xi \hookrightarrow \xi'$

$$
\begin{array}{ll}
\sigma; s; E[(\lambda x{:}\hat{\tau}.e)\ v] & \hookrightarrow \sigma; s; E[e[v/x]] \\
\sigma; s; E[\mathsf{let}\ \{\overline{\ell{=}x}\}{=}\{\overline{\ell{=}v}\}\ \mathsf{in}\ e] & \hookrightarrow \sigma; s; E[e[\overline{v/x}]] \\
\sigma; s; E[(\lambda \alpha{:}\hat{\kappa}.e)\ \tau] & \hookrightarrow \sigma; s; E[e[\tau/\alpha]] \\
\sigma; s; E[\mathsf{let}\ \langle \alpha, x \rangle{=}\langle \tau, v \rangle_{\tau'}\ \mathsf{in}\ e] & \hookrightarrow \sigma; s; E[e[\tau/\alpha][v/x]] \\
\sigma; s; E[\mathsf{new}\ \tau] & \hookrightarrow \sigma; s, x{:}?\tau; E[x] \\
\sigma; s_1, x{:}?\tau, s_2; E[\mathsf{def}\ x{:}{=}e] & \hookrightarrow \sigma; s_1, x{:}{=}e{:}\tau, s_2; E[\{\}] \\
\sigma; s_1, x{:}{=}e{:}\tau, s_2; E[!x] & \hookrightarrow \sigma; s_1, x{:}{=}e{:}\tau, s_2; E[e] \\
\sigma; s_1, x{:}?\tau, s_2; E[!x] & \hookrightarrow \bullet \\
\sigma; s; E[\mathsf{new}\ \alpha{:}\kappa\ \mathsf{in}\ e] & \hookrightarrow \sigma, \alpha{:}?\kappa; s; E[e] \\
\sigma_1, \alpha{:}?\kappa, \sigma_2; s; E[\mathsf{def}\ \alpha{:}{=}\tau\ \mathsf{in}\ e] & \hookrightarrow \sigma_1, \alpha{:}{=}\tau{:}\kappa, \sigma_2; s; E[e]
\end{array}
$$

Config and Store Typing: $\vdash \xi : \tau \quad \vdash \sigma : \Delta; \Psi \quad \Delta; \Psi \vdash s : \Gamma \quad \Delta' \vdash \sigma : \Delta; \Psi \quad \Xi \vdash s : \Gamma$

$$\frac{\vdash \sigma : \Delta; \Psi \qquad \Delta^{\mathrm{U}!}; \Psi \vdash s : \Gamma \qquad \Delta; \Psi; \Gamma \vdash e : \tau^{\mathrm{U}} \qquad \epsilon \vdash \tau : \mathsf{type}^{\mathrm{U}}}{\vdash \sigma; s; e : \tau} \qquad \frac{\epsilon \vdash \tau : \mathsf{type}^{\mathrm{U}}}{\vdash \bullet : \tau}$$

$$\frac{\vdash \Delta^{\mathrm{U}!} \qquad \Delta^{\mathrm{U}!} \vdash \sigma : \Delta; \Psi}{\vdash \sigma : \Delta; \Psi} \qquad \frac{\Delta \vdash \Gamma^{\mathrm{U}!} \qquad \Delta; \Psi; \Gamma^{\mathrm{U}!} \vdash s : \Gamma}{\Delta; \Psi \vdash s : \Gamma}$$

$$\frac{\Delta' \preceq \mathrm{U}}{\Delta' \vdash \epsilon : \epsilon; \epsilon} \qquad \frac{\Delta' \vdash \sigma : \Delta; \Psi}{\Delta' \vdash \sigma, \alpha{:}?\kappa : \Delta, \alpha{:}\kappa^{\mathrm{L}}; \Psi} \qquad \frac{\Delta' \vdash \sigma : \Delta; \Psi \qquad \Delta' \vdash \tau : \kappa^{\mathrm{U}}}{\Delta' \vdash \sigma, \alpha{:}{=}\tau{:}\kappa : \Delta, \alpha{:}\kappa^{\mathrm{U}}; \Psi, \alpha{:}{=}\tau}$$

$$\frac{\Xi \preceq \mathrm{U}}{\Xi \vdash \epsilon : \epsilon} \qquad \frac{\Xi \vdash s : \Gamma \qquad \Delta \vdash \tau : \mathsf{type}^{\mathrm{U}}}{\Xi * \Delta \vdash s, x{:}?\tau : \Gamma, x{:}(?\tau)^{\mathrm{L}}} \qquad \frac{\Xi_1 \vdash s : \Gamma \qquad \Xi_2 \vdash e : \tau^{\mathrm{U}} \qquad \Xi_2 \preceq \mathrm{U}}{\Xi_1 * \Xi_2 \vdash s, x{:}{=}e{:}\tau : \Gamma, x{:}(?\tau)^{\mathrm{U}}}$$

Context Typing: $\Xi \vdash E : \hat{\tau} \Rightarrow \tau$

$$\frac{\Xi, x{:}\hat{\tau} \vdash E[x] : \tau^{\mathrm{U}}}{\Xi \vdash E : \hat{\tau} \Rightarrow \tau}$$

Fig. 15. LTG Dynamic Semantics

(*2*) *If* $\Xi \vdash v : \tau^{\mathrm{U}}$*, then* $\Xi \preceq \mathrm{U}$.

Note that part (2) of the lemma only holds for values. In general, expressions may involve eliminations of linear variables while still yielding an unrestricted result (for example, consider $f{:}(\{\}^{\mathrm{U}} \to \{\}^{\mathrm{U}})^{\mathrm{L}} \vdash f\ \{\} : \{\}^{\mathrm{U}}$). Fortunately, we only care about this property for values (see below).

7.3. LTG Operational Semantics

Figure 15 gives the dynamic semantics of LTG. To avoid clutter, we implicitly assume that x is fresh with respect to s whenever we write $s, x{:}?\tau$ in this figure, and likewise for all similar extensions of stores or environments.

Substitution Typing: $\Delta' \vdash \delta : \Delta \quad \Psi' \vdash \delta : \Psi \quad \Delta'; \Psi' \vdash \delta : \Delta; \Psi \quad \Xi \vdash \gamma : \Gamma$

$$\frac{\Delta' \preceq \mathrm{U}}{\Delta' \cup \Delta \vdash \epsilon : \Delta} \qquad \frac{\Delta'_1 \vdash \delta : \Delta \qquad \Delta'_2 \vdash \tau : \hat{\kappa} \qquad \alpha \notin \mathrm{dom}(\delta)}{\Delta'_1 * \Delta'_2 \vdash \delta, \alpha{\mapsto}\tau : \Delta, \alpha{:}\hat{\kappa}}$$

$$\frac{}{\Psi' \vdash \delta : \epsilon} \qquad \frac{\Psi' \vdash \delta : \Psi \qquad \Psi' \vdash \delta(\alpha) \equiv \delta(\tau) \qquad \alpha \notin \mathrm{dom}(\Psi)}{\Psi' \vdash \delta : \Psi, \alpha{:=}\tau}$$

$$\frac{\Delta' \vdash \delta : \Delta \qquad \Psi' \vdash \delta : \Psi}{\Delta'; \Psi' \vdash \delta : \Delta; \Psi}$$

$$\frac{\Xi \preceq \mathrm{U}}{\Xi \cup \Gamma \vdash \epsilon : \Gamma} \qquad \frac{\Xi_1 \vdash \gamma : \Gamma \qquad \Xi_2 \vdash v : \hat{\tau} \qquad x \notin \mathrm{dom}(\gamma)}{\Xi_1 * \Xi_2 \vdash \gamma, x{\mapsto}v : \Gamma, x{:}\hat{\tau}}$$

Substitution Splitting: $\gamma_1 * \gamma_2$

$$\begin{array}{rcll} \gamma_1, x{\mapsto}v * \gamma_2 & = & (\gamma_1 * \gamma_2), x{\mapsto}v & \text{if } x \notin \mathrm{dom}(\gamma_2) \\ \gamma_1, x{\mapsto}v * \gamma_2, x{\mapsto}v & = & (\gamma_1 * \gamma_2), x{\mapsto}v & \\ \gamma_1 * \gamma_2 & = & \gamma_2 * \gamma_1 & \end{array}$$

Fig. 16. LTG Substitutions

Configurations and Reduction. Reduction is defined over configurations ξ, which consist of an expression e to evaluate and a two-part store: the *type store* σ records allocated type names and the *value store* s references. In either store, an allocated variable can be in one of two states: undefined ($\alpha{:}?\kappa$ and $x{:}?\tau$) or defined ($\alpha{:=}\tau{:}\kappa$ and $x{:=}e{:}\tau$). Because references are non-strict, their definition in the value store can actually be an expression e instead of just a value. An exceptional configuration is the error state $\bullet$ (*black hole*). It indicates the erroneous attempt to access an as-yet-undefined reference.

The reduction rules define a mostly straightforward call-by-value semantics. The only interesting operation is reading of references ($!x$), where we have two possible cases: either x is defined, in which case we simply return its definition e, or x is undefined, in which case $!x$ incurs a runtime error, which we indicate by $\bullet$.

Typing of Stores, Configurations, and Substitutions. Figure 15 also defines well-formedness judgments for the various entities in the dynamic semantics (in particular, stores and configurations), while Figure 16 gives the rules for substitutions. Typing of stores is mostly straightforward, but has to take into account that both type and value store can be recursive. The definition of each type or value in the respective store has to be unrestricted and not use any linear resources.

A configuration is typed according to the type of its computation e under an environment derived from the store. A configuration is only well-formed if e has unrestricted type, so that it cannot result in an unconsumed linear value. Moreover, if e already is a value then Lemma 7.2 implies almost immediately that there are no undefined variables left in the store.

Some typing rules make use of the "liberation" meta-operator $(\_)^{\mathrm{U}!}$, defined in Figure 14, for making environments unrestricted. This operator merely provides a convenient (and deterministic) means of splitting off an unrestricted copy from a given environment. More precisely, it satisfies the following property:

LEMMA 7.3 (LIBERATION).

(*1*) $\Delta * \Delta^{\mathrm{U!}} = \Delta$.
(*2*) $\Gamma * \Gamma^{\mathrm{U!}} = \Gamma$.

7.4. LTG Type Soundness

The structure of the type soundness proof is mostly standard, but requires extra work in proving the substitution property, because of environment splitting. The central lemmas and theorems are as follows.

The first two lemmas state frequently needed, standard structural properties:

LEMMA 7.4 (VARIABLE CONTAINMENT).

(*1*) *If* $\Delta \vdash \tau : \hat{\kappa}$, *then* $\mathrm{fv}(\tau) \subseteq \mathrm{dom}(\Delta)$.
(*2*) *If* $\Xi \vdash e : \hat{\tau}$, *then* $\mathrm{fv}(e) \subseteq \mathrm{dom}(\Xi)$.

LEMMA 7.5 (WEAKENING).

(*1*) *If* $\Delta_1, \Delta_2 \vdash \tau : \hat{\kappa}$ *and* $\alpha \notin \mathrm{dom}(\Delta_1, \Delta_2)$, *then* $\Delta_1, \alpha{:}\kappa^{\mathrm{U}}, \Delta_2 \vdash \tau : \hat{\kappa}$.
(*2*) *If* $\Psi_1, \Psi_2 \vdash \tau_1 \equiv \tau_2$ *and* $\alpha \notin \mathrm{dom}(\Psi_1, \Psi_2)$, *then* $\Psi_1, \alpha{:=}\tau, \Psi_2 \vdash \tau_1 \equiv \tau_2$.
(*3*) *If* $\Xi_1, \Xi_2 \vdash e : \hat{\tau}$ *and* $\alpha \notin \mathrm{dom}(\Delta \text{ of } \Xi_1, \Xi_2)$, *then* $\Xi_1, \alpha{:}\kappa^{\mathrm{U}}, \Xi_2 \vdash e : \hat{\tau}$.
(*4*) *If* $\Xi_1, \Xi_2 \vdash e : \hat{\tau}$ *and* $\alpha \notin \mathrm{dom}(\Psi \text{ of } \Xi_1, \Xi_2)$, *then* $\Xi_1, \alpha{:=}\tau, \Xi_2 \vdash e : \hat{\tau}$.
(*5*) *If* $\Xi_1, \Xi_2 \vdash e : \hat{\tau}$ *and* $x \notin \mathrm{dom}(\Gamma \text{ of } \Xi_1, \Xi_2)$, *then* $\Xi_1, x{:}\tau^{\mathrm{U}}, \Xi_2 \vdash e : \hat{\tau}$.

In the remaining statements we implicitly assume that all environments occurring left of a precondition's turnstile "$\vdash$" are well-formed.

A key property in dealing with substitutions and linearity is the following. It states that any substitution that can be assigned a split environment can itself be split into two respective substitutions. It is a prerequisite for proving the substitution lemma stated below. (Note how our definition of splitting for Δ, that always copies all variables, is essential for the first part.)

LEMMA 7.6 (SUBSTITUTION SPLITTING).

(*1*) *If* $\Delta' \vdash \delta : \Delta_1 * \Delta_2$, *then* $\Delta' = \Delta'_1 * \Delta'_2$ *with* $\Delta'_1 \vdash \delta : \Delta_1$ *and* $\Delta'_2 \vdash \delta : \Delta_2$.
(*2*) *If* $\Xi \vdash \gamma : \Gamma_1 * \Gamma_2$, *then* $\Xi = \Xi_1 * \Xi_2$ *and* $\gamma = \gamma_1 * \gamma_2$ *with* $\Xi_1 \vdash \gamma_1 : \Gamma_1$ *and* $\Xi_2 \vdash \gamma_2 : \Gamma_2$.

PROOF. By induction on the derivation and inspection of the cases for the split. □

LEMMA 7.7 (SUBSTITUTION).

(*1*) *If* $\Delta \vdash \tau : \hat{\kappa}$ *and* $\Delta' \vdash \delta : \Delta$, *then* $\Delta' \vdash \delta(\tau) : \hat{\kappa}$.
(*2*) *If* $\Delta \vdash \Gamma$ *and* $\Delta' \vdash \delta : \Delta$, *then* $\Delta' \vdash \delta(\Gamma)$.
(*3*) *If* $\Psi \vdash \tau_1 \equiv \tau_2$ *and* $\Psi' \vdash \delta : \Psi$, *then* $\Psi' \vdash \delta(\tau_1) \equiv \delta(\tau_2)$.
(*4*) *If* $\Delta; \Psi; \Gamma \vdash e : \hat{\tau}$ *and* $\Delta'; \Psi' \vdash \delta : \Delta; \Psi$, *then* $\Delta'; \Psi'; \delta(\Gamma) \vdash \delta(e) : \delta(\hat{\tau})$.
(*5*) *If* $\Delta; \Psi; \Gamma \vdash e : \hat{\tau}$ *and* $\Delta'; \Psi; \Gamma' \vdash \gamma : \Gamma$ *with* $\Delta'' = \Delta * \Delta'$, *then* $\Delta''; \Psi; \Gamma' \vdash \gamma(e) : \hat{\tau}$.

Note that in part (5) of the previous lemma $\Delta'' = \Delta * \Delta'$ is a precondition, *i.e.,* the lemma only holds if Δ and Δ' can be merged.

The following lemma is required for proving preservation. It allows us to break up a well-typed term into a well-typed context and a well-typed subterm placed in this context (and inversely, reassemble them).

LEMMA 7.8 (CONTEXT COMPOSITION). *Suppose* $x \notin \mathrm{fv}(E)$.

(*1*) *If and only if* $\Xi \vdash E[e] : \hat{\tau}$, *then* $\Xi_1, x{:}\hat{\tau}' \vdash E[x] : \hat{\tau}$ *and* $\Xi_2 \vdash e : \hat{\tau}'$ *with* $\Xi = \Xi_1 * \Xi_2$.
(*2*) *If and only if* $\Xi \vdash E[e] : \tau^{\mathrm{U}}$, *then* $\Xi_1 \vdash E : \hat{\tau} \Rightarrow \tau$ *and* $\Xi_2 \vdash e : \hat{\tau}$ *with* $\Xi = \Xi_1 * \Xi_2$.

PROOF. The second part of this lemma is a simple corollary of the first, which is proved by induction on the corresponding derivations. □

Since the LTG type system contains a conversion rule (econv), we also need the usual generalized inversion lemma modulo type equivalence:

LEMMA 7.9 (INVERSION). *Suppose* $\Xi \vdash e : \tau^\iota$.

(*1*) *If* $e = x$, *then* $\Xi = \Xi_1, x{:}(\tau')^\iota, \Xi_2$ *and* Ψ of $\Xi \vdash \tau \equiv \tau'$, *with* $\Xi_1, \Xi_2 \preceq \mathrm{U}$.
(*2*) *If* $e = \lambda x{:}\tau_1^{\iota_1}.e_2$, *then* Ψ of $\Xi \vdash \tau \equiv \tau_1^{\iota_1} \to \hat\tau_2$ *and* $\Xi = \Delta_1 * \Xi_2$, *with* $\Delta_1 \vdash \tau_1 : \mathsf{type}^{\mathrm{U}}$ *and* $\Xi_2, x{:}\tau_1^{\iota_1} \vdash e_2 : \hat\tau_2$ *and* $\Xi_2 \preceq \iota$.
(*3*) *If* $e = e_1\, e_2$, *then* Ψ of $\Xi \vdash \tau \equiv \tau'$ *and* $\Xi = \Xi_1 * \Xi_2$, *with* $\Xi_1 \vdash e_1 : (\hat\tau_2 \to (\tau')^\iota)^{\iota_1}$ *and* $\Xi_2 \vdash e_2 : \hat\tau_2$.
(*4*) *... similarly for all other constructs.*

With these preparations in hand, we can prove the standard preservation property, which is the first half of soundness:

THEOREM 7.10 (PRESERVATION). *If* $\xi \hookrightarrow \xi'$ *and* $\vdash \xi : \tau$, *then* $\vdash \xi' : \tau$.

PROOF. By the previous lemmas and analysis of the cases for $\xi \hookrightarrow \xi'$. The proof also relies on consistency of type equivalence, which we prove separately in Section 7.5. We show only the first case, the others are proved in similar ways:

Case $\sigma; s; E[(\lambda x{:}\tau^\iota.e)\, v] \hookrightarrow \sigma; s; E[e[v/x]]$:
— by inversion of config typing, $\vdash \sigma : \Delta; \Psi$ and $\Delta^{\mathrm{U}!}; \Psi \vdash s : \Gamma$ and $\Xi_0 \vdash E[(\lambda x{:}\tau^\iota.e)\, v] : \tau_0^{\mathrm{U}}$ for some $\Xi_0 = \Delta; \Psi; \Gamma$ and $\epsilon \vdash \tau_0 : \mathsf{type}^{\mathrm{U}}$
— by Lemma 7.8, $\Xi' \vdash E : \tau_1^{\iota_1} \Rightarrow \tau_0$ and $\Xi \vdash (\lambda x{:}\tau^\iota.e)\, v : \tau_1^{\iota_1}$ with $\Xi_0 = \Xi' * \Xi$
— by Lemma 7.9, $\Xi_1 \vdash \lambda x{:}\tau^\iota.e : (\tau_2^{\iota_2} \to \tau_1'^{\iota_1})^{\iota_0}$ and $\Xi_2 \vdash v : \tau_2^{\iota_2}$, with $\Xi = \Xi_1 * \Xi_2$ and Ψ of $\Xi \vdash \tau_1 \equiv \tau_1'$
— by Lemma 7.9, $\Delta_{11} \vdash \tau : \mathsf{type}^{\mathrm{U}}$ and $\Xi_{12}, x{:}\tau^\iota \vdash e : \tau_3^{\iota_3}$ with $\Xi_1 = \Delta_{11} * \Xi_{12}$ and Ψ of $\Xi_1 \vdash \tau_2^{\iota_2} \to \tau_1'^{\iota_1} \equiv \tau^\iota \to \tau_3^{\iota_3}$
— by Lemma 7.2, $\Delta_{11} \preceq \mathrm{U}$, and thus $\Delta_{11} * \Xi_{12} = \Xi_{12}$, that is, $\Xi_1 = \Xi_{12}$
— by Consistency (Corollary 7.23), Ψ of $\Xi_1 \vdash \tau_2 \equiv \tau$ and and $\iota_2 = \iota$ and Ψ of $\Xi_1 \vdash \tau_1' \equiv \tau_3$ and $\iota_1 = \iota_3$
— by the definition of $*$, Ψ of $\Xi_1 = \Psi$ of $\Xi = \Psi$ of Ξ_2
— hence, Ψ of $\Xi_2 \vdash \tau_2 \equiv \tau$ and Ψ of $\Xi \vdash \tau_1' \equiv \tau_3$, and by transitivity, Ψ of $\Xi \vdash \tau_1 \equiv \tau_3$
— by rule econv, $\Xi_2 \vdash v : \tau^\iota$
— thus, $\Xi \vdash [v/x] : (\Gamma \text{ of } \Xi), x{:}\tau^\iota$, and by Lemma 7.7, $\Xi \vdash e[v/x] : \tau_3^{\iota_3}$
— by rule econv, $\Xi \vdash e[v/x] : \tau_1^{\iota_1}$
— by Lemma 7.8, $\Xi_0 \vdash E[e[v/x]] : \tau_0^{\mathrm{U}}$, and by config typing, $\sigma; s; E[e[v/x]] : \tau_0$ □

To prove progress, the other half of soundness, we rely on the usual lemmas about canonical values. However, we also need similar lemmas about the stores. In particular, they tell us that in well-formed stores, a linear (term or type) variable is yet undefined, while an unrestricted one is always defined.

LEMMA 7.11 (CANONICAL VALUES). *Let* $\vdash \Delta; \Psi; \Gamma$ *and* $\Delta^{\mathrm{U}!}; \Psi \vdash s : \Gamma$, i.e., Γ *is a store typing. Assume* $\Delta; \Psi; \Gamma \vdash v : \tau^\iota$.

(*1*) *If* $\Psi \vdash \tau \equiv \hat\tau_1 \to \hat\tau_2$, *then* $v = \lambda x{:}\hat\tau_1'.e$.
(*2*) *If* $\Psi \vdash \tau \equiv \{\overline{\ell : \hat\tau}\}$, *then* $v = \{\overline{\ell = v'}\}$.
(*3*) *If* $\Psi \vdash \tau \equiv \forall \alpha{:}\hat\kappa.\hat\tau$, *then* $v = \lambda \alpha{:}\hat\kappa.e$.
(*4*) *If* $\Psi \vdash \tau \equiv \exists \alpha{:}\hat\kappa.\hat\tau$, *then* $v = \langle \tau_1, v' \rangle_{\exists \alpha:\hat\kappa.\hat\tau'}$.

(5) *If* $\Psi \vdash \tau \equiv ?\tau_1$*, then* $v = x$.

PROOF. By induction on the typing derivation for v. The case analysis again relies on consistency of type equivalence, which we discuss in Section 7.5 below. □

LEMMA 7.12 (CANONICAL VALUE STORES). *Let* $\Delta; \Psi \vdash s : \Gamma$ *and* $\Gamma = \Gamma_1, x{:}\tau^\iota, \Gamma_2$.

(1) *If* $\iota = \text{L}$*, then* $s = s_1, x{:}?\tau', s_2$.
(2) *If* $\iota = \text{U}$*, then* $s = s_1, x{:=}e{:}\tau', s_2$.

LEMMA 7.13 (CANONICAL TYPE STORES). *Let* $\vdash \sigma : \Delta; \Psi$ *and* $\Delta = \Delta_1, \alpha{:}\kappa^\iota, \Delta_2$.

(1) *If* $\iota = \text{L}$*, then* $\sigma = \sigma_1, \alpha{:}?\kappa', \sigma_2$.
(2) *If* $\iota = \text{U}$*, then* $\sigma = \sigma_1, \alpha{:=}\tau{:}\kappa', \sigma_2$.

PROOF. (Both previous lemmas) By induction on the respective derivation. □

We now have the relevant ingredients for proving progress. Besides the usual property, the theorem also says that whenever a configuration ξ is terminal, it contains only defined variables in the stores. That is the core soundness property for linear modes.

THEOREM 7.14 (PROGRESS).
Let $\vdash \xi : \tau'$ *and* $\xi \neq \bullet$*. Then either* $\xi = (\overline{\alpha{:=}\tau{:}\kappa}; \overline{x{:=}e{:}\tau}; v)$*, or* $\xi \hookrightarrow \xi'$.

PROOF. By inversion, $\vdash \sigma : \Delta; \Psi$ and $\Delta^{\text{U}!}; \Psi \vdash s : \Gamma$ and $\Delta; \Psi; \Gamma \vdash e : \tau^{\text{U}}$, where $\xi = \sigma; s; e$. If e is a value, then the conclusion follows easily from Lemma 7.2 and iterating the previous two lemmas. Otherwise, weaken the typing assumption on e to $\Xi_1 \vdash e' : \hat{\tau}$ with $e = E[e']$ and $\Delta; \Psi; \Gamma = \Xi_1 * \Xi_2$. The result follows by induction on the typing derivation, generalizing E, e', $\hat{\tau}$, Ξ_1, and Ξ_2. □

Finally, we have the following property, which states that instead of applying a type substitution to a derivation, we can also turn the substitution into an explicit type equivalence environment. This property is not needed to show soundness of the calculus, but we will need it later to prove correctness of the MixML elaboration.

LEMMA 7.15 (SUBSTITUTION REVERSAL). *Let* $\Delta; \Psi \vdash \delta : \Delta'; \Psi'$ *with* $\mathrm{dom}(\delta) \subseteq \mathrm{dom}(\Delta') \cup \mathrm{dom}(\Psi')$ *and* $\Delta = \Delta' - \mathrm{dom}(\delta)$ *and* $\Psi = \Psi' - \mathrm{dom}(\delta)$.

(1) $\Psi' \vdash \tau \equiv \delta\tau$.
(2) *If* $\Psi \vdash \delta\tau_1 \equiv \delta\tau_2$*, then* $\Psi' \vdash \tau_1 \equiv \tau_2$.
(3) *If* $\Delta \vdash \delta\tau : \hat{\kappa}$*, then* $\Delta' \vdash \tau : \hat{\kappa}$.
(4) *If* $\Delta; \Psi; \delta\Gamma \vdash \delta e : \delta\hat{\tau}$*, then* $\Delta'; \Psi'; \Gamma \vdash e : \hat{\tau}$.

PROOF. The first part is by easy induction on the structure of τ. The second part is by induction on the derivation, where we first distinguish the cases $\tau_1 \in \mathrm{dom}(\delta)$ and $\tau_1 \notin \mathrm{dom}(\delta)$, and directly use Part (1) and transitivity of type equivalence in the former. The remaining parts then follow easily. □

7.5. Consistency of Type Equivalence

Consistency of type equivalence is an essential property: it ensures that no two types formed from different base type constructors (*e.g.,* $\forall$ and $\to$) can ever be deemed equivalent by the type system, and moreover that whenever two base types of the same shape are equivalent (*e.g.,* $\tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'$), their constituent types are correspondingly equivalent as well (*e.g.,* $\tau_1 \equiv \tau_1'$ and $\tau_2 \equiv \tau_2'$). In particular, this property is necessary to

Type paths $\rho ::= \alpha \mid \rho\ \tau \mid \forall\alpha{:}\hat{\kappa}.\hat{\tau} \qquad (\alpha \notin \mathrm{dom}(\Psi))$

Logical Approximation of Types: $\vdash \tau_1 \preccurlyeq_n \tau_2 : \kappa$

$$\frac{\vdash \tau_1 \equiv \tau_2 : \kappa' \to \kappa'' \qquad \forall \tau_1', \tau_2', j \leq n.\ (\vdash \tau_1' \preccurlyeq_j \tau_2' : \kappa') \Longrightarrow (\vdash \tau_1\ \tau_1' \preccurlyeq_j \tau_2\ \tau_2' : \kappa'')}{\vdash \tau_1 \preccurlyeq_n \tau_2 : \kappa' \to \kappa''}$$

$$\frac{\vdash \tau_1 \equiv \tau_2 : \mathsf{type} \qquad \forall j \leq n.\ (\vdash \tau_1 \xrightarrow{\mathrm{hd}}{}^{j}\ \rho_1) \Longrightarrow (\vdash \tau_2 \xrightarrow{\mathrm{hd}}{}^{*}\ \rho_2\ \wedge\ \vdash \rho_1 \leftrightarrow \rho_2 : \mathsf{type})}{\vdash \tau_1 \preccurlyeq_n \tau_2 : \mathsf{type}}$$

Path Equivalence: $\vdash \rho_1 \leftrightarrow \rho_2 : \kappa$

$$\frac{\alpha \notin \mathrm{dom}(\Psi)}{\vdash \alpha \leftrightarrow \alpha : \kappa_\alpha} \qquad \frac{\vdash \rho_1 \leftrightarrow \rho_2 : \kappa' \to \kappa \qquad \vdash \tau_1 \equiv \tau_2}{\vdash \rho_1\ \tau_1 \leftrightarrow \rho_2\ \tau_2 : \kappa} \qquad \frac{\vdash \tau_1 \equiv \tau_2}{\vdash \forall\alpha{:}\hat{\kappa}.\tau_1^{\iota} \leftrightarrow \forall\alpha{:}\hat{\kappa}.\tau_2^{\iota} : \mathsf{type}}$$

Head Reduction: $\vdash \tau_1 \xrightarrow{\mathrm{hd}} \tau_2$

$$\frac{\alpha \in \mathrm{dom}(\Psi)}{\vdash \alpha \xrightarrow{\mathrm{hd}} \Psi(\alpha)} \qquad \frac{}{\vdash (\lambda\alpha.\tau)\ \tau' \xrightarrow{\mathrm{hd}} \tau[\tau'/\alpha]} \qquad \frac{\vdash \tau_1 \xrightarrow{\mathrm{hd}} \tau_1'}{\vdash \tau_1\ \tau_2 \xrightarrow{\mathrm{hd}} \tau_1'\ \tau_2}$$

Logical Approximation of Substitutions: $\vdash \delta_1 \preccurlyeq_n \delta_2$

$$\frac{\overline{\alpha} = \mathrm{dom}(\delta_1) = \mathrm{dom}(\delta_2) \,\#\, \mathrm{dom}(\Psi) \cup \mathrm{fv}(\Psi) \qquad \forall \alpha \in \overline{\alpha}.\ \vdash \delta_1\alpha \preccurlyeq_n \delta_2\alpha : \kappa_\alpha}{\vdash \delta_1 \preccurlyeq_n \delta_2}$$

Fig. 17. Logical Relations for Proving LTG Consistency

prove Preservation (Theorem 7.10 in the previous section) and the Canonical Values lemma (Lemma 7.11) in the presence of a type conversion rule like econv.

Typically, consistency is established either by proving a normalization theorem or by exhibiting a direct algorithm for deciding type equivalence, from which it falls out as a simple corollary [Stone and Harper 2006]. In the case of LTG's type system, however, neither option applies because of the potential for cyclic type definitions in our Ψ context; at least we do not know of any algorithm for deciding type equivalence or normalizing types in LTG.

Fortunately, we can instead prove consistency directly, using a logical-relations argument that is quite similar to those commonly used for proving the correctness of normalization and equivalence-checking routines. The key idea is (1) to define a notion of head reduction, in the standard way but augmented with the reduction of type variables in Ψ to their definitions, and (2) to show that, if two types τ_1 and τ_2 (of base kind type) are equivalent, and if τ_1 head-normalizes, then τ_2 does as well, and to a type with the same head constructor (*e.g.,* $\to$, $\forall$) as τ_1. It may be that *neither* τ_1 nor τ_2 head-normalizes, but this is fine since our goal is to prove consistency, not normalization.

In order to account for the possibility of head reduction diverging, we employ a *step-indexed* logical relation [Appel and McAllester 2001; Ahmed 2006]. Our use of step-indexing is a bit degenerate in the sense that the step-index is not used to make the definition well-founded but merely in order to facilitate a form of coinductive reasoning

in the proof that the logical relation is *reflexive* (Lemma 7.20 below). This is to our knowledge a novel application of step-indexing, but a fairly natural one, and the proof follows closely in the style of Ahmed [2006]. Crucially, we rely on her non-standard formulation of the *transitivity* property of the logical relation (Lemma 7.21 below).

Before sketching the proof, let us mention a few salient points of hygiene:

(1) We assume at the outset of the proof that we are given an equivalence environment Ψ that is well-formed. This Ψ is constant throughout the proof, so we do not bother to mention it explicitly everywhere.
(2) We assume (as before) that type variables are sorted according to their kind, and write κ_α to denote the implicit kind of α.
(3) We assume implicitly that all types we work with are well-kinded in some suitable Δ, and often omit the kind κ from the judgments in Figure 17 since it can always be inferred (given hygienic point 2).
(4) For brevity, we show here the relevant rules for only one of the base type constructors ($\forall\alpha{:}\hat{\kappa}.\tau^\iota$); the others are similar.

Figure 17 displays the definition of the logical relation employed in our proof. The main judgment, $\vdash \tau_1 \preccurlyeq_n \tau_2 : \kappa$, defines a *logical approximation* relation on types, which is by definition included in the definitional type equivalence judgment $\vdash \tau_1 \equiv \tau_2 : \kappa$. (We write $\vdash \tau_1 \equiv \tau_2 : \kappa$ here to denote that, in addition to being equivalent, τ_1 and τ_2 have kind κ.) It is defined inductively on the kind κ. Type constructors of arrow kind are logically related if they map logically related arguments to logically related results; we quantify over a smaller step-index j in order to ensure downward-closure of the logical relation (Lemma 7.17 below). For type constructors τ_1 and τ_2 of base kind type, we say that τ_1 logically approximates τ_2 for n steps if, whenever τ_1 head-normalizes in n or fewer steps to a *path* (*i.e.,* head-normal form) ρ_1, it is also true that τ_2 head-normalizes to a path ρ_2, and furthermore ρ_1 and ρ_2 are equivalent according to the *path equivalence* judgment $\vdash \rho_1 \leftrightarrow \rho_2 : \kappa$, which compares the paths structurally.

We will show that definitional equivalence implies logical approximation at any step index. That, together with the fact that base types are by definition in head-normal form, gives us consistency.

LEMMA 7.16 (SOUNDNESS OF PATH EQUIVALENCE AND HEAD REDUCTION).

(*1*) *If* $\vdash \rho_1 \leftrightarrow \rho_2 : \kappa$, *then* $\vdash \rho_1 \equiv \rho_2$.
(*2*) *If* $\vdash \tau_1 \xrightarrow{\mathrm{hd}}{}^* \tau_2$, *then* $\vdash \tau_1 \equiv \tau_2$.

PROOF. By straightforward induction on the derivation of the premise. □

LEMMA 7.17 (DOWNWARD CLOSURE OF THE LOGICAL RELATION).
If $\vdash \tau_1 \preccurlyeq_n \tau_2 : \kappa$ *and* $j \leq n$, *then* $\vdash \tau_1 \preccurlyeq_j \tau_2 : \kappa$.

LEMMA 7.18 (CLOSURE UNDER HEAD EXPANSION).

(*1*) *If* $\vdash \tau_1' \preccurlyeq_n \tau_2' : \kappa$ *and* $\vdash \tau_1 \xrightarrow{\mathrm{hd}}{}^* \tau_1'$ *and* $\vdash \tau_2 \xrightarrow{\mathrm{hd}}{}^* \tau_2'$, *then* $\vdash \tau_1 \preccurlyeq_n \tau_2 : \kappa$.
(*2*) *If* $\vdash \tau_1' \preccurlyeq_n \tau_2 : \kappa$ *and* $\vdash \tau_1 \xrightarrow{\mathrm{hd}}{}^j \tau_1'$, *then* $\vdash \tau_1 \preccurlyeq_{n+j} \tau_2 : \kappa$.
(*3*) *If* $\vdash \tau_1 \equiv \tau_2 : \kappa$ *and* $\vdash \tau_1 \xrightarrow{\mathrm{hd}}{}^j \tau_1'$, *then* $\forall n < j.\ \vdash \tau_1 \preccurlyeq_n \tau_2 : \kappa$.

PROOF. By straightforward induction on κ. We show the most interesting case:

(*2*) — Case: $\kappa = \kappa' \to \kappa''$. Suppose $\vdash \tau_1'' \preccurlyeq_k \tau_2'' : \kappa'$, where $k \leq n + j$. By definition, $\vdash \tau_1\tau_1'' \xrightarrow{\mathrm{hd}}{}^j \tau_1'\tau_1''$. It remains to show $\vdash \tau_1\tau_1'' \preccurlyeq_k \tau_2\tau_2'' : \kappa''$.
— Let $m = \min(k, n)$.

— By Lemma 7.17 and the assumption, we have $\vdash \tau_1'\tau_1'' \preccurlyeq_m \tau_2\tau_2'' : \kappa''$.
— By induction, $\vdash \tau_1\tau_1'' \preccurlyeq_{m+j} \tau_2\tau_2'' : \kappa''$.
— By Lemma 7.17, since $k \leq m + j$, we have $\vdash \tau_1\tau_1'' \preccurlyeq_k \tau_2\tau_2'' : \kappa''$.

Note that the proof of (3) is trivial since n is strictly less than j (so the reduction of τ_1 causes the step-index "clock" to run out and we don't have anything to show). □

LEMMA 7.19 ("MAIN" LEMMA: $\leftrightarrow \subseteq \preccurlyeq \subseteq \equiv$).

(1) *If* $\vdash \tau_1 \preccurlyeq_n \tau_2 : \kappa$, *then* $\vdash \tau_1 \equiv \tau_2$.
(2) *If* $\vdash \rho_1 \leftrightarrow \rho_2 : \kappa$, *then* $\forall n. \vdash \rho_1 \preccurlyeq_n \rho_2 : \kappa$.
(3) *If* $\vdash \delta_1 \preccurlyeq_n \delta_2$ *and* $\vdash \tau_1 \equiv \tau_2$, *then* $\vdash \delta_1\tau_1 \equiv \delta_2\tau_2$.
(4) *If* $\overline{\alpha} \# \mathrm{dom}(\Psi) \cup \mathrm{fv}(\Psi)$, *then* $\forall n. \vdash \{\overline{\alpha \mapsto \alpha}\} \preccurlyeq_n \{\overline{\alpha \mapsto \alpha}\}$.

PROOF.

(1) Immediate, by definition.
(2) By Lemma 7.16, we have $\vdash \rho_1 \equiv \rho_2 : \kappa$. Then, by induction on κ:
— Case: $\kappa = \mathsf{type}$. Immediate.
— Case: $\kappa = \kappa' \to \kappa''$.
— Suppose $\vdash \tau_1' \preccurlyeq_k \tau_2' : \kappa'$ for $k \leq n$.
— By part (1), $\vdash \tau_1' \equiv \tau_2'$.
— By the assumption, $\vdash \rho_1\ \tau_1' \leftrightarrow \rho_2\ \tau_2' : \kappa''$.
— By induction, $\vdash \rho_1\ \tau_1' \preccurlyeq_k \rho_2\ \tau_2' : \kappa''$.
(3) By part (1), this reduces to a standard "functionality" property, which is straightforward to prove by induction on the derivation of $\vdash \tau_1 \equiv \tau_2$.
(4) Straightforward, by part (2). □

The next lemma is the one in which step-indices play a crucial role.

LEMMA 7.20 (REFLEXIVITY OF THE LOGICAL RELATION).

(1) *If* $\vdash \tau : \kappa$ *and* $\vdash \delta_1 \preccurlyeq_n \delta_2$, *then* $\vdash \delta_1\tau \preccurlyeq_n \delta_2\tau : \kappa$.
(2) *If* $\vdash \tau : \kappa$, *then* $\forall n. \vdash \tau \preccurlyeq_n \tau : \kappa$.

PROOF.

(1) By induction first on n and second on τ. All cases are straightforward, using the above lemmas (standard logical-relations proof). The only interesting cases are the ones for variables α:
— Case: $\tau = \alpha$, where $\alpha \notin \mathrm{dom}(\Psi)$.
— If $\alpha \in \mathrm{dom}(\delta_1)$, then the result follows from the second assumption.
— Else, the result follows from part (2) of Lemma 7.19, choosing $\rho_1 = \rho_2 = \alpha$.
— Case: $\tau = \alpha$, where $\alpha := \tau' \in \Psi$ and $\vdash \tau' : \kappa$ (by our implicit assumption on Ψ).
— By the second assumption, $\delta_1\tau = \delta_2\tau = \alpha$ and $\delta_1\tau' = \delta_2\tau' = \tau'$ and $\vdash \alpha \xrightarrow{\mathrm{hd}} \tau'$.
— If $n = 0$, then by part (3) of Lemma 7.18, we have $\vdash \alpha \preccurlyeq_0 \alpha : \kappa$.
— If $n > 0$, then by induction, $\vdash \tau' \preccurlyeq_{n-1} \tau' : \kappa$, and by Lemma 7.18, $\vdash \alpha \preccurlyeq_n \alpha : \kappa$.
(2) Immediate, from part (1), picking δ_1 and δ_2 to be the empty substitution. □

Our statement and proof of the transitivity of the logical relation follow Ahmed [2006]. The reason that the theorem is stated in a somewhat odd way is that transitivity of logical approximation does not hold for a fixed n—it only holds if the second pair of types (τ_2 and τ_3 below) are logically related at *all* step indices. Intuitively, this is because the logical-relatedness of τ_1 and τ_2 for n steps yields no information about how many steps τ_2 may take to head-normalize. Furthermore, like Ahmed's proof, ours

makes critical use of the fact that we have (implicitly) built our logical relation over syntactically well-kinded types. This is the essential technical device that she used (as do we) in order to invoke the reflexivity lemma at a key point in the proof. For more context, we refer the interested reader to her paper.

LEMMA 7.21 (TRANSITIVITY OF THE LOGICAL RELATION).

(*1*) *If* $\vdash \rho_1 \leftrightarrow \rho_2 : \kappa$ *and* $\vdash \rho_2 \leftrightarrow \rho_3 : \kappa$*, then* $\vdash \rho_1 \leftrightarrow \rho_3 : \kappa$.
(*2*) *If* $\vdash \tau_1 \preccurlyeq_n \tau_2 : \kappa$ *and* $\forall k. \vdash \tau_2 \preccurlyeq_k \tau_3 : \kappa$*, then* $\vdash \tau_1 \preccurlyeq_n \tau_3 : \kappa$.

PROOF.

(1) By straightforward induction on the structure of ρ_1.
(2) By induction on κ. Clearly, $\vdash \tau_1 \equiv \tau_3 : \kappa$ by transitivity of $\equiv$.
   — Case: $\kappa = \mathsf{type}$. Suppose $j_1 \leq n$ and $\vdash \tau_1 \xrightarrow{\mathrm{hd}}^{j_1} \rho_1$.
      — By the first assumption, $\exists j_2. \vdash \tau_2 \xrightarrow{\mathrm{hd}}^{j_2} \rho_2$ and $\vdash \rho_1 \leftrightarrow \rho_2 : \mathsf{type}$.
      — By the second assumption (thanks to the universal quantification over k, which we instantiate with j_2), $\vdash \tau_3 \xrightarrow{\mathrm{hd}}^{*} \rho_3$ and $\vdash \rho_2 \leftrightarrow \rho_3 : \mathsf{type}$.
      — By part (1), $\vdash \rho_1 \leftrightarrow \rho_3 : \mathsf{type}$.
   — Case: $\kappa = \kappa' \to \kappa''$. Suppose $j \leq n$ and $\vdash \tau_1' \preccurlyeq_j \tau_2' : \kappa'$.
      — By the first assumption, $\vdash \tau_1\tau_1' \preccurlyeq_j \tau_2\tau_2' : \kappa''$.
      — By Lemma 7.20, $\forall k. \vdash \tau_2' \preccurlyeq_k \tau_2' : \kappa'$.
      — By the second assumption, $\forall k. \vdash \tau_2\tau_2' \preccurlyeq_k \tau_3\tau_2' : \kappa''$.
      — By induction, $\vdash \tau_1\tau_1' \preccurlyeq_j \tau_3\tau_2' : \kappa''$. □

THEOREM 7.22 (FUNDAMENTAL THEOREM OF LOGICAL RELATIONS).

(*1*) *If* $\vdash \tau_1 \equiv \tau_2 : \kappa$ *and* $\vdash \delta_1 \preccurlyeq_n \delta_2$*, then* $\vdash \delta_1\tau_1 \preccurlyeq_n \delta_2\tau_2 : \kappa$ *and* $\vdash \delta_1\tau_2 \preccurlyeq_n \delta_2\tau_1 : \kappa$.
(*2*) *If* $\vdash \tau_1 \equiv \tau_2 : \kappa$*, then* $\forall n. \vdash \tau_1 \preccurlyeq_n \tau_2 : \kappa$.

PROOF.

(1) By induction on the derivation of the first assumption. The proofs for all structural equivalence rules are analogous to the proofs for the corresponding formation rules in Lemma 7.20. The proofs for the reduction rules (qdelta, qbeta, and qeta) follow easily from Lemmas 7.18 and 7.20. The proof of symmetry follows directly from the generalized statement of the theorem (*i.e.,* the fact that we prove both $\delta_1\tau_1 \preccurlyeq_n \delta_2\tau_2$ and $\delta_1\tau_2 \preccurlyeq_n \delta_2\tau_1$ simultaneously). The interesting case is the one for transitivity:
   — Case: $\vdash \tau_1 \equiv \tau_2 : \kappa$ and $\vdash \tau_2 \equiv \tau_3 : \kappa$, and we must show $\vdash \delta_1\tau_1 \preccurlyeq_n \delta_2\tau_3 : \kappa$ and $\vdash \delta_1\tau_3 \preccurlyeq_n \delta_2\tau_1 : \kappa$. We show the former; the proof of the latter is symmetric.
      — By induction, $\vdash \delta_1\tau_1 \preccurlyeq_n \delta_2\tau_2 : \kappa$.
      — By Lemma 7.20, $\forall k. \vdash \delta_2 \preccurlyeq_k \delta_2$.
      — By induction, $\forall k. \vdash \delta_2\tau_2 \preccurlyeq_k \delta_2\tau_3 : \kappa$.
      — By Lemma 7.21, $\vdash \delta_1\tau_1 \preccurlyeq_n \delta_2\tau_3 : \kappa$.
(2) Immediate, from part (1), picking δ_1 and δ_2 to be the empty substitution. □

COROLLARY 7.23 (CONSISTENCY). *Suppose* $\Psi \vdash \rho_1 \equiv \rho_2$.

(*1*) *If* $\rho_1 = \forall\alpha{:}\hat{\kappa}.\tau_1'$*, then* $\Psi \vdash \rho_2 \xrightarrow{\mathrm{hd}}^{*} \forall\alpha{:}\hat{\kappa}.\tau_2'$*, where* $\Psi \vdash \tau_1 \equiv \tau_2$.
(*2*) *. . . similarly for the other base type constructors (*$\to$*,* $\{\overline{\ell : \_}\}$*,* $\exists$*,* $?$*).*

8. EVIDENCE TRANSLATION AND SOUNDNESS

We define the dynamic semantics of MixML by translation into the internal language LTG as defined in the previous section. The translation employs a backpatching semantics, using reference cells to enable recursive linking for dynamic module components. Thus, a module mod of signature Σ translates to a *function*—the module *initializer*—whose argument has the same shape as Σ but with uninitialized references corresponding to the exports of Σ (the imports may or may not be initialized). When called, this function will patch in definitions for those exports. In so doing, the function may attempt to dereference any component of the argument (import or export), which may in turn result in a run-time error if that component has not yet been backpatched. This approach enables a translation of recursive linking that avoids the need for a complex fixed-point operation. At the same time, the linear typing of LTG ensures that every module component actually gets defined (in a complete module).

The translation is given by the rules in Figures 18-21. The structure of these rules is identical to that of the typing rules from Section 4, except that each judgment produces an LTG term as additional output.

For notational convenience, we omit kind annotations from type variables in LTG terms. As before, we write κ_α for the implicit kind of α. In binders, we mean α^ι to be short-hand for $\alpha{:}\kappa_\alpha^\iota$. Moreover, we identify internal and external language kinds, that is, we assume that syntactic kinds knd and "semantic kinds" κ range over the same phrases, and we use them interchangeably.

Assumptions about the core language. In order to be able to translate atomic modules, we assume that the core language judgments introduced at the end of Section 4.1 can be extended to translation judgments producing well-formed LTG terms. Further, we assume that these judgments can be proven sound and complete in conjunction with the module judgments (the proofs are interdependent because the grammars are). The details of the required properties are given with the respective Theorems 8.1 and 8.7, and Lemmas 8.4 and 8.5 in Section 8.3 below.

8.1. Erasure

Figure 18 defines an *erasure* $(\_)^\circ$ from MixML semantic signatures (cf. Figure 4) into LTG types. The erasure of the semantic signatures derived by the MixML typing rules corresponds to the LTG terms that are produced by the translation, *i.e.,* the derived terms serve as *evidence* for the derived types.

A unit of signature $\forall\overline{\alpha}.\,\exists\overline{\beta}.\,(\mathcal{L};\Sigma)$ is represented by a polymorphic initializer function of type $\forall^{\mathsf{U}}\overline{\alpha}.\forall^{\mathsf{L}}\overline{\beta}.(\Sigma^\circ \to \{\})$, which is in *destination-passing style* [Dreyer 2007a]. That is, the function takes import types $\overline{\alpha}$, as-yet-undefined export type names $\overline{\beta}$ (with linear kind), and the representation of the complete module as a value of type Σ° with all dynamic content represented by references, and where the export components are yet uninitialized (and thus linear). The function defines the export types and fills in the dynamic content of the export terms. Recall that the the notation $\forall^\iota\alpha.\tau$ used here was defined in Figure 12; it implies that the inner $(\Sigma^\circ \to \{\})$ has to be linear if and only if $\overline{\beta}$ is not empty.

Dynamic atomic modules—*i.e.,* values and higher-order units—are therefore represented as lazy references $(?\tau)^\iota$. A reference has linear mode if it corresponds to an export (because the respective initializer is expected to define it), and unrestricted mode if it corresponds to an import.

A structure $\{\!|\overline{\ell : \Sigma}|\!\}$ is represented as an LTG record $\{\overline{\ell : \Sigma^\circ}\}^\iota$, being linear if at least one of its components is.

$$
\begin{array}{lcl@{\qquad}lcl}
[\![= \mathrm{A}]\!]^\circ & \stackrel{\text{def}}{=} & (\forall \alpha^{\mathrm{U}}.(\alpha\,\mathrm{A}^\circ \to \alpha\,\mathrm{A}^\circ)^{\mathrm{U}})^{\mathrm{U}} & \langle\!|\Phi|\!\rangle^\circ & \stackrel{\text{def}}{=} & \Phi^\circ \\
([\![\mathrm{A}]\!]^{+/-})^\circ & \stackrel{\text{def}}{=} & (?\mathrm{A}^\circ)^{\mathrm{L/U}} & \epsilon^\circ & \stackrel{\text{def}}{=} & \epsilon \\
([\![\Phi]\!]^{+/-})^\circ & \stackrel{\text{def}}{=} & (?\Phi^\circ)^{\mathrm{L/U}} & (\Gamma, \mathrm{X}:\Sigma)^\circ & \stackrel{\text{def}}{=} & \Gamma^\circ, \mathrm{X}:\Sigma^{-\circ} \\
\{\!|\overline{\ell:\Sigma}|\!\}^\circ & \stackrel{\text{def}}{=} & \{\overline{\ell:\Sigma^\circ}\}^{\mathrm{U}} \text{ (if } \overline{\Sigma^\circ \preceq \mathrm{U}}) & \epsilon^\circ & \stackrel{\text{def}}{=} & \epsilon \\
\{\!|\overline{\ell:\Sigma}|\!\}^\circ & \stackrel{\text{def}}{=} & \{\overline{\ell:\Sigma^\circ}\}^{\mathrm{L}} \text{ (otherwise)} & (\delta, \alpha \mapsto \mathrm{A})^\circ & \stackrel{\text{def}}{=} & \delta^\circ, \alpha \mapsto \mathrm{A}^\circ \\
(\forall \overline{\alpha}.\, \exists \overline{\beta}.\, (\mathcal{L}; \Sigma))^\circ & \stackrel{\text{def}}{=} & \forall^{\mathrm{U}} \overline{\alpha}.\, \forall^{\mathrm{L}} \overline{\beta}.\, (\Sigma^\circ \to \{\}^{\mathrm{U}}) & \Sigma^{-\circ} & \stackrel{\text{def}}{=} & (-|\Sigma|)^\circ
\end{array}
$$

$$
\begin{array}{lcl@{\qquad}lcl}
\mathrm{Create}([\![= \mathrm{A}]\!]) & \stackrel{\text{def}}{=} & \lambda \alpha^{\mathrm{U}}.\, \lambda x{:}(\alpha\,\mathrm{A}^\circ)^{\mathrm{U}}.\, x & \mathrm{Copy}(e_1, e_2 : [\![= \mathrm{A}]\!]) & \stackrel{\text{def}}{=} & \{\} \\
\mathrm{Create}([\![\mathrm{A}]\!]^+) & \stackrel{\text{def}}{=} & \mathsf{new}\ \mathrm{A}^\circ & \mathrm{Copy}(e_1, e_2 : [\![\mathrm{A}]\!]^{+/-}) & \stackrel{\text{def}}{=} & \mathsf{def}\ e_{2/1} := !\, e_{1/2} \\
\mathrm{Create}([\![\Phi]\!]^+) & \stackrel{\text{def}}{=} & \mathsf{new}\ \Phi^\circ & \mathrm{Copy}(e_1, e_2 : [\![\Phi]\!]^{+/-}) & \stackrel{\text{def}}{=} & \mathsf{def}\ e_{2/1} := !\, e_{1/2} \\
\mathrm{Create}(\{\!|\overline{\ell:\Sigma}|\!\}) & \stackrel{\text{def}}{=} & \{\overline{\ell = \mathrm{Create}(\Sigma)}\} & \mathrm{Copy}(e_1, e_2 : \{\!|\overline{\ell:\Sigma}|\!\}) & \stackrel{\text{def}}{=} & \mathsf{let}\{\overline{\ell = x_1}\} = e_1\ \mathsf{in} \\
 & & & & & \mathsf{let}\{\overline{\ell = x_2}\} = e_2\ \mathsf{in} \\
 & & & & & \overline{\mathrm{Copy}(x_1, x_2 : \Sigma)}
\end{array}
$$

Fig. 18. Auxiliary Definitions for Evidence Translation

Following Rossberg et al. [2010], atomic type components $[\![= \mathrm{A}]\!]$ are represented by higher-kinded polymorphic functions $\forall \alpha.\alpha\,\mathrm{A}^\circ \to \alpha\,\mathrm{A}^\circ$, where $\kappa_\alpha = \kappa_{\mathrm{A}} \to \mathsf{type}$. This is merely a coding trick: the computational content of values of this type is not actually relevant, we only care that it exists and that it uniquely determines the type A.

The erasure A° of core types and constructors simply decorates all constituent types with mode U and erases all contained package types into LTG universal types, according to the definition given in Figure 18.

Note the little subtlety that module signatures Σ erase to moded types τ^ι, while unit signatures Φ and core types A erase to plain types τ. This is because the latter two typically appear in syntactic contexts where the mode is determined separately, or is not present at all (*e.g.,* for the content type of a reference).

To relate derivations for our external language MixML to derivations in our internal language LTG, erasure is extended to module environments Γ in a pointwise fashion[8]—however, all types are made unrestricted in the environment (notation $\Sigma^{-\circ}$). That is because the free module variables of an initializer are different in nature from its arguments: the latter are to be defined by the initializer, but the former are merely intended to be read.

Example. Consider the following example of a simple module expression that defines a unit importing type `t` and term `v`, and exporting the types `u` and `s`—with the latter being abstract—along with the term `w`:

$$
\left[\left\{\begin{array}{l}
\mathtt{t} = \mathtt{[:type]}, \\
\mathtt{u} = \mathtt{[t \times t]}, \\
\mathtt{s} = \mathtt{[:type]\ seals\ [t \times t]}, \\
\mathtt{v} = \mathtt{[:t]}, \\
\mathtt{w} = \mathtt{[(v, v)]}
\end{array}\right\}\right]
$$

[8]We use Γ to range over external language module environments as well as internal language term environments, but it should always be clear from context which is meant. Likewise for type substitutions δ.

The semantic signature of this unit, as derived by the typing rules, is

$$\forall\alpha.\,\exists\beta.\ \left(\{\mathtt{t} : [\![=\alpha]\!]\};\ \begin{Bmatrix}\mathtt{t} : [\![=\alpha]\!],\\ \mathtt{u} : [\![=\alpha\times\alpha]\!],\\ \mathtt{s} : [\![=\beta]\!],\\ \mathtt{v} : [\![\alpha]\!]^-,\\ \mathtt{w} : [\![\alpha\times\alpha]\!]^+\end{Bmatrix}\right)$$

Erasure of this signature yields the following polymorphic function type, where export types and terms turn into linear "destination" arguments:

$$\forall\alpha^{\mathrm{U}}.\,\forall\beta^{\mathrm{L}}.\ \left(\begin{Bmatrix}\mathtt{t} : [\![=\alpha]\!]^\circ,\\ \mathtt{u} : [\![=\alpha\times\alpha]\!]^\circ,\\ \mathtt{s} : [\![=\beta]\!]^\circ,\\ \mathtt{v} : (?\alpha)^{\mathrm{U}},\\ \mathtt{w} : (?(\alpha\times\alpha))^{\mathrm{L}}\end{Bmatrix}^{\mathrm{L}} \to \{\}^{\mathrm{U}}\right)^{\mathrm{L}}$$

A possible evidence term for this unit is the following:

$$\lambda\alpha^{\mathrm{U}}.\,\lambda\beta^{\mathrm{L}}.\,\lambda x : \begin{Bmatrix}\mathtt{t} : [\![=\alpha]\!]^\circ,\\ \mathtt{u} : [\![=\alpha\times\alpha]\!]^\circ,\\ \mathtt{s} : [\![=\beta]\!]^\circ,\\ \mathtt{v} : (?\alpha)^{\mathrm{U}},\\ \mathtt{w} : (?(\alpha\times\alpha))^{\mathrm{L}}\end{Bmatrix}^{\mathrm{L}}.\ \mathsf{def}\ \beta := \alpha\times\alpha\ \mathsf{in}\ \{\};\ \mathsf{def}\ x.\mathtt{w} := (!\,x.\mathtt{v}, !\,x.\mathtt{v})$$

It takes import type α and (linear) export type name β, as well as the argument x carrying the module to initialize, and initializes its exports by defining the type name β and the reference at $x.\mathtt{w}$. (Because $\mathtt{s}$ has no other components besides the sealed type, the body of the corresponding def for β is empty.)

The actual translation rules have to be formulated in a compositional manner, hence they will actually produce a somewhat more complicated term for the example unit, but it will be operationally equivalent to the one just shown.

8.2. Translation rules

Given the ideas just described, most of the evidence translation is rather straightforward.

Modules and Units. The main judgment for translating modules (Figure 19) yields an LTG function e that is an initializer for a module of type Σ°. It takes a value x of type Σ° as a partial representation of the module, and defines all linear references it contains, thereby initializing its term exports. It will also define all exported type names. (To avoid clutter, we omit duplicating the type annotation on all initializer arguments x in the evidence terms. We also omit the type annotations $\hat\tau$ for expressions $\mathsf{def}\ \alpha{:=}\tau\ \mathsf{in}\ (e : \hat\tau)$, since it is just $\{\}^{\mathrm{U}}$ in all places of our translation.)

Being in destination-passing style, initializers are seemingly "backward" with respect to the modules they define. 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 to pass to the respective operand initializers.

The main points of interest are thus the following. Rule $\mathsf{UNIT}^{\rightsquigarrow}$ closes an initializer over all its type arguments, producing a stand-alone unit initializer. Conversely, rule $\mathsf{NEW}^{\rightsquigarrow}$ applies this function to initialize a unit (dereferencing the cell it is taken from first). In an analogous manner, rule $\mathsf{UNPACK}^{\rightsquigarrow}$ handles unpacking of units that have been packaged as first-class values. Rule $\mathsf{COMPL}^{\rightsquigarrow}$ implements a complete, initialized module by first creating a fresh, uninitialized module (with the help of the

meta-function $\mathrm{Create}(\_)$ defined in Figure 18) along with fresh export type names, and then invoking the initializer expression e.

New modules also have to be created in rule $\mathsf{DOT}^{\rightsquigarrow}$ for projection and rule $\mathsf{SEAL}^{\rightsquigarrow}$ for opaque linking, which are the two constructs that mask parts of a larger module—the evidence expression thus must inversely create the skeletons for these larger modules. Specifically, the latter rule must locally create the combined module comprising both mod_1 and mod_2, and then copy out the restricted export to the destination. Moreover, it defines the abstract types $\overline{\alpha_1}$ that are introduced by the opaque linking. Notably, the evidence terms for this rule and rule $\mathsf{LINK}^{\rightsquigarrow}$ let-bind X_1 in the scope of e_2, in order to mirror the environment extension in the premises. (The terms also bind further variables, such as X_2, which we assume to be fresh and hence non-capturing.)

Copying is defined in Figure 18 by induction on the annotated semantic signature Σ. Its definition is bidirectional: given two module representations with opposite import/export polarities, it generates code for copying every import slot from one of the modules to the respective export slot of the other, and vice versa. In the main judgment, all uses of copying are unidirectional (left to right), since the respective Σ's are all absolute, but bidirectional copying is in fact used in the translation of unit signature matching (rule $\mathsf{MATCH}^{\rightsquigarrow}$ below). Note: it is important in both rules $\mathsf{SEAL}^{\rightsquigarrow}$ and $\mathsf{MATCH}^{\rightsquigarrow}$ that the copying operation merely links components together and does not actually dereference any right away, since the evidence for those rules only defines the components in question *after* the copying is performed. This behavior is guaranteed by the semantics of $\mathsf{def}\, e_{1/2} := !\, e_{2/1}$, which delays the computation of $!\, e_{2/1}$ until $e_{1/2}$ is accessed.

Merging. The translation of transparent and opaque linking relies on evidence for signature merging: the merging rules (Figure 21) produce evidence functions f_1 and f_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 in the linking rules to initialize the operands of linking. (For the rules regarding structures, recall the various abbreviations for manipulating LTG records that we defined in Figure 12.)

The only interesting bits in the translation of merging itself 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 reference that applies the coercion function obtained as evidence of the respective subtyping judgment. In the case of units, the coercion function is created as evidence of the signature matching rule $\mathsf{MATCH}^{\rightsquigarrow}$.

As in the case of copying, it is vital for this translation that references are non-strict (Section 7.1): coercions have to be applied lazily, so that they do not request a recursive definition prematurely. Consider the following (contrived) example, where $\mathtt{sqrt} : \mathtt{float} \to \mathtt{float}$, but we take `int` to be a (coercive) subtype of `float`:

$$(\mathrm{X}_1 = \{\mathsf{a} = [\mathtt{:float}]\})\ \mathtt{with}\ \{\mathsf{a} = [2], \mathsf{b} = [\mathtt{sqrt}\, \mathrm{X}_1.\mathsf{a}]\}$$

The signature of this module is $\{\!|\mathsf{a} : [\![\mathtt{int}]\!]^+, \mathsf{b} : [\![\mathtt{float}]\!]^+|\!\}$, because $\mathtt{int} \leq \mathtt{float}$. The evidence translation will produce the moral equivalent (modulo a number of simplifications) of the following initializer term, assuming f is the coercion function witnessing $\mathtt{int} \leq \mathtt{float}$ in the core language:

$$\begin{array}{l}
\lambda x : \{\mathsf{a} : (?\mathtt{int})^{\mathrm{L}}, \mathsf{b} : (?\mathtt{float})^{\mathrm{L}}\}^{\mathrm{L}}. \\
\quad \mathsf{let}\, \mathrm{X}_1 = \mathsf{let}\, x_1 = \{\mathsf{a} = \mathsf{new}(\mathtt{float})\}\ \mathsf{in}\ \mathsf{def}\, x_1.\mathsf{a} := f\,(!\, x.\mathsf{a}); x_1\ \mathsf{in} \\
\quad \mathsf{let}\, \mathrm{X}_2 = x\ \mathsf{in} \\
\quad \mathsf{let}\, x_{\mathsf{a}} = 2\ \mathsf{in}\ \mathsf{def}\, \mathrm{X}_2.\mathsf{a} := x_{\mathsf{a}}; \\
\quad \mathsf{let}\, x_{\mathsf{b}} = \mathtt{sqrt}\,(!\mathrm{X}_1.\mathsf{a})\ \mathsf{in}\ \mathsf{def}\, \mathrm{X}_2.\mathsf{b} := x_{\mathsf{b}}
\end{array}$$

If the expression $f(!\,x.\mathrm{a})$ were evaluated eagerly in the assignment, reading $x.\mathrm{a}$ would result in a runtime error because it would yet be undefined at that point. But once $x.\mathrm{a}$ has been initialized, which happens before $!\,\mathrm{X}_1.\mathrm{a}$ is evaluated, it is safe to read it.

To circumvent laziness where it is *not* wanted, our translation puts the actual expressions defining the components $x.\mathrm{a}$ and $x.\mathrm{b}$ into let-bindings before their respective def-expressions in rule $\mathsf{EVAL}^{\widetilde{\rightarrow}}$. This way, they are still evaluated strictly, and ultimately, all initialization is entirely sequential, as expected. In other words, our use of non-strictness is benign and not externally observable. (For the other place defining a reference, rule $\mathsf{EUN}^{\widetilde{\rightarrow}}$, strictness does not matter, since e is the translation of a unit, which always is a lambda.)

Unit Signature Matching. The evidence of unit signature matching is a higher-order function taking a unit initializer y of the smaller type and delivering one of the larger (rule $\mathsf{MATCH}^{\widetilde{\rightarrow}}$, Figure 21). To do so, the resulting function creates evidence for an auxiliary module x of signature $|\Sigma|$, which makes the connection between the "smaller" module signature Σ_1 and the "larger" Σ_2. It also creates a fresh set of export types $\overline{\beta_1}$ for the original unit initializer y and defines its own exports $\overline{\beta_2}$ using the type substitution δ derived by the typing rule. In a similar manner, substituted import types $\overline{\alpha_1}$ are passed to y.

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 $f_2\,x$, 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 (Figure 18) is such that the assignment is done in the appropriate direction for each component, depending on its variance.

8.3. Soundness and Completeness of Evidence Translation

In order to prove that our evidence translation yields a sound operational semantics for MixML, we need to show that the translation is sound with respect to LTG's typing rules, and complete with respect to the MixML typing rules.

First, for every module deemed well-typed by the MixML typing rules there is a suitable translation—and vice versa, *i.e.,* every module we can translate is well-typed. Since the translation rules just decorate the typing rules, without introducing additional constraints, the proof for both directions is trivial:

THEOREM 8.1 (COMPLETENESS OF TRANSLATION).

(*1*) *If and only if* $\Gamma \vdash \mathit{exp} : \mathrm{A}$*, then* $\Gamma \vdash \mathit{exp} : \mathrm{A} \rightsquigarrow e$.
(*2*) *If and only if* $\Gamma; \mathcal{R}; \overline{\beta} \vdash \mathit{mod} : \Sigma$*, then* $\Gamma; \mathcal{R}; \overline{\beta} \vdash \mathit{mod} : \Sigma \rightsquigarrow e$.
(*3*) *If and only if* $\Gamma \vdash \mathit{mod} : \Sigma$*, then* $\Gamma \vdash \mathit{mod} : \Sigma \rightsquigarrow e$.
(*4*) *If and only if* $\Gamma \vdash \mathit{mod} : \Phi$*, then* $\Gamma \vdash \mathit{mod} : \Phi \rightsquigarrow e$.
(*5*) *If and only if* $\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma$*, then* $\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma \rightsquigarrow f_1/f_2$.
(*6*) *If and only if* $\vdash \mathrm{A}_1 \leq \mathrm{A}_2$*, then* $\vdash \mathrm{A}_1 \leq \mathrm{A}_2 \rightsquigarrow f$.
(*7*) *If and only if* $\vdash \Phi_1 \leq \Phi_2$*, then* $\vdash \Phi_1 \leq \Phi_2 \rightsquigarrow f$.

PROOF. Both directions by straightforward induction on the derivation. The arguments for properties 1 and 6 clearly depend on the core language. We assume them to be provable for any additional constructs not present in our grammar. □

Before we can proceed to show that the LTG terms produced by the translation are actually well-formed, we need to make precise what we mean by "well-formed". We start by defining when a judgment is well-formed relative to a given LTG type evironment Δ:

Modules: $\Gamma; \mathcal{R}; \overline{\beta} \vdash \mathit{mod} : \Sigma \rightsquigarrow e$

$$\frac{\mathrm{X} : |\Sigma| \in \Gamma}{\Gamma; \{\!|\}; \emptyset \vdash \mathrm{X} : |\Sigma| \rightsquigarrow \lambda x.\, \mathrm{Copy}(\mathrm{X}, x : |\Sigma|)}\ (\widetilde{\mathsf{VAR}}) \qquad \frac{}{\Gamma; \{\!|\}; \emptyset \vdash \{\} : \{\!|\}\ \rightsquigarrow \lambda x.\{\}}\ (\widetilde{\mathsf{EMP}})$$

$$\frac{\vdash \mathrm{A} \Uparrow \mathit{knd}}{\Gamma; [\![= \mathrm{A}]\!]; \emptyset \vdash \texttt{[:}\mathit{knd}\texttt{]} : [\![= \mathrm{A}]\!] \rightsquigarrow \lambda x.\{\}}\ (\widetilde{\mathsf{ITYP}}) \qquad \frac{\Gamma \vdash \mathit{typ} \rightsquigarrow \mathrm{A}}{\Gamma; \{\!|\}; \emptyset \vdash \texttt{[}\mathit{typ}\texttt{]} : [\![= \mathrm{A}]\!] \rightsquigarrow \lambda x.\{\}}\ (\widetilde{\mathsf{ETYP}})$$

$$\frac{\Gamma \vdash \mathit{typ} \rightsquigarrow \mathrm{A} \qquad \vdash \mathrm{A} \Uparrow \mathsf{type}}{\Gamma; \{\!|\}; \emptyset \vdash \texttt{[:}\mathit{typ}\texttt{]} : [\![\mathrm{A}]\!]^- \rightsquigarrow \lambda x.\{\}}\ (\widetilde{\mathsf{IVAL}}) \qquad \frac{\Gamma \vdash \mathit{exp} : \mathrm{A} \rightsquigarrow e}{\Gamma; \{\!|\}; \emptyset \vdash \texttt{[}\mathit{exp}\texttt{]} : [\![\mathrm{A}]\!]^+ \rightsquigarrow \lambda x.\, \mathsf{let}\, x'{=}e\ \mathsf{in}\ \mathsf{def}\, x{:=}x'}\ (\widetilde{\mathsf{EVAL}})$$

$$\frac{\Gamma; \mathcal{R}; \overline{\beta} \vdash \mathit{mod} : \Sigma \rightsquigarrow e}{\Gamma; \{\!|\ell : \mathcal{R}|\!\}; \overline{\beta} \vdash \{\ell = \mathit{mod}\} : \{\!|\ell : \Sigma|\!\} \rightsquigarrow \lambda x.\ \mathsf{let}\, \{\ell{=}x'\} = x\ \mathsf{in}\ e\, x'}\ (\widetilde{\mathsf{STR}})$$

$$\frac{\Gamma; \{\!|\ell : \mathcal{R}|\!\}; \overline{\beta} \vdash \mathit{mod} : \{\!|\ell : \Sigma, \overline{\ell' : |\Sigma'|}|\!\} \rightsquigarrow e}{\Gamma; \mathcal{R}; \overline{\beta} \vdash \mathit{mod}.\ell : \Sigma \rightsquigarrow \lambda x.\ \mathsf{let}\, \{\overline{\ell'{=}x'}\} = \mathrm{Create}(\{\overline{\ell' : |\Sigma'|}\})\ \mathsf{in}\ e\, \{\ell{=}x, \overline{\ell'{=}x'}\}}\ (\widetilde{\mathsf{DOT}})$$

$$\frac{\begin{array}{rr} \vdash \mathcal{L}_1\ \mathsf{locates}\ \overline{\alpha_1} & \mathcal{R}_1 \# \Sigma_2 \\ \vdash \mathcal{L}_2\ \mathsf{locates}\ \overline{\alpha_2} & \mathcal{R}_2 \# \Sigma_1 \\ \vdash (\mathcal{L}_1; \Sigma_1) \rightleftarrows (\mathcal{L}_2; \Sigma_2') \rightsquigarrow \delta & \\ \overline{\alpha_1}, \overline{\alpha_2}\ \mathsf{fresh} & \end{array} \quad \begin{array}{r} \Gamma; \mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{L}_1; \overline{\beta_1} \vdash \mathit{mod}_1 : \Sigma_1 \rightsquigarrow e_1 \\ \Gamma, \mathrm{X}_1 : |\Sigma_1|; \mathcal{R} \uplus \mathcal{R}_2 \uplus \mathcal{L}_2; \overline{\beta_2} \vdash_{\mathrm{stat}} \mathit{mod}_2 : \Sigma_2' \\ \Gamma, \mathrm{X}_1 : |\delta\Sigma_1|; \mathcal{R} \uplus \mathcal{R}_2 \uplus \delta\mathcal{L}_2; \overline{\beta_2} \vdash \mathit{mod}_2 : \Sigma_2 \rightsquigarrow e_2 \\ \vdash \delta\Sigma_1 + \Sigma_2 \Rightarrow \Sigma \rightsquigarrow f_1 / f_2 \end{array}}{\begin{array}{l} \Gamma; \mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{R}_2; \overline{\beta_1}, \overline{\beta_2} \vdash (\mathrm{X}_1 = \mathit{mod}_1)\ \texttt{with}\ \mathit{mod}_2 : \Sigma \rightsquigarrow \lambda x.\, \mathsf{let}\, \mathrm{X}_1 = f_1\, x\ \mathsf{in} \\ \qquad\qquad \mathsf{let}\, \mathrm{X}_2 = f_2\, x\ \mathsf{in} \\ \qquad\qquad \delta^\circ e_1\, \mathrm{X}_1;\ e_2\, \mathrm{X}_2 \end{array}}\ (\widetilde{\mathsf{LINK}})$$

$$\frac{\begin{array}{rr} \vdash \mathcal{L}_1\ \mathsf{locates}\ \overline{\alpha_1} & \\ \vdash \mathcal{L}_2\ \mathsf{locates}\ \overline{\alpha_2} & \\ \vdash (\mathcal{L}_1; \Sigma_1) \rightleftarrows (\mathcal{L}_2; \Sigma_2') \rightsquigarrow \delta & \\ \overline{\beta_2}, \overline{\alpha_2}\ \mathsf{fresh} & \end{array} \quad \begin{array}{r} \Gamma; \mathcal{L}_1; \overline{\beta_1} \vdash \mathit{mod}_1 : \Sigma_1 \rightsquigarrow e_1 \\ \Gamma, \mathrm{X}_1 : |\Sigma_1|; \mathcal{L}_2; \overline{\beta_2} \vdash_{\mathrm{stat}} \mathit{mod}_2 : \Sigma_2' \\ \delta\Gamma, \mathrm{X}_1 : |\delta\Sigma_1|; \delta\mathcal{L}_2; \overline{\beta_2} \vdash \mathit{mod}_2 : \Sigma_2 \rightsquigarrow e_2 \\ \vdash \delta\Sigma_1 + \Sigma_2 \Rightarrow |\Sigma| \rightsquigarrow f_1 / f_2 \end{array}}{\begin{array}{l} \Gamma; \{\!|\}; \overline{\beta_1}, \overline{\alpha_1} \vdash (\mathrm{X}_1 = \mathit{mod}_1)\ \texttt{seals}\ \mathit{mod}_2 : |\Sigma_1| \rightsquigarrow \lambda x.\, \mathsf{new}\, \overline{\beta_2}\ \mathsf{in}\ \mathsf{def}\, \overline{\alpha_1 := \delta^\circ \alpha_1}\ \mathsf{in} \\ \qquad\qquad \mathsf{let}\, x' = \mathrm{Create}(|\Sigma|)\ \mathsf{in} \\ \qquad\qquad \mathsf{let}\, \mathrm{X}_1 = f_1\, x',\ \mathrm{X}_2 = f_2\, x'\ \mathsf{in} \\ \qquad\qquad \mathrm{Copy}(\mathrm{X}_1, x : |\Sigma_1|);\ e_1\, \mathrm{X}_1;\ e_2\, \mathrm{X}_2 \end{array}}\ (\widetilde{\mathsf{SEAL}})$$

$$\frac{\Gamma \vdash \mathit{usig} \rightsquigarrow \Phi}{\Gamma; \{\!|\}; \emptyset \vdash \texttt{[:}\mathit{usig}\texttt{]} : [\![\Phi]\!]^- \rightsquigarrow \lambda x.\{\}}\ (\widetilde{\mathsf{IUN}}) \qquad \frac{\Gamma \vdash \mathit{mod} : \Phi \rightsquigarrow e}{\Gamma; \{\!|\}; \emptyset \vdash \texttt{[}\mathit{mod}\texttt{]} : [\![\Phi]\!]^+ \rightsquigarrow \lambda x.\, \mathsf{def}\, x{:=}e}\ (\widetilde{\mathsf{EUN}})$$

$$\frac{\Gamma \vdash \mathit{mod} : [\![\forall \overline{\alpha}.\, \exists \overline{\beta}.\, (\mathcal{L}; \Sigma)]\!]^+ \rightsquigarrow e \qquad \mathrm{dom}(\delta) = \{\overline{\alpha}, \overline{\beta}\}}{\Gamma; \delta\mathcal{L}; \delta\overline{\beta} \vdash \texttt{new}\ \mathit{mod} : \delta\Sigma \rightsquigarrow \lambda x.\, (!\, e)\ \overline{\delta^\circ \alpha}\ \ \overline{\delta^\circ \beta}\ x}\ (\widetilde{\mathsf{NEW}})$$

$$\frac{\Gamma \vdash \mathit{usig} \rightsquigarrow \forall \overline{\alpha}.\, \exists \overline{\beta}.\, (\mathcal{L}; \Sigma) \qquad \mathrm{dom}(\delta) = \{\overline{\alpha}, \overline{\beta}\} \qquad \Gamma \vdash \mathit{exp} : \langle\!|\forall \overline{\alpha}.\, \exists \overline{\beta}.\, (\mathcal{L}; \Sigma)|\!\rangle \rightsquigarrow e}{\Gamma; \delta\mathcal{L}; \delta\overline{\beta} \vdash \texttt{unpack}(\mathit{exp}\ \texttt{as}\ \mathit{usig}) : \delta\Sigma \rightsquigarrow \lambda x.\, e\ \overline{\delta^\circ \alpha}\ \ \overline{\delta^\circ \beta}\ x}\ (\widetilde{\mathsf{UNPACK}})$$

Fig. 19. Evidence Translation Rules for MixML

Complete Modules: $\Gamma \vdash mod : \Sigma \rightsquigarrow e$

$$\frac{\Gamma; \{\!|\}; \overline{\beta} \vdash mod : |\Sigma| \rightsquigarrow e \qquad \overline{\beta}\ \mathsf{fresh} \qquad \overline{\beta} \notin \mathrm{fv}(\Sigma)}{\Gamma \vdash mod : |\Sigma| \rightsquigarrow \mathsf{new}\,\overline{\beta}\ \mathsf{in}\ \mathsf{let}\, x = \mathrm{Create}(|\Sigma|)\ \mathsf{in}\ e\ x; x}\ (\mathsf{COMPL}^{\rightsquigarrow})$$

Units: $\Gamma \vdash mod : \Phi \rightsquigarrow e$

$$\frac{\Gamma; \mathcal{L}; \overline{\beta} \vdash mod : \Sigma \rightsquigarrow e \qquad \vdash \mathcal{L}\ \mathsf{locates}\ \overline{\alpha} \qquad \overline{\alpha}, \overline{\beta}\ \mathsf{fresh}}{\Gamma \vdash mod : \forall \overline{\alpha}.\, \exists \overline{\beta}.\, (\mathcal{L}; \Sigma) \rightsquigarrow \lambda \overline{\alpha^{\mathrm{U}}}.\, \lambda \overline{\beta^{\mathrm{L}}}.\, e}\ (\mathsf{UNIT}^{\rightsquigarrow})$$

Core-Language Terms: $\Gamma \vdash exp : \mathrm{A} \rightsquigarrow e$

$$\frac{\Gamma \vdash mod : [\![\mathrm{A}]\!]^{+} \rightsquigarrow e}{\Gamma \vdash \mathtt{val}(mod) : \mathrm{A} \rightsquigarrow\, !e}\ (\mathsf{PVAL}^{\rightsquigarrow})$$

$$\frac{\Gamma \vdash mod : \Phi \rightsquigarrow e}{\Gamma \vdash \mathtt{pack}(mod) : \langle\!|\Phi|\!\rangle \rightsquigarrow e}\ (\mathsf{PACK}^{\rightsquigarrow})$$

Fig. 20. Evidence Translation Rules for MixML (continued)

Definition 8.2 (*Well-Typed Judgment*). For any MixML judgment $\vdash \mathcal{J}$ we define:

$$\Delta \vdash \mathcal{J} \quad \stackrel{\text{def}}{\Leftrightarrow} \quad \vdash \mathcal{J} \wedge \mathrm{fv}(\mathcal{J}) \subseteq \mathrm{dom}(\Delta) \wedge \Delta \preceq \mathrm{U}$$

Furthermore, Figure 22 defines well-formed elaboration environments. An environment is well-formed if all contained semantic signatures are well-formed. For well-formed signatures we distinguish between *synthesis* and *analysis* signatures. Intuitively, synthesis signatures are those that can be derived for modules by the typing rules, while analysis signatures are a subset that correspond to signatures the programmer could have written down explicitly. For both, all contained type constructors must be well-formed, and the import locators in contained unit signatures must be well-formed (*i.e.,* fit the signature). For analysis signatures, the export locators in unit signatures must be well-formed as well. Only analysis signatures may be used on the right-hand side of the unit signature matching judgment. Accordingly, all unit signatures describing unit *imports* must be analysis, even inside synthesis signatures. (This is analogous to corresponding definitions for functor signatures and their arguments in previous work, *e.g.,* Dreyer et al. [2003] or Rossberg et al. [2010].) Finally, note that locators $\mathcal{L}$ and realizers $\mathcal{R}$ are special cases of signatures, so the definitions of analysis and synthesis are readily applicable to them.

The following lemma states a number of easy properties about synthesis and analysis signatures and the relation between them:

LEMMA 8.3 (ANALYSIS AND SYNTHESIS SIGNATURES).

(*1*) *If* $\Delta \vdash \Sigma \Downarrow$, *then* $\Delta \vdash \Sigma \Uparrow$.
(*2*) *If* $\Delta \vdash \Phi \Downarrow$, *then* $\Delta \vdash \Phi \Uparrow$.
(*3*) *If* $\Delta \vdash \Sigma \Uparrow$, *then* $\Delta \vdash |\Sigma| \Uparrow$.
(*4*) *If* $\Delta \vdash \Sigma \Downarrow$, *then* $\Delta \vdash -\Sigma \Downarrow$.
(*5*) *If* $\Delta \vdash |\Sigma| \Downarrow$, *then* $\Delta \vdash \Sigma \Uparrow$.
(*6*) *If* $\Delta \vdash \Sigma \Uparrow$ *and* $|\Sigma| = -\Sigma$, *then* $\Delta \vdash \Sigma \Downarrow$.
(*7*) *If* $\Delta \vdash \Sigma \Uparrow$ *and* $\mathcal{R} \subseteq \Sigma$, *then* $\Delta \vdash \mathcal{R} \Uparrow$.

Signature Merging: $\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma \rightsquigarrow f_1/f_2$

$$\frac{\vdash \Sigma_2 + \Sigma_1 \Rightarrow \Sigma \rightsquigarrow f_2/f_1}{\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma \rightsquigarrow f_1/f_2}\ (\mathsf{MSYM}\widetilde{\rightarrow})$$

$$\frac{}{\vdash [\![= \mathrm{A}]\!] + [\![= \mathrm{A}]\!] \Rightarrow [\![= \mathrm{A}]\!] \rightsquigarrow \lambda x.x \ / \ \lambda x.x}\ (\mathsf{MTYP}\widetilde{\rightarrow})$$

$$\frac{\vdash \mathrm{A}_1 \leq \mathrm{A}_2 \rightsquigarrow f}{\vdash [\![\mathrm{A}_1]\!]^{\pm} + [\![\mathrm{A}_2]\!]^{-} \Rightarrow [\![\mathrm{A}_1]\!]^{\pm} \rightsquigarrow \lambda x.x \ / \ \lambda x.\, \mathsf{let}\, x_2 = \mathsf{new}\, \mathrm{A}_2^{\circ}\, \mathsf{in}\, \mathsf{def}\, x_2 := f\,(!\,x);\ x_2}\ (\mathsf{MVAL}\widetilde{\rightarrow})$$

$$\frac{}{\vdash [\![\Phi]\!]^{-} + [\![\Phi]\!]^{-} \Rightarrow [\![\Phi]\!]^{-} \rightsquigarrow \lambda x.x \ / \ \lambda x.x}\ (\mathsf{MUN1}\widetilde{\rightarrow})$$

$$\frac{\vdash \Phi_1 \leq \Phi_2 \rightsquigarrow f}{\vdash [\![\Phi_1]\!]^{+} + [\![\Phi_2]\!]^{-} \Rightarrow [\![\Phi_1]\!]^{+} \rightsquigarrow \lambda x.x \ / \ \lambda x.\, \mathsf{let}\, x_2 = \mathsf{new}\, \Phi_2^{\circ}\, \mathsf{in}\, \mathsf{def}\, x_2 := f\,(!\,x);\ x_2}\ (\mathsf{MUN2}\widetilde{\rightarrow})$$

$$\frac{}{\vdash \Sigma + \{\!|\} \Rightarrow \Sigma \rightsquigarrow \lambda x.x \ / \ \lambda x.\{\}}\ (\mathsf{MEMP}\widetilde{\rightarrow})$$

$$\frac{\ell \notin \overline{\ell_2} \qquad \vdash \{\!|\overline{\ell_1 : \Sigma_1}|\!\} + \{\!|\overline{\ell_2 : \Sigma_2}|\!\} \Rightarrow \{\!|\overline{\ell_3 : \Sigma_3}|\!\} \rightsquigarrow f_1/f_2}{\begin{array}{l}\vdash \{\!|\ell : \Sigma, \overline{\ell_1 : \Sigma_1}|\!\} + \{\!|\overline{\ell_2 : \Sigma_2}|\!\} \Rightarrow \{\!|\ell : \Sigma, \ell_3 : \Sigma_3|\!\} \rightsquigarrow \lambda x.\{f_1\,(x|_{\overline{\ell_3}}),\ \ell = x.\ell\} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad / \ \lambda x.\, f_2\,(x|_{\overline{\ell_3}})\end{array}}\ (\mathsf{MSTR1}\widetilde{\rightarrow})$$

$$\frac{\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma_3 \rightsquigarrow f_1/f_2 \qquad \vdash \{\!|\overline{\ell_1 : \Sigma_1'}|\!\} + \{\!|\overline{\ell_2 : \Sigma_2'}|\!\} \Rightarrow \{\!|\overline{\ell_3 : \Sigma_3'}|\!\} \rightsquigarrow f_1'/f_2'}{\begin{array}{l}\vdash \{\!|\ell : \Sigma_1, \overline{\ell_1 : \Sigma_1'}|\!\} + \{\!|\ell : \Sigma_2, \overline{\ell_2 : \Sigma_2'}|\!\} \Rightarrow \{\!|\ell : \Sigma_3, \overline{\ell_3 : \Sigma_3'}|\!\} \rightsquigarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \lambda x.\{f_1'\,(x|_{\overline{\ell_3}}),\ \ell = f_1\,(x.\ell)\} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad / \ \lambda x.\{f_2'\,(x|_{\overline{\ell_3}}),\ \ell = f_2\,(x.\ell)\}\end{array}}\ (\mathsf{MSTR2}\widetilde{\rightarrow})$$

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

$$\frac{\vdash (\mathcal{L}_1^{-}; \Sigma_1) \rightleftarrows (\mathcal{L}_2^{+}; \Sigma_2) \rightsquigarrow \delta \qquad \vdash \delta\Sigma_1 + -\delta\Sigma_2 \Rightarrow |\Sigma| \rightsquigarrow f_1/f_2}{\begin{array}{l}\vdash \forall\overline{\alpha_1}.\, \exists\overline{\beta_1}.\, (\mathcal{L}_1^{-}; \mathcal{L}_1^{+}; \Sigma_1) \leq \forall\overline{\alpha_2}.\, \exists\overline{\beta_2}.\, (\mathcal{L}_2^{-}; \mathcal{L}_2^{+}; \Sigma_2) \rightsquigarrow \\ \qquad \lambda y : (\forall^{\mathrm{U}}\overline{\alpha_1}.\, \forall^{\mathrm{L}}\overline{\beta_1}.\, (\Sigma_1^{\circ} \to \{\}^{\mathrm{U}}))^{\mathrm{U}}. \\ \qquad \lambda\overline{\alpha_2^{\mathrm{U}}}.\, \lambda\overline{\beta_2^{\mathrm{L}}}.\, \lambda x_2 : \Sigma_2^{\circ}. \\ \qquad \mathsf{new}\, \overline{\beta_1}\, \mathsf{in}\, \mathsf{def}\, \overline{\beta_2 := \delta^{\circ}\beta_2}\, \mathsf{in} \\ \qquad \mathsf{let}\, x = \mathrm{Create}(|\Sigma|)\, \mathsf{in} \\ \qquad \mathrm{Copy}(f_2\, x,\ x_2 : \Sigma_2);\ y\ \overline{\delta^{\circ}\alpha_1}\ \overline{\beta_1}\ (f_1\, x)\end{array}}\ (\mathsf{MATCH}\widetilde{\rightarrow})$$

Fig. 21. Evidence Translation Rules for MixML (continued)

Synthesis Environments: $\vdash \Gamma \Uparrow \quad \vdash (\Gamma; \mathcal{R}; \overline{\beta}) \Uparrow$

$$\frac{}{\vdash \epsilon \Uparrow} \qquad \frac{\vdash \Gamma \Uparrow \quad \vdash \Sigma \Uparrow}{\vdash \Gamma, \mathrm{X} : \Sigma \Uparrow} \qquad \frac{\vdash \Gamma \Uparrow \quad \vdash \mathcal{R} \Uparrow}{\vdash \Gamma; \mathcal{R}; \overline{\beta} \Uparrow}$$

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

$$\frac{\vdash \mathrm{A} \Uparrow \kappa}{\vdash [\![= \mathrm{A}]\!] \Uparrow} \qquad \frac{\vdash \mathrm{A} \Uparrow \mathtt{type}}{\vdash [\![\mathrm{A}]\!]^{\pm} \Uparrow} \qquad \frac{\vdash \Phi \Uparrow}{\vdash [\![\Phi]\!]^{+} \Uparrow} \qquad \frac{\vdash \Phi \Downarrow}{\vdash [\![\Phi]\!]^{-} \Uparrow} \qquad \frac{\overline{\vdash \Sigma \Uparrow}}{\vdash \{\!|\overline{\ell : \Sigma}|\!\} \Uparrow}$$

$$\frac{\vdash \Sigma \Uparrow \quad \vdash \mathcal{L} \text{ locates } \overline{\alpha} \quad \mathcal{L} \subseteq \Sigma}{\vdash \forall \overline{\alpha}. \exists \overline{\beta}. (\mathcal{L}; \Sigma) \Uparrow}$$

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

$$\frac{\vdash \mathrm{A} \Uparrow \kappa}{\vdash [\![= \mathrm{A}]\!] \Downarrow} \qquad \frac{\vdash \mathrm{A} \Uparrow \mathtt{type}}{\vdash [\![\mathrm{A}]\!]^{\pm} \Downarrow} \qquad \frac{\vdash \Phi \Downarrow}{\vdash [\![\Phi]\!]^{\pm} \Downarrow} \qquad \frac{\overline{\vdash \Sigma \Downarrow}}{\vdash \{\!|\overline{\ell : \Sigma}|\!\} \Downarrow}$$

$$\frac{\vdash \Sigma \Downarrow \quad \vdash \mathcal{L}^{-} \text{ locates } \overline{\alpha} \quad \vdash \mathcal{L}^{+} \text{ locates } \overline{\beta} \quad \mathcal{L}^{-} \subseteq \Sigma \quad \mathcal{L}^{+} \subseteq \Sigma}{\vdash \forall \overline{\alpha}. \exists \overline{\beta}. (\mathcal{L}^{-}; \mathcal{L}^{+}; \Sigma) \Downarrow}$$

Fig. 22. Synthesis and Analysis Signatures

(8) *Let* $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$. *If and only if* $\Delta \vdash \mathcal{R} \Uparrow$, *then* $\Delta \vdash \mathcal{R}_1 \Uparrow$ *and* $\Delta \vdash \mathcal{R}_2 \Uparrow$.
(9) $\vdash \mathcal{L} \Downarrow$.
(10) *If* $\Delta \vdash \Sigma \Uparrow$ *and* $\Delta' \vdash \delta^{\circ} : \Delta$, *then* $\Delta' \vdash \delta\Sigma \Uparrow$.
(11) *If* $\Delta \vdash \Sigma \Downarrow$ *and* $\Delta' \vdash \delta^{\circ} : \Delta$, *then* $\Delta' \vdash \delta\Sigma \Downarrow$.
(12) *If* $\Delta \vdash \Gamma \Uparrow$ *and* $\Delta' \vdash \delta^{\circ} : \Delta$, *then* $\Delta' \vdash \delta\Gamma \Uparrow$.

More interestingly, our module typing rules always derive well-formed synthesis signatures (or, in the case of the unit signature judgment, analysis signatures):

LEMMA 8.4 (DERIVED TYPES AND SIGNATURES). *Suppose* $\Delta \vdash (\Gamma; \mathcal{R}; \overline{\beta}) \Uparrow$.

(1) *If* $\Gamma \vdash \mathit{typ} \rightsquigarrow \mathrm{A}$, *then* $\Delta \vdash \mathrm{A} \Uparrow \kappa$.
(2) *If* $\Gamma \vdash \mathit{exp} : \mathrm{A}$, *then* $\Delta \vdash \mathrm{A} \Uparrow \mathtt{type}$.
(3) *If* $\Gamma; \mathcal{R}; \overline{\beta} \vdash \mathit{mod} : \Sigma$, *then* $\Delta \vdash \Sigma \Uparrow$ *and* $\mathcal{R} \subseteq \Sigma$.
(4) *If* $\Gamma \vdash \mathit{mod} : \Sigma$, *then* $\Delta \vdash \Sigma \Uparrow$.
(5) *If* $\Gamma \vdash \mathit{mod} : \Phi$, *then* $\Delta \vdash \Phi \Uparrow$.
(6) *If* $\Gamma \vdash \mathit{usig} \rightsquigarrow \Phi$, *then* $\Delta \vdash \Phi \Downarrow$.
(7) *If* $\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma$, *and* $\Delta \vdash \Sigma_1 \Uparrow$ *and* $\Delta \vdash \Sigma_2 \Uparrow$, *then* $\Delta \vdash \Sigma \Uparrow$.
Also, if $\mathcal{R}_1 \subseteq \Sigma_1$ *and* $\mathcal{R}_2 \subseteq \Sigma_2$, *then* $\mathcal{R}_1 \cup \mathcal{R}_2 \subseteq \Sigma$.

(The properties also hold for the respective static judgments.)

PROOF. By straightforward simultaneous induction on the derivation. The properties 1 and 2 again depend on the core language, and we assume they are provable for

any additional constructs. □

As a next step, the definitions of erasure are sound:

LEMMA 8.5 (PROPERTIES OF ERASURE).

(*1*) $(\delta \mathrm{A})^\circ = \delta^\circ \mathrm{A}^\circ$.
(*2*) $(\delta \Sigma)^\circ = \delta^\circ \Sigma^\circ$.
(*3*) $(\delta \Gamma)^\circ = \delta^\circ \Gamma^\circ$.
(*4*) *If* $\Delta \vdash \Sigma \Uparrow$, *then* $\Sigma^\circ = \tau^\iota$ *with* $\Delta \vdash \tau : \mathsf{type}^{\mathrm{U}}$.
(*5*) $\Sigma^{-\circ} \preceq \mathrm{U}$.
(*6*) $\Gamma^\circ \preceq \mathrm{U}$.
(*7*) $|\Sigma|^{-\circ} = \Sigma^{-\circ}$.
(*8*) $\Sigma^\circ = \Sigma^\circ * \Sigma^{-\circ}$.
(*9*) $\Gamma^\circ = \Gamma^\circ * \Gamma^\circ$.
(*10*) *If* $\Delta \vdash \mathrm{A} \Uparrow \kappa$, *then* $\Delta \vdash \mathrm{A}^\circ : \kappa^{\mathrm{U}}$.
(*11*) *If* $\Delta \vdash \Gamma \Uparrow$, *then* $\Delta \vdash \Gamma^\circ$.
(*12*) *If* $\Delta \vdash (\Gamma; \mathcal{R}; \overline{\beta}) \Uparrow$, *then* $\vdash (\Delta * \overline{\beta^{\mathrm{L}}}; \epsilon; \Gamma^\circ)$.

Here and in the following, we write $\Delta * \overline{\beta^{\mathrm{L}}}$ as shorthand for the environment Δ' that is the same as Δ except that all variables $\overline{\beta}$ are made linear.[9] Some of the properties (1 and 10) depend on additional cases for core-level types, which we assume to be well-defined.

Given that, it is straightforward to show that the auxiliary Create and Copy functions make sense:

LEMMA 8.6 (PROPERTIES OF MODULE CREATION AND COPYING). *Let* $\Delta \vdash \Sigma \Uparrow$.

(*1*) *If* $\Sigma = |\Sigma|$, *then* $\Delta; \epsilon; \epsilon \vdash \mathrm{Create}(\Sigma) : \Sigma^\circ$.
(*2*) *If* $\Delta; \Psi; \Gamma_1 \vdash e_- : (-\Sigma)^\circ$ *and* $\Delta; \Psi; \Gamma_2 \vdash e_+ : \Sigma^\circ$ *and* $\Gamma = \Gamma_1 * \Gamma_2$, *then* $\Delta; \Psi; \Gamma \vdash \mathrm{Copy}(e_-, e_+ : \Sigma) : \{\}^{\mathrm{U}}$.
(*3*) $\mathrm{Create}(\Sigma) = \mathrm{Create}(\delta\Sigma)$.
(*4*) $\mathrm{Copy}(e_-, e_+ : \Sigma) = \mathrm{Copy}(e_-, e_+ : \delta\Sigma)$.

After these preparations, the following property is our main result. It shows that all valid derivations of the evidence translation produce only well-formed LTG types and terms.

THEOREM 8.7 (SOUNDNESS OF TRANSLATION). *Suppose* $\Delta \vdash (\Gamma; \mathcal{R}; \overline{\beta}) \Uparrow$.

(*1*) *If* $\Gamma \vdash \mathit{exp} : \mathrm{A} \rightsquigarrow e$, *then* $\Delta; \epsilon; \Gamma^\circ \vdash e : (\mathrm{A}^\circ)^{\mathrm{U}}$.
(*2*) *If* $\Gamma \vdash \mathit{typ} \rightsquigarrow \mathrm{A}$, *then* $\Delta \vdash \mathrm{A}^\circ : \kappa^{\mathrm{U}}$.
(*3*) *If* $\Gamma; \mathcal{R}; \overline{\beta} \vdash \mathit{mod} : \Sigma \rightsquigarrow e$, *then* $\Delta * \overline{\beta^{\mathrm{L}}}; \epsilon; \Gamma^\circ \vdash e : (\Sigma^\circ \to \{\}^{\mathrm{U}})^\iota$, *where* $\iota = \mathrm{U}$ *iff* $\overline{\beta}$ *is empty, and* $\iota = \mathrm{L}$ *otherwise.*
(*4*) *If* $\Gamma \vdash \mathit{mod} : \Sigma \rightsquigarrow e$, *then* $\Delta; \epsilon; \Gamma^\circ \vdash e : \Sigma^{-\circ}$.
(*5*) *If* $\Gamma \vdash \mathit{mod} : \Phi \rightsquigarrow e$, *then* $\Delta; \epsilon; \Gamma^\circ \vdash e : (\Phi^\circ)^{\mathrm{U}}$.
(*6*) *If* $\Gamma \vdash \mathit{usig} \rightsquigarrow \Phi$, *then* $\Delta \vdash \Phi^\circ : \mathsf{type}^{\mathrm{U}}$.
(*7*) *If* $\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma \rightsquigarrow f_1/f_2$ *and* $\Delta \vdash \Sigma_1 \Uparrow$ *and* $\Delta \vdash \Sigma_2 \Uparrow$, *then* $\Sigma^\circ = {\Sigma'_1}^\circ * {\Sigma'_2}^\circ$ *with* $\Delta; \epsilon; \epsilon \vdash f_1 : {\Sigma'_1}^\circ \to \Sigma_1^\circ$ *and* $\Delta; \epsilon; \epsilon \vdash f_2 : {\Sigma'_2}^\circ \to \Sigma_2^\circ$.
(*8*) *If* $\vdash \mathrm{A}_1 \leq \mathrm{A}_2 \rightsquigarrow f$ *and* $\Delta \vdash \mathrm{A}_1 \Uparrow \mathsf{type}$, $\Delta \vdash \mathrm{A}_2 \Uparrow \mathsf{type}$, *then* $\Delta; \epsilon; \epsilon \vdash f : (\mathrm{A}_1^\circ)^{\mathrm{U}} \to (\mathrm{A}_2^\circ)^{\mathrm{U}}$.
(*9*) *If* $\vdash \Phi_1 \leq \Phi_2 \rightsquigarrow f$ *and* $\Delta \vdash \Phi_1 \Uparrow$ *and* $\Delta \vdash \Phi_2 \Downarrow$, *then* $\Delta; \epsilon; \epsilon \vdash f : (\Phi_1^\circ)^{\mathrm{U}} \to (\Phi_2^\circ)^{\mathrm{U}}$.

[9] Recall that splitting for type environments does not actually allow dropping bindings on either side, so because $\mathrm{dom}(\Delta) \supseteq \overline{\beta}$, this is not a "proper" split, but a slightly generalized notation.

(*10*) *If* $\vdash (\mathcal{L}_1;\Sigma_1) \rightleftarrows (\mathcal{L}_2;\Sigma_2) \rightsquigarrow \delta$ *and* $\Delta, \overline{\alpha_1^{\mathrm{U}}}, \overline{\alpha_2^{\mathrm{U}}} \vdash \Sigma_1 \Uparrow$ *and* $\Delta, \overline{\alpha_1^{\mathrm{U}}}, \overline{\alpha_2^{\mathrm{U}}} \vdash \Sigma_2 \Uparrow$ *and* $\vdash \mathcal{L}_1$ locates $\overline{\alpha_1}$ *and* $\vdash \mathcal{L}_2$ locates $\overline{\alpha_2}$ *and* $\overline{\alpha}_1, \overline{\alpha}_2, \overline{\beta}$ *disjoint, then* $\mathrm{dom}(\delta) = \{\overline{\alpha}_1, \overline{\alpha}_2\}$ *with* $\Delta * \overline{\beta^{\mathrm{L}}} \vdash \delta^\circ : (\Delta * \overline{\beta^{\mathrm{L}}}), \overline{\alpha_1^{\mathrm{U}}}, \overline{\alpha_2^{\mathrm{U}}}$.

PROOF. By induction on the derivations, relying on the previous lemmas, and on Lemma 7.15 (Substitution Reversal) for the rules $\mathsf{SEAL}^{\rightsquigarrow}$ and $\mathsf{MATCH}^{\rightsquigarrow}$, which define type names in correspondence to a substitution that the rules compute. We give the details of the interesting cases in Appendix A. Once more, the arguments for cases 1, 2 and 8 depend on the core language, and we assume they are provable for any additional constructs. □

9. ALGORITHMIC TYPE-CHECKING AND DECIDABILITY

Let us recapitulate: we have a type system for MixML, we have an operational semantics (by means of translation into an internal language), and we have a proof that the type system is sound under this semantics. However, as is standard practice, the type system that we have given is merely a declarative specification. Such a formulation has certain advantages—for example, it is relatively easy to understand and to prove correct. But it does not necessarily suggest an obvious *algorithm* for deciding whether a given program is actually well-formed. For a language's practical implementation in a compiler, the existence of such an algorithm is clearly important.

The MixML typing rules are syntax-directed, so for the most part, they can already be read as a recursive algorithm taking a typing context and the program to check as input, and producing a respective type as output (the elaboration rules additionally produce a term). For example, in order to type-check the module expression $mod.\ell$ under a given context $\Gamma;\mathcal{R};\overline{\beta}$, we need to use rule DOT. Consequently, we recursively type-check mod under the modified context $\Gamma;\{\!|\ell : \mathcal{R}|\!\};\overline{\beta}$ and verify that the result is a signature of the form $\{\!|\ell : \Sigma, \ldots|\!\}$, so that we can extract the desired Σ. Likewise for most other constructs.

However, on closer inspection of all the rules, we find two relevant sources of nondeterminism that seemingly require appropriate guesses to proceed successfully:

(1) In rule EVAL, in the static pass, the type A classifying the exported term has to be chosen without looking at the term.
(2) In rules LINK, SEAL, COMPL, and UNIT, new locators $\mathcal{L}$ and/or export type names $\overline{\beta}$ need to be chosen for the premises. (Moreover, the input realizer and export variables have to be split up into suitable disjoint parts in rules LINK and SEAL.)

In this section, we are going to show that in all these instances the right guesses can, in fact, be made algorithmically. By plugging the decision procedures we give into the existing typing rules, we then have a deterministic algorithm for type-checking MixML. As a corollary, we will furthermore find that the types assigned by the MixML type system are unique.

Assumptions about the core language. Once more, we need to make appropriate assumptions about the definitions of the core language judgments that are plugged into our type system (cf. Section 4.1). Concretely, we assume that deterministic algorithms exist for checking these judgments in mutual recursion with module type-checking. The details are given with the respective Theorems 9.2, 9.8, and 9.9 below.

9.1. The Static Pass

Let us first address the issue of guessing a type A in rule EVAL during the static pass (Point 1 above).

Modules: $\Gamma; \mathcal{R}; \overline{\beta} \models_{\overline{\mathrm{stat}}\bot} mod : \Sigma$

$$\frac{}{\Gamma; \{\!|\}; \emptyset \models_{\overline{\mathrm{stat}}\bot} \texttt{[}exp\texttt{]} : [\![\bot]\!]^+} \ (\textsc{eval-det})$$

Fig. 23. Deterministic Typing Rule for the Static Pass

The role of the static pass is to compute the *static* components of a signature, before performing the full type-check. That is, its only purpose is to collect the type exports needed for the type lookup in rules LINK and SEAL. We don't really care about the dynamic components of the signature during the static pass because they are irrelevant for type lookup.

That observation gives us some leeway regarding the concrete choice of A in rule EVAL during the static pass, because it only describes a *dynamic* component. In fact, it does not matter at all what type we pick in any static instance of that rule, as long as it still allows a successful completion of the static pass! The choice can never affect any static component.

But what is a sufficient condition for making the pass succeed? Inspecting the rules of our system, we see that the only two rules that might be affected by a wrong choice for atomic term export signatures $[\![\mathrm{A}]\!]^+$ are rule MVAL for merging them, where the subtyping premise might be violated, and rule COMPL for complete modules, where an export contained in Σ might capture a local type variable from $\overline{\beta}$ and thus violate the side condition about $\mathrm{fv}(\Sigma)$. The only other rule that cares about atomic term exports is the term-level rule PVAL, and that will not be used in the static pass, because all uses of the term typing judgment are shaded.

Intuitively, then, the answer is: any *subtype* of the A derived in the main pass will suffice for the static pass, as long as it doesn't have additional free variables. In particular, a canonical choice would be the bottom type (which is closed and is a subtype of all types), if the core language type system provides such a type. For example, in ML, the polymorphic type scheme $\forall\alpha.\alpha$ would be appropriate. But even if the language does *not* already provide such a type, we can easily add it *pro forma*, just for the purpose of module type-checking: since we don't type-check expressions in the static pass, the only place where it will actually interact with proper core types is in the (static) merging rule for atomic term signatures. It will never escape to, or otherwise show up in, the main judgment.

For this purpose, assume that $\bot$ denotes a closed core type for which the subsumption $\vdash \bot \leq \mathrm{A}$ holds for all A. Let $\models_{\overline{\mathrm{stat}}\bot}$ stand for a variant of the static judgment that is the same as $\models_{\overline{\mathrm{stat}}}$, except that rule EVAL is replaced by the deterministic rule EVAL-DET shown in Figure 23.

In order to prove that the $\models_{\overline{\mathrm{stat}}\bot}$ judgment produces the same results as the original $\models_{\overline{\mathrm{stat}}}$ judgment—*i.e.,* computes a Σ with the same static components—we first have to define a simple notion of *signature approximation*, given in Figure 24. It expresses the relation between the signatures derived by $\models_{\overline{\mathrm{stat}}\bot}$ and the ones derived by $\models_{\overline{\mathrm{stat}}}$, *i.e.,* a signature Σ_1 approximates Σ_2, written $\Sigma_1 \preceq \Sigma_2$, if they coincide on all their components except for term exports, which may vary through subtyping. We extend the relation pointwise to environments Γ.

Here, we collect a number of straightforward properties of signature approximation:

LEMMA 9.1 (PROPERTIES OF SIGNATURE APPROXIMATION).

(*1*) $\Sigma \preceq \Sigma$.
(*2*) *If* $\Sigma_1 \preceq \Sigma_2$, *then* $|\Sigma_1| \preceq |\Sigma_2|$.

$$
\begin{array}{rcll}
[\![= \mathrm{A}]\!] & \preceq & [\![= \mathrm{A}]\!] & \\
[\![\mathrm{A}]\!]^- & \preceq & [\![\mathrm{A}]\!]^- & \\
[\![\mathrm{A}_1]\!]^+ & \preceq & [\![\mathrm{A}_2]\!]^+ & \text{if } \vdash \mathrm{A}_1 \leq \mathrm{A}_2 \text{ and } \mathrm{fv}(\mathrm{A}_1) \subseteq \mathrm{fv}(\mathrm{A}_2) \\
[\![\Phi]\!]^- & \preceq & [\![\Phi]\!]^- & \\
[\![\Phi_1]\!]^+ & \preceq & [\![\Phi_2]\!]^+ & \text{if } \Phi_1 \preceq \Phi_2 \\
\{\!|\overline{\ell : \Sigma_1}|\!\} & \preceq & \{\!|\overline{\ell : \Sigma_2}|\!\} & \text{if } \overline{\Sigma_1 \preceq \Sigma_2} \\
\forall \overline{\alpha}.\, \exists \overline{\beta}.\, (\mathcal{L}_1; \mathcal{L}_2; \Sigma_1) & \preceq & \forall \overline{\alpha}.\, \exists \overline{\beta}.\, (\mathcal{L}_1; \mathcal{L}_2; \Sigma_2) & \text{if } \Sigma_1 \preceq \Sigma_2
\end{array}
$$

Fig. 24. Signature Approximation

(3) *If* $\Sigma_1 \preceq \Sigma_2$*, then* $\delta\Sigma_1 \preceq \delta\Sigma_2$.
(4) *If* $\Sigma_1 \preceq \Sigma_2$*, then* $\mathrm{fv}(\Sigma_1) \subseteq \mathrm{fv}(\Sigma_2)$.
(5) *If* $\Sigma_1 \preceq \Sigma_2$ *and* $|\Sigma_2| = -\Sigma_2$*, then* $\Sigma_1 = \Sigma_2$.
(6) *If* $\Sigma_1 \preceq \Sigma_2$*, then* $\mathrm{dom}(\Sigma_1) = \mathrm{dom}(\Sigma_2)$ *and* $\forall \ell s \in \mathrm{dom}(\Sigma_1)$, $\Sigma_1(\ell s) = \Sigma_2(\ell s)$.

The last property is the most interesting one, because it implies that looking up types from an approximation of a signature Σ will have the same result as looking up those types from Σ itself (recall the definitions for $\Sigma(\ell s)$ and $\mathrm{dom}(\Sigma)$ from Figure 4, which only consider type components). That is the property we ultimately rely on when we want to implement the non-deterministic static pass with a deterministic one. The next theorem and its corollary use this property:

THEOREM 9.2 (DETERMINISTIC SIGNATURE APPROXIMATION).
Assume $\Gamma' \preceq \Gamma$ *and* $\Sigma'_1 \preceq \Sigma_1$ *and* $\Sigma'_2 \preceq \Sigma_2$.

(1) *If* $\Gamma \vdash_{\mathrm{stat}} \mathit{typ} \rightsquigarrow \mathrm{A}$*, then* $\Gamma' \vdash_{\mathrm{stat}\bot} \mathit{typ} \rightsquigarrow \mathrm{A}$.
(2) *If* $\Gamma; \mathcal{R}; \overline{\beta} \vdash_{\mathrm{stat}} \mathit{mod} : \Sigma$*, then* $\Gamma'; \mathcal{R}; \overline{\beta} \vdash_{\mathrm{stat}\bot} \mathit{mod} : \Sigma'$ *with* $\Sigma' \preceq \Sigma$.
(3) *If* $\Gamma \vdash_{\mathrm{stat}} \mathit{mod} : \Sigma$*, then* $\Gamma' \vdash_{\mathrm{stat}\bot} \mathit{mod} : \Sigma'$ *with* $\Sigma' \preceq \Sigma$.
(4) *If* $\Gamma \vdash_{\mathrm{stat}} \mathit{mod} : \Phi$*, then* $\Gamma' \vdash_{\mathrm{stat}\bot} \mathit{mod} : \Phi'$ *with* $\Phi' \preceq \Phi$.
(5) *If* $\Gamma \vdash_{\mathrm{stat}} \mathit{usig} \rightsquigarrow \Phi$*, then* $\Gamma' \vdash_{\mathrm{stat}\bot} \mathit{usig} \rightsquigarrow \Phi$.
(6) *If* $\vdash_{\mathrm{stat}} \Sigma_1 + \Sigma_2 \Rightarrow \Sigma$*, then* $\vdash_{\mathrm{stat}\bot} \Sigma'_1 + \Sigma'_2 \Rightarrow \Sigma'$ *with* $\Sigma' \preceq \Sigma$.
(7) *If* $\vdash (\mathcal{L}_1; \Sigma_1) \rightleftarrows (\mathcal{L}_2; \Sigma_2) \rightsquigarrow \delta$*, then* $\vdash (\mathcal{L}_1; \Sigma'_1) \rightleftarrows (\mathcal{L}_2; \Sigma'_2) \rightsquigarrow \delta$.

PROOF. By easy induction on the derivation. Once more, Part 1 depends on the details of the core language, and we assume it provable for any additional cases. □

COROLLARY 9.3 (COMPLETENESS OF DETERMINISTIC STATIC JUDGMENTS).
If $\Gamma; \mathcal{R}; \overline{\beta} \vdash_{\mathrm{stat}} \mathit{mod} : \Sigma$*, then* $\Gamma; \mathcal{R}; \overline{\beta} \vdash_{\mathrm{stat}\bot} \mathit{mod} : \Sigma'$*, such that* $\mathrm{dom}(\Sigma) = \mathrm{dom}(\Sigma')$ *and* $\forall \ell s \in \mathrm{dom}(\Sigma)$, $\Sigma(\ell s) = \Sigma'(\ell s)$.

The inverse is trivial, because $\vdash_{\mathrm{stat}\bot}$ is just a restriction of $\vdash_{\mathrm{stat}}$:

THEOREM 9.4 (SOUNDNESS OF DETERMINISTIC STATIC JUDGMENTS).
If $\Gamma; \mathcal{R}; \overline{\beta} \vdash_{\mathrm{stat}\bot} \mathit{mod} : \Sigma$*, then* $\Gamma; \mathcal{R}; \overline{\beta} \vdash_{\mathrm{stat}} \mathit{mod} : \Sigma$.

With the last two properties together, we are free to replace $\vdash_{\mathrm{stat}}$ with $\vdash_{\mathrm{stat}\bot}$, and can hence cope with the first source of non-determinism in the MixML typing rules.

9.2. Templates

Dealing with the other source of non-determinism—the up-front choice of locators and abstract type names for the context (Point 2 above)—requires a bit more work.

Locators and abstract type names represent the type imports and exports, respectively, of a module mod. Both are consumed by the typing rules in a deterministic, linear fashion (locators—potentially refined to realizers on the way—in rule ITYP, and abstract type names in rule SEAL, or both simultaneously in rules NEW or UNPACK). It should hence be possible to predict suitable choices beforehand by just looking at the structure of mod, and these choices should be unique up to renaming. However, it is not entirely obvious that we can do this. For example, consider the module

$$\{\mathrm{X} = [mod], \mathrm{Y} = [: usig], \mathrm{Z} = \texttt{new}\ \mathrm{Y}\ \texttt{with new}\ \mathrm{X}\}$$

Without actually type-checking mod and $usig$, how can we tell the imports or exports in the module's Z component?

Obviously, we need some amount of inference. The central insight, however, is that it is enough to infer the "shape" of a module's signature. Such a shape does not need to contain any concrete type information—knowing the kinds of type imports and exports is enough. Computing a shape hence does not require full type-checking.

Making the notion of shape precise, Figure 25 defines *template signatures* $\mathbb{S}$, which are essentially semantic signatures with all type information, up to kinds, erased (term imports and exports are erased completely). In the same style, *template type locators* $\mathbb{L}$ are defined. The figure also defines *template erasure* $(\_)^{\mathbb{T}}$, which maps semantic objects into corresponding template objects, plus other meta-notation that corresponds to the respective operations on semantic signatures (cf. Figures 4 and 8).

Template Computation. The template signature of a module expression mod can be computed by a simple recursive pre-pass over mod. Figures 26–27 specify the algorithm. Given a module and a template context (a template-erased version of a full typing environment Γ) it returns a template signature $\mathbb{S}$, a template type locator $\mathbb{L}$, and a list of kinds for the export type names of the module—without choosing actual names. The algorithm makes use of the meta-notation defined in Figure 25. (Note that, unlike the similar notation for proper locators, field removal $\mathbb{L}\setminus\overline{\ell s}$ is defined on templates even in the case where $\overline{\ell s}$ contains paths not defined in $\mathbb{L}$—this provides some convenience in rule $\mathsf{LINK}^{\mathbb{T}}$.)

The template locators and export kinds computed by these rules mirror the locators and variables occurring in the context of the corresponding typing rules from Figure 5. In addition to computing locators and export kinds, the template signatures $\mathbb{S}$ produced by the algorithm also keep track of (the templates of) atomic unit signatures defined in the module. This knowledge is necessary for dealing with examples like the one given above, where `new` is applied to local units. It is used in rule $\mathsf{NEW}^{\mathbb{T}}$, accordingly.

The definition of template signature merging is given in Figure 25. It mirrors the merging judgment on proper signatures. Signature merging is the sole reason that we need to track polarity of atomic unit signatures in templates: during template computation we cannot check unit signature matching, so the definition of merging for atomic unit signatures may be based solely on polarity—this is where the need for the restriction on unit merging discussed in Section 5.4 becomes manifest in the type-checking algorithm.

Completeness. Our algorithm for template computation looks sufficiently straightforward. Some work remains, though, in order to actually prove it complete.

The following lemma states some easy facts about template erasure and its interaction with the other meta-operations:

LEMMA 9.5 (PROPERTIES OF TEMPLATES).

(*1*) $\mathcal{R}^{\mathbb{T}} = \mathbb{L}$ *for some* $\mathbb{L}$.
(*2*) $(-\Sigma)^{\mathbb{T}} = -\Sigma^{\mathbb{T}}$.

$$
\begin{array}{rrl}
\textbf{Module Templates} & \mathbb{S} & ::= [\![\kappa]\!] \mid [\![\mathbb{F}]\!]^{\pm} \mid \{\!|\overline{\ell : \mathbb{S}}|\!\} \\
\textbf{Locator Templates} & \mathbb{L} & ::= [\![\kappa]\!] \mid \{\!|\overline{\ell : \mathbb{L}}|\!\} \\
\textbf{Unit Templates} & \mathbb{F} & ::= \mathbb{L}; \overline{\kappa}; \mathbb{S}
\end{array}
$$

$$
\begin{array}{rcl}
\mathbb{S}.\ell s & \stackrel{\text{def}}{=} & \begin{cases} \mathbb{S} & \text{if } \ell s = \epsilon \\ \mathbb{S}' & \text{if } \ell s = \ell s'.\ell \text{ and } \mathbb{S}.\ell s' = \{\!|\ell : \mathbb{S}', \ldots|\!\} \end{cases} \\
\mathbb{S}(\ell s) & \stackrel{\text{def}}{=} & \kappa \text{ if } \mathbb{S}.\ell s = [\![\kappa]\!] \\
\mathrm{dom}(\mathbb{S}) & \stackrel{\text{def}}{=} & \{\ell s \mid \mathbb{S}(\ell s) = \kappa\} \\
\mathbb{L} \subseteq \mathbb{S} & \stackrel{\text{def}}{\Leftrightarrow} & \forall\, \ell s \in \mathrm{dom}(\mathbb{L}).\ \mathbb{L}(\ell s) = \mathbb{S}(\ell s) \\
\mathbb{L} \# \mathbb{S} & \stackrel{\text{def}}{\Leftrightarrow} & \mathrm{dom}(\mathbb{L}) \cap \mathrm{dom}(\mathbb{S}) = \emptyset \\
\mathbb{L}_1 \uplus \mathbb{L}_2 & \stackrel{\text{def}}{=} & \mathbb{L} \text{ such that } \mathrm{dom}(\mathbb{L}) = \mathrm{dom}(\mathbb{L}_1) \uplus \mathrm{dom}(\mathbb{L}_2) \\
 & & \text{and } \forall \ell s \in \mathrm{dom}(\mathbb{L}).\ \mathbb{L}(\ell s) = \mathbb{L}_1(\ell s) \vee \mathbb{L}(\ell s) = \mathbb{L}_2(\ell s)
\end{array}
$$

$$
\begin{array}{rcl}
[\![= \mathrm{A}]\!]^{\mathbb{T}} & \stackrel{\text{def}}{=} & [\![\kappa]\!] \quad \text{where } \vdash \mathrm{A} \Uparrow \kappa \\
([\![\mathrm{A}]\!]^{\pm})^{\mathbb{T}} & \stackrel{\text{def}}{=} & \{\!|\}\!\} \\
([\![\Phi]\!]^{\pm})^{\mathbb{T}} & \stackrel{\text{def}}{=} & [\![\Phi^{\mathbb{T}}]\!]^{\pm} \\
\{\!|\overline{\ell : \Sigma}|\!\}^{\mathbb{T}} & \stackrel{\text{def}}{=} & \{\!|\overline{\ell : \Sigma^{\mathbb{T}}}|\!\} \\
(\forall \overline{\alpha}.\, \exists \overline{\beta}.\, (\mathcal{L}_1; \mathcal{L}_2; \Sigma))^{\mathbb{T}} & \stackrel{\text{def}}{=} & \mathcal{L}_1^{\mathbb{T}}; \overline{\kappa_\beta}; \Sigma^{\mathbb{T}} \\
\epsilon^{\mathbb{T}} & \stackrel{\text{def}}{=} & \epsilon \\
(\Gamma, \mathrm{X} : \Sigma)^{\mathbb{T}} & \stackrel{\text{def}}{=} & \Gamma^{\mathbb{T}}, \mathrm{X} : \Sigma^{\mathbb{T}}
\end{array}
$$

$$
\begin{array}{rcl}
|[\![\kappa]\!]| & \stackrel{\text{def}}{=} & [\![\kappa]\!] \\
|[\![\mathbb{F}]\!]^{\pm}| & \stackrel{\text{def}}{=} & [\![\mathbb{F}]\!]^{+} \\
|\{\!|\overline{\ell : \mathbb{S}}|\!\}| & \stackrel{\text{def}}{=} & \{\!|\overline{\ell : |\mathbb{S}|}|\!\}
\end{array}
\qquad
\begin{array}{rcl}
-[\![\kappa]\!] & \stackrel{\text{def}}{=} & [\![\kappa]\!] \\
-[\![\mathbb{F}]\!]^{\pm} & \stackrel{\text{def}}{=} & [\![\mathbb{F}]\!]^{\mp} \\
-\{\!|\overline{\ell : \mathbb{S}}|\!\} & \stackrel{\text{def}}{=} & \{\!|\overline{\ell : -\mathbb{S}}|\!\}
\end{array}
$$

$$
\begin{array}{rcl}
|\mathbb{S}|_{\epsilon} & \stackrel{\text{def}}{=} & |\mathbb{S}| \\
|\{\!|\ell : \mathbb{S}, \overline{\ell' : \mathbb{S}'}|\!\}|_{\ell.\ell s} & \stackrel{\text{def}}{=} & \{\!|\ell : |\mathbb{S}|_{\ell s}, \overline{\ell' : \mathbb{S}'}|\!\} \\
|\mathbb{S}|_{\ell s_1, \ldots, \ell s_n} & \stackrel{\text{def}}{=} & |\ldots|\mathbb{S}|_{\ell s_1} \ldots|_{\ell s_n}
\end{array}
\qquad
\begin{array}{rcl}
\mathbb{L} \setminus \epsilon & \stackrel{\text{def}}{=} & \{\!|\}\!\} \\
\{\!|\ell : \mathbb{L}, \overline{\ell' : \mathbb{L}'}|\!\} \setminus \ell.\ell s & \stackrel{\text{def}}{=} & \{\!|\ell : \mathbb{L} \setminus \ell s, \overline{\ell' : \mathbb{L}'}|\!\} \\
\{\!|\overline{\ell' : \mathbb{L}'}|\!\} \setminus \ell.\ell s & \stackrel{\text{def}}{=} & \{\!|\overline{\ell' : \mathbb{L}'}|\!\} \text{ if } \ell \notin \overline{\ell'} \\
\mathbb{L} \setminus \ell s_1, \ldots, \ell s_n & \stackrel{\text{def}}{=} & \mathbb{L} \setminus \ell s_1 \ldots \setminus \ell s_n
\end{array}
$$

$$
\begin{array}{rcl}
\mathbb{S}_1 + \mathbb{S}_2 & \stackrel{\text{def}}{=} & \mathbb{S}_2 + \mathbb{S}_1 \\
[\![\kappa]\!] + [\![\kappa]\!] & \stackrel{\text{def}}{=} & [\![\kappa]\!] \\
[\![\mathbb{F}]\!]^{-} + [\![\mathbb{F}]\!]^{-} & \stackrel{\text{def}}{=} & [\![\mathbb{F}]\!]^{-} \\
[\![\mathbb{F}_1]\!]^{+} + [\![\mathbb{F}_2]\!]^{-} & \stackrel{\text{def}}{=} & [\![\mathbb{F}_1]\!]^{+} \\
\mathbb{S} + \{\!|\}\!\} & \stackrel{\text{def}}{=} & \mathbb{S} \\
\{\!|\overline{\ell : \mathbb{S}_1}, \overline{\ell_1 : \mathbb{S}_1'}|\!\} + \{\!|\overline{\ell : \mathbb{S}_2}, \overline{\ell_2 : \mathbb{S}_2'}|\!\} & \stackrel{\text{def}}{=} & \{\!|\overline{\ell : \mathbb{S}_1 + \mathbb{S}_2}, \overline{\ell_1 : \mathbb{S}_1'}, \overline{\ell_2 : \mathbb{S}_2'}|\!\} \quad \text{where } \overline{\ell_1} \cap \overline{\ell_2} = \emptyset
\end{array}
$$

Fig. 25. Signature Templates

Type Constructor Templates: $\Gamma^{\mathbb{T}} \vdash typ \Rightarrow \kappa$

$$\frac{\Gamma^{\mathbb{T}} \vdash mod \Rightarrow \{\!|\}; \overline{\kappa}; [\![\kappa']\!]}{\Gamma^{\mathbb{T}} \vdash \mathtt{typ}(mod) \Rightarrow \kappa'}\ (\mathsf{PTYP}^{\mathbb{T}}) \qquad \frac{}{\Gamma^{\mathbb{T}} \vdash \mathtt{pack}(usig) \Rightarrow \mathsf{type}}\ (\mathsf{PACKAGE}^{\mathbb{T}})$$

$$\frac{}{\Gamma^{\mathbb{T}} \vdash \alpha \Rightarrow \kappa_\alpha}\ (\mathsf{TVAR}^{\mathbb{T}}) \qquad \frac{\Gamma^{\mathbb{T}} \vdash typ \Rightarrow \kappa}{\Gamma^{\mathbb{T}} \vdash \lambda\alpha.typ \Rightarrow \kappa_\alpha \to \kappa}\ (\mathsf{LAM}^{\mathbb{T}}) \qquad \frac{\Gamma^{\mathbb{T}} \vdash typ_1 \Rightarrow \kappa_2 \to \kappa}{\Gamma^{\mathbb{T}} \vdash typ_1\ typ_2 \Rightarrow \kappa}\ (\mathsf{APP}^{\mathbb{T}})$$

Module Templates: $\Gamma^{\mathbb{T}} \vdash mod \Rightarrow \mathbb{F}$

$$\frac{\mathrm{X} : |\mathbb{S}| \in \Gamma^{\mathbb{T}}}{\Gamma^{\mathbb{T}} \vdash \mathrm{X} \Rightarrow \{\!|\}; \emptyset; |\mathbb{S}|}\ (\mathsf{VAR}^{\mathbb{T}}) \qquad \frac{}{\Gamma^{\mathbb{T}} \vdash \{\} \Rightarrow \{\!|\}; \emptyset; \{\!|\}}\ (\mathsf{EMP}^{\mathbb{T}})$$

$$\frac{}{\Gamma^{\mathbb{T}} \vdash \mathtt{[:}knd\mathtt{]} \Rightarrow [\![knd]\!]; \emptyset; [\![knd]\!]}\ (\mathsf{ITYP}^{\mathbb{T}}) \qquad \frac{\Gamma^{\mathbb{T}} \vdash typ \Rightarrow \kappa}{\Gamma^{\mathbb{T}} \vdash \mathtt{[}typ\mathtt{]} \Rightarrow \{\!|\}; \emptyset; [\![\kappa]\!]}\ (\mathsf{ETYP}^{\mathbb{T}})$$

$$\frac{}{\Gamma^{\mathbb{T}} \vdash \mathtt{[:}typ\mathtt{]} \Rightarrow \{\!|\}; \emptyset; \{\!|\}}\ (\mathsf{IVAL}^{\mathbb{T}}) \qquad \frac{}{\Gamma^{\mathbb{T}} \vdash \mathtt{[}exp\mathtt{]} \Rightarrow \{\!|\}; \emptyset; \{\!|\}}\ (\mathsf{EVAL}^{\mathbb{T}})$$

$$\frac{\Gamma^{\mathbb{T}} \vdash mod \Rightarrow \mathbb{L}; \overline{\kappa}; \mathbb{S}}{\Gamma^{\mathbb{T}} \vdash \{\ell = mod\} \Rightarrow \{\!|\ell : \mathbb{L}|\!\}; \overline{\kappa}; \{\!|\ell : \mathbb{S}|\!\}}\ (\mathsf{STR}^{\mathbb{T}}) \qquad \frac{\Gamma^{\mathbb{T}} \vdash mod \Rightarrow \{\!|\ell : \mathbb{L}|\!\}; \overline{\kappa}; \{\!|\ell : \mathbb{S}, \overline{\ell' : \mathbb{S}'}|\!\}}{\Gamma^{\mathbb{T}} \vdash mod.\ell \Rightarrow \mathbb{L}; \overline{\kappa}; \mathbb{S}}\ (\mathsf{DOT}^{\mathbb{T}})$$

$$\frac{\Gamma^{\mathbb{T}} \vdash mod_1 \Rightarrow \mathbb{L} \uplus \mathbb{L}_1; \overline{\kappa}_1; \mathbb{S}_1 \qquad \Gamma^{\mathbb{T}}, \mathrm{X} : |\mathbb{S}_1| \vdash mod_2 \Rightarrow \mathbb{L} \uplus \mathbb{L}_2; \overline{\kappa}_2; \mathbb{S}_2 \qquad \mathbb{L}_1 \# \mathbb{L}_2}{\Gamma^{\mathbb{T}} \vdash (\mathrm{X} = mod_1)\ \mathtt{with}\ mod_2 \Rightarrow \mathbb{L} \uplus \mathbb{L}_1 \backslash \mathrm{dom}(\mathbb{S}_2) \uplus \mathbb{L}_2 \backslash \mathrm{dom}(\mathbb{S}_1); \overline{\kappa}_1, \overline{\kappa}_2; \mathbb{S}_1 + \mathbb{S}_2}\ (\mathsf{LINK}^{\mathbb{T}})$$

$$\frac{\Gamma^{\mathbb{T}} \vdash mod_1 \Rightarrow \mathbb{L}; \overline{\kappa}; \mathbb{S} \qquad \vdash \mathbb{L}\ \mathsf{locates}\ \overline{\kappa}'}{\Gamma^{\mathbb{T}} \vdash (\mathrm{X} = mod_1)\ \mathtt{seals}\ mod_2 \Rightarrow \{\!|\}; \overline{\kappa}, \overline{\kappa}'; |\mathbb{S}|}\ (\mathsf{SEAL}^{\mathbb{T}})$$

$$\frac{\Gamma^{\mathbb{T}} \vdash usig \Rightarrow \mathbb{F}}{\Gamma^{\mathbb{T}} \vdash \mathtt{[:}usig\mathtt{]} \Rightarrow \{\!|\}; \emptyset; [\![\mathbb{F}]\!]^-}\ (\mathsf{IUN}^{\mathbb{T}}) \qquad \frac{\Gamma^{\mathbb{T}} \vdash mod \Rightarrow \mathbb{F}}{\Gamma^{\mathbb{T}} \vdash \mathtt{[}mod\mathtt{]} \Rightarrow \{\!|\}; \emptyset; [\![\mathbb{F}]\!]^+}\ (\mathsf{EUN}^{\mathbb{T}})$$

$$\frac{\Gamma^{\mathbb{T}} \vdash mod \Rightarrow \{\!|\}; \overline{\kappa}; [\![\mathbb{F}]\!]^+}{\Gamma^{\mathbb{T}} \vdash \mathtt{new}\ mod \Rightarrow \mathbb{F}}\ (\mathsf{NEW}^{\mathbb{T}}) \qquad \frac{\Gamma^{\mathbb{T}} \vdash usig \Rightarrow \mathbb{F}}{\Gamma^{\mathbb{T}} \vdash \mathtt{unpack}(exp\ \mathtt{as}\ usig) \Rightarrow \mathbb{F}}\ (\mathsf{UNPACK}^{\mathbb{T}})$$

Unit Signature Templates: $\Gamma^{\mathrm{S}} \vdash usig \Rightarrow \mathbb{F}$

$$\frac{\Gamma^{\mathbb{T}} \vdash mod \Rightarrow \mathbb{L}; \emptyset; \mathbb{S} \qquad \mathbb{L} = \mathbb{L}_1 \uplus \mathbb{L}_2 \qquad \mathbb{L}_1 = \mathbb{L} \setminus \overline{\ell s} \qquad \vdash \mathbb{L}_2\ \mathsf{locates}\ \overline{\kappa}}{\Gamma^{\mathbb{T}} \vdash mod\ \mathtt{export}\ \overline{\ell s} \Rightarrow \mathbb{L}_1; \overline{\kappa}; |\mathbb{S}|_{\overline{\ell s}}}\ (\mathsf{EXPORT}^{\mathbb{T}})$$

$$\frac{\Gamma^{\mathbb{T}} \vdash mod \Rightarrow \mathbb{L}; \emptyset; \mathbb{S} \qquad \mathbb{L} = \mathbb{L}_1 \uplus \mathbb{L}_2 \qquad \mathbb{L}_2 = \mathbb{L} \setminus \overline{\ell s} \qquad \vdash \mathbb{L}_2\ \mathsf{locates}\ \overline{\kappa}}{\Gamma^{\mathbb{T}} \vdash mod\ \mathtt{import}\ \overline{\ell s} \Rightarrow \mathbb{L}_1; \overline{\kappa}; -|\mathbb{S}|_{\overline{\ell s}}}\ (\mathsf{IMPORT}^{\mathbb{T}})$$

Fig. 26. Template Computation

Template Locators: $\vdash \mathbb{L}$ locates $\overline{\kappa}$

$$\frac{\vdash \mathcal{L} \text{ locates } \overline{\alpha}}{\vdash \mathcal{L}^{\mathbb{T}} \text{ locates } \overline{\kappa_\alpha}} \ (\mathsf{LOC}^{\mathbb{T}})$$

Fig. 27. Template Computation (continued)

$$\begin{array}{rcll} [\![\kappa_1]\!] & \approx & [\![\kappa_2]\!] & \\ [\![\mathbb{F}_1]\!]^{\pm} & \approx & [\![\mathbb{F}_2]\!]^{\pm} & \text{if } \mathbb{F}_1 \approx \mathbb{F}_2 \\ \{\!|\overline{\ell : \mathbb{S}_1}|\!\} & \approx & \{\!|\overline{\ell : \mathbb{S}_2}|\!\} & \text{if } \overline{\mathbb{S}_1 \approx \mathbb{S}_2} \\ (\mathbb{L}_1; \overline{\kappa}_1; \mathbb{S}_1) & \approx & (\mathbb{L}_2; \overline{\kappa}_2; \mathbb{S}_2) & \text{if } \mathbb{S}_1 \approx \mathbb{S}_2 \end{array}$$

Fig. 28. Template Approximation

(3) $|\Sigma|^{\mathbb{T}} = |\Sigma^{\mathbb{T}}|$.
(4) $(|\Sigma|_{\overline{\ell s}})^{\mathbb{T}} = |\Sigma^{\mathbb{T}}|_{\overline{\ell s}}$.
(5) $(\mathcal{L} \setminus \overline{\ell s})^{\mathbb{T}} = \mathcal{L}^{\mathbb{T}} \setminus \overline{\ell s}$.
(6) $(\mathcal{L}_1 \uplus \mathcal{L}_2)^{\mathbb{T}} = \mathcal{L}_1^{\mathbb{T}} \uplus \mathcal{L}_2^{\mathbb{T}}$.
(7) $\mathrm{dom}(\Sigma^{\mathbb{T}}) = \mathrm{dom}(\Sigma)$.
(8) $(\delta\Sigma)^{\mathbb{T}} = \Sigma^{\mathbb{T}}$.

To deal with the rule LINK in the completeness proof below, we need one further property. Namely, we need to know that the signatures Σ_2' and Σ_2 computed by the static and the regular pass over mod_2 in this rule have the same *domain*. This cannot be proved directly, but we can prove it as a corollary of the following more general lemma that states that the *templates* of the two signatures are sufficiently similar. More precisely, we define a simple notion of *template approximation* in Figure 28 (which we extend pointwise to template environements $\Gamma^{\mathbb{T}}$). This approximation ignores the kinds of type imports, which is convenient to make the induction go through easily, independent of local choices of locators and type variables. Obviously, the definition is reflexive, *i.e.,* $\mathbb{S}_1 = \mathbb{S}_2$ implies $\mathbb{S}_1 \approx \mathbb{S}_2$ (and thus likewise for template environments).

LEMMA 9.6 (DETERMINISTIC SHAPES). *Assume* $\Gamma^{\mathbb{T}} \approx \Gamma'^{\mathbb{T}}$.

(1) If $\Gamma; \mathcal{R}; \overline{\beta} \vdash mod : \Sigma$ *and* $\Gamma'; \mathcal{R}'; \overline{\beta}' \vdash mod : \Sigma'$*, then* $\Sigma^{\mathbb{T}} \approx \Sigma'^{\mathbb{T}}$.
(2) If $\Gamma \vdash mod : \Sigma$ *and* $\Gamma' \vdash mod : \Sigma'$*, then* $\Sigma^{\mathbb{T}} \approx \Sigma'^{\mathbb{T}}$.
(3) If $\Gamma \vdash mod : \Phi$ *and* $\Gamma' \vdash mod : \Phi'$*, then* $\Phi^{\mathbb{T}} \approx \Phi'^{\mathbb{T}}$.
(4) If $\Gamma \vdash usig \rightsquigarrow \Phi$ *and* $\Gamma' \vdash usig \rightsquigarrow \Phi'$*, then* $\Phi^{\mathbb{T}} \approx \Phi'^{\mathbb{T}}$.
(5) If $\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma$ *and* $\vdash \Sigma_1' + \Sigma_2' \Rightarrow \Sigma'$ *with* $\Sigma_1^{\mathbb{T}} \approx \Sigma_1'^{\mathbb{T}}$ *and* $\Sigma_2^{\mathbb{T}} \approx \Sigma_2'^{\mathbb{T}}$*, then* $\Sigma^{\mathbb{T}} \approx \Sigma'^{\mathbb{T}}$.

(All properties also hold if either or both judgments are static.)

COROLLARY 9.7 (DETERMINISTIC DOMAINS). *If* $\Gamma; \mathcal{R}; \overline{\beta} \vdash mod : \Sigma$ *and* $\Gamma'; \mathcal{R}'; \overline{\beta}' \vdash mod : \Sigma'$ *with* $\Gamma^{\mathbb{T}} = \Gamma'^{\mathbb{T}}$*, then* $\mathrm{dom}(\Sigma) = \mathrm{dom}(\Sigma')$. *(Likewise if either or both judgments are static.)*

Using these properties we can prove that template computation is complete with respect to the typing judgment:

THEOREM 9.8 (COMPLETENESS OF TEMPLATE COMPUTATION). *Suppose* $\vdash \Gamma \Uparrow$.

(*1*) *If* $\Gamma \vdash typ \rightsquigarrow \mathrm{A}$ *and* $\vdash \mathrm{A} \Uparrow \kappa$*, then* $\Gamma^{\mathbb{T}} \vdash typ \Rightarrow \kappa$.
(*2*) *If* $\Gamma; \mathcal{R}; \overline{\beta} \vdash mod : \Sigma$*, then* $\Gamma^{\mathbb{T}} \vdash mod \Rightarrow \mathcal{R}^{\mathbb{T}}; \overline{\kappa_\beta}; \Sigma^{\mathbb{T}}$.
(*3*) *If* $\Gamma \vdash mod : \Sigma$*, then* $\Gamma^{\mathbb{T}} \vdash mod \Rightarrow \{\!|\}; \overline{\kappa}; \Sigma^{\mathbb{T}}$ *for some* $\overline{\kappa}$.
(*4*) *If* $\Gamma \vdash mod : \Phi$*, then* $\Gamma^{\mathbb{T}} \vdash mod \Rightarrow \Phi^{\mathbb{T}}$.
(*5*) *If* $\Gamma \vdash usig \rightsquigarrow \Phi$*, then* $\Gamma^{\mathbb{T}} \vdash usig \Rightarrow \Phi^{\mathbb{T}}$.
(*6*) *If* $\vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma$*, then* $\Sigma_1^{\mathbb{T}} + \Sigma_2^{\mathbb{T}} = \Sigma^{\mathbb{T}}$.
(*7*) *If* $\vdash (\mathcal{L}_1; \Sigma_1) \rightleftarrows (\mathcal{L}_2; \Sigma_2) \rightsquigarrow \delta$ *and* $\mathcal{L}_1 \subseteq \Sigma_1$ *and* $\mathcal{L}_2 \subseteq \Sigma_2$*, then* $\mathcal{L}_1 \# \mathcal{L}_2$ *and* $\mathrm{dom}(\mathcal{L}_1) \subseteq \mathrm{dom}(\Sigma_2)$ *and* $\mathrm{dom}(\mathcal{L}_2) \subseteq \mathrm{dom}(\Sigma_1)$.

PROOF. By induction on derivations. The first part again depends on the details of the core language, and we assume that it can be proved for all cases not specified by our grammar. The most interesting module cases are the following:

— Rule LINK:
  — to show: $\Gamma^{\mathbb{T}} \vdash (\mathrm{X} = mod_1)$ `with` $mod_2 \Rightarrow (\mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{R}_2)^{\mathbb{T}}; \overline{\kappa_{\beta_1}}, \overline{\kappa_{\beta_2}}; \Sigma^{\mathbb{T}}$
  — premise: $\Gamma; \mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{L}_1; \overline{\beta_1} \vdash mod_1 : \Sigma_1$ and $\Gamma, \mathrm{X} : |\Sigma_1|; \mathcal{R} \uplus \mathcal{R}_2 \uplus \mathcal{L}_2; \overline{\beta_2} \vdash_{\mathrm{stat}} mod_2 : \Sigma_2'$ and $\Gamma, \mathrm{X} : |\delta\Sigma_1|; \mathcal{R} \uplus \mathcal{R}_2 \uplus \delta\mathcal{L}_2; \overline{\beta_2} \vdash mod_2 : \Sigma_2$ with $\mathcal{R}_1 \# \Sigma_2$ and $\mathcal{R}_2 \# \Sigma_1$, and $\vdash (\mathcal{L}_1; \Sigma_1) \rightleftarrows (\mathcal{L}_2; \Sigma_2') \rightsquigarrow \delta$ and $\vdash \delta\Sigma_1 + \Sigma_2 \Rightarrow \Sigma$
  — let $\mathbb{S}_1 = \Sigma_1^{\mathbb{T}}$ and $\mathbb{S}_2 = \Sigma_2^{\mathbb{T}}$
  — let $\mathbb{L} = \mathcal{R}^{\mathbb{T}}$ and $\mathbb{L}_1 = (\mathcal{R}_1 \uplus \mathcal{L}_1)^{\mathbb{T}}$ and $\mathbb{L}_2 = (\mathcal{R}_2 \uplus \mathcal{L}_2)^{\mathbb{T}}$
  — by Lemma 9.5, $(\mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{L}_1)^{\mathbb{T}} = \mathbb{L} \uplus \mathbb{L}_1$ and $(\mathcal{R} \uplus \mathcal{R}_2 \uplus \mathcal{L}_2)^{\mathbb{T}} = \mathbb{L} \uplus \mathbb{L}_2$
  — by Rule LINK$^{\mathbb{T}}$, we need to show:
    (1) $\Gamma^{\mathbb{T}} \vdash mod_1 \Rightarrow \mathbb{L} \uplus \mathbb{L}_1; \overline{\kappa_{\beta_1}}; \mathbb{S}_1$
      — follows by induction (Part 2)
    (2) $\Gamma^{\mathbb{T}}, \mathrm{X} : |\mathbb{S}_1| \vdash mod_2 \Rightarrow \mathbb{L} \uplus \mathbb{L}_2; \overline{\kappa_{\beta_2}}; \mathbb{S}_2$
      — follows by induction (Part 2), using Lemma 9.5 to show that $|\mathbb{S}_1| = |\Sigma_1|^{\mathbb{T}} = |\delta\Sigma_1|^{\mathbb{T}}$ and $\mathbb{L} \uplus \mathbb{L}_2 = \mathcal{R}^{\mathbb{T}} \uplus \mathcal{R}_2^{\mathbb{T}} \uplus \mathcal{L}_2^{\mathbb{T}} = \mathcal{R}^{\mathbb{T}} \uplus \mathcal{R}_2^{\mathbb{T}} \uplus (\delta\mathcal{L}_2)^{\mathbb{T}}$
    (3) $\mathbb{L}_1 \# \mathbb{L}_2$, which using Lemma 9.5 reduces to showing:
      (a) $\mathcal{R}_1 \# \mathcal{R}_2$: by definition of $\uplus$
      (b) $\mathcal{L}_1 \# \mathcal{R}_2$: by Lemma 8.4, $\mathcal{L}_1 \subseteq \Sigma_1$, and by assumption, $\Sigma_1 \# \mathcal{R}_2$
      (c) $\mathcal{R}_1 \# \mathcal{L}_2$: $\mathrm{dom}(\mathcal{L}_2) = \mathrm{dom}(\delta\mathcal{L}_2)$, and by Lemma 8.4, $\delta\mathcal{L}_2 \subseteq \Sigma_2$, and by assumption, $\Sigma_2 \# \mathcal{R}_1$
      (d) $\mathcal{L}_1 \# \mathcal{L}_2$: follows from Part 7, using Lemma 8.4 to show that $\mathcal{L}_1 \subseteq \Sigma_1$ and $\mathcal{L}_2 \subseteq \Sigma_2'$
    (4) $\mathcal{R}_1^{\mathbb{T}} = \mathbb{L}_1 \backslash \mathrm{dom}(\mathbb{S}_2)$, *i.e.,* $\mathcal{R}_1^{\mathbb{T}} = \mathcal{R}_1^{\mathbb{T}} \backslash \mathrm{dom}(\mathbb{S}_2) \uplus \mathcal{L}_1^{\mathbb{T}} \backslash \mathrm{dom}(\mathbb{S}_2)$
      By Lemma 9.5 this reduces to showing:
      (a) $\mathcal{R}_1^{\mathbb{T}} \backslash \mathrm{dom}(\mathbb{S}_2) = \mathcal{R}_1^{\mathbb{T}}$:
        this is entailed by $\mathcal{R}_1 \# \Sigma_2$, which was an assumption
      (b) $\mathcal{L}_1^{\mathbb{T}} \backslash \mathrm{dom}(\mathbb{S}_2) = \emptyset$:
        this is entailed by showing $\mathrm{dom}(\mathcal{L}_1) \subseteq \mathrm{dom}(\Sigma_2)$:
        — by Lemma 8.4, $\mathcal{L}_1 \subseteq \Sigma_1$ and $\mathcal{L}_2 \subseteq \Sigma_2'$
        — by Part 7, $\mathrm{dom}(\mathcal{L}_1) \subseteq \mathrm{dom}(\Sigma_2')$
        — by Lemma 9.5, $(\Gamma, \mathrm{X} : |\Sigma_1|)^{\mathbb{T}} = (\delta\Gamma, \mathrm{X} : |\delta\Sigma_1|)^{\mathbb{T}}$
        — by Corollary 9.7, $\mathrm{dom}(\Sigma_2') = \mathrm{dom}(\Sigma_2)$
    (5) $\mathcal{R}_2^{\mathbb{T}} = \mathbb{L}_2 \backslash \mathrm{dom}(\mathbb{S}_1)$, *i.e.,* $\mathcal{R}_2^{\mathbb{T}} = \mathcal{R}_2^{\mathbb{T}} \backslash \mathrm{dom}(\mathbb{S}_1) \uplus \mathcal{L}_2^{\mathbb{T}} \backslash \mathrm{dom}(\mathbb{S}_1)$
      By Lemma 9.5, this reduces to showing:
      (a) $\mathcal{R}_2^{\mathbb{T}} \backslash \mathrm{dom}(\mathbb{S}_1) = \mathcal{R}_2^{\mathbb{T}}$:
        this is entailed by $\mathcal{R}_2 \# \Sigma_1$, which was an assumption
      (b) $\mathcal{L}_2^{\mathbb{T}} \backslash \mathrm{dom}(\mathbb{S}_1) = \emptyset$:
        this is entailed by showing $\mathrm{dom}(\mathcal{L}_2) \subseteq \mathrm{dom}(\Sigma_1)$:
        — by Lemma 8.4, $\mathcal{L}_1 \subseteq \Sigma_1$ and $\mathcal{L}_2 \subseteq \Sigma_2'$
        — by Part 7, $\mathrm{dom}(\mathcal{L}_2) \subseteq \mathrm{dom}(\Sigma_1)$

(6) $\Sigma^\mathbb{T} = \mathbb{S}_1 + \mathbb{S}_2$
— by Lemma 9.8, $(\delta\Sigma_1)^\mathbb{T} = \Sigma_1^\mathbb{T}$
— by induction (Part 6), $\mathbb{S}_1 + \mathbb{S}_2 = \Sigma^\mathbb{T}$

— Rule NEW:
— to show: $\Gamma^\mathbb{T} \vdash \mathtt{new}\ \mathit{mod} \Rightarrow (\delta\mathcal{L})^\mathbb{T}; \overline{\kappa_{\delta\beta}}; (\delta\Sigma)^\mathbb{T}$
— premise: $\Gamma \vdash \mathit{mod} : [\![\forall\overline{\alpha}.\, \exists\overline{\beta}.\, (\mathcal{L}; \Sigma)]\!]^+$
— by induction (Part 3), $\Gamma^\mathbb{T} \vdash \mathit{mod} \Rightarrow \{\![\}\!\}; \overline{\kappa}; [\![\mathcal{L}^\mathbb{T}; \overline{\kappa_\beta}; \Sigma^\mathbb{T}]\!]^+$
— by rule NEW$^\mathbb{T}$, $\Gamma^\mathbb{T} \vdash \mathtt{new}\ \mathit{mod} \Rightarrow \mathcal{L}^\mathbb{T}; \overline{\kappa_\beta}; \Sigma^\mathbb{T}$
— by Lemma 9.5, $(\delta\mathcal{L})^\mathbb{T} = \mathcal{L}^\mathbb{T}$ and $(\delta\Sigma)^\mathbb{T} = \Sigma^\mathbb{T}$
— by the assumption that δ is kind-preserving, $\overline{\kappa_{\delta\beta}} = \overline{\kappa_\beta}$

— Rule LOOKUP:
— to show:
(1) $\mathcal{L}_1 \# \mathcal{L}_2$:
— proof by contradiction: assume $\ell s \in \mathrm{dom}(\mathcal{L}_1) \cap \mathrm{dom}(\mathcal{L}_2)$
— then $\mathcal{L}_1(\ell s) = \alpha_1$ and $\mathcal{L}_2(\ell s) = \alpha_2$ for some α_1, α_2
— consequently, $(\Sigma_2 \circ \mathcal{L}_1^{-1})(\alpha_1) = \mathrm{A}_2$ and $(\Sigma_1 \circ \mathcal{L}_2^{-1})(\alpha_2) = \mathrm{A}_1$ for some $\mathrm{A}_1, \mathrm{A}_2$
— that is, $\Sigma_2(\ell s) = \mathrm{A}_2$ and $\Sigma_1(\ell s) = \mathrm{A}_1$
— by the definition of $\uplus$, $\alpha_1 \neq \alpha_2$
— consequently, $(\Sigma_2 \circ \mathcal{L}_1^{-1}) \uplus (\Sigma_1 \circ \mathcal{L}_2^{-1}) \supseteq \{\alpha_1 \mapsto \mathrm{A}_2, \alpha_2 \mapsto \mathrm{A}_1\}$
— by assumption, $\mathcal{L}_1 \subseteq \Sigma_1$ and $\mathcal{L}_2 \subseteq \Sigma_2$
— hence, $\mathrm{A}_1 = \alpha_1$ and $\mathrm{A}_2 = \alpha_2$
— then $\{\alpha_1 \mapsto \mathrm{A}_2, \alpha_2 \mapsto \mathrm{A}_1\}$ is already cyclic, and suitable δ_i cannot possibly exist
(2) $\mathrm{dom}(\mathcal{L}_1) \subseteq \mathrm{dom}(\Sigma_2)$:
— follows directly from the definitions of $\mathcal{L}_1^{-1}$ and $\circ$
(3) $\mathrm{dom}(\mathcal{L}_2) \subseteq \mathrm{dom}(\Sigma_1)$:
— likewise □

Putting It All Together. As an example of the use of the template computation algorithm, consider computing the unit signature of a module expression mod under context Γ according to rule UNIT. First compute $\Gamma^\mathbb{T} \vdash \mathit{mod} \Rightarrow \mathbb{L}; \overline{\kappa}; \mathbb{S}$ to obtain the templates $\mathbb{L}$ and $\overline{\kappa}$ ($\mathbb{S}$ is not needed here). Then pick a fresh type variable, with the respective kind, for each component in $\mathbb{L}$ and each of $\overline{\kappa}$, from which you obtain a suitable locator $\mathcal{L}$ and export type names $\overline{\beta}$ to check the premise $\Gamma; \mathcal{L}; \overline{\beta} \vdash \mathit{mod} : \Sigma$. Essentially, this all amounts to adding the side condition $\Gamma^\mathbb{T} \vdash \mathit{mod} \Rightarrow \mathcal{L}^\mathbb{T}; \overline{\kappa_\beta}; \mathbb{S}$ to the original rule, which determines a sufficiently unique choice for $\mathcal{L}$ and $\overline{\beta}$ locally. Doing so results in the deterministic typing rule UNIT-DET shown in Figure 29.

For a similar effect, rule COMPL can be decorated with an additional premise for computing a template, yielding the rule COMPL-DET in the same figure.

Rule SEAL needs two premises in its deterministic incarnation SEAL-DET, for both constituent submodules. Note that the template computation spits out $\overline{\kappa_{\beta_1}}$, and thereby also determines upfront how the split $\overline{\beta}_1, \overline{\alpha}_1$ of the linear type variables from the input context has to be performed.

A more complicated case is the remaining rule LINK, where the locators $\mathcal{L}_1$ and $\mathcal{L}_2$ are only part of the realizers used in the premises. In this case, the template locators $\mathbb{L}_1$ and $\mathbb{L}_2$ computed for mod_1 and mod_2 by the algorithm correspond to the *whole* realizers $\mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{L}_1$ and $\mathcal{R} \uplus \mathcal{R}_2 \uplus \mathcal{L}_2$, respectively. We can employ a similar technique as in the template computation rule LINK$^\mathbb{T}$ to both determine how to split the rule's input realizer into $\mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{R}_2$, and to obtain the actual $\mathcal{L}_1$ and $\mathcal{L}_2$. More precisely, we have

Modules: $\Gamma; \mathcal{R}; \overline{\beta} \vdash mod : \Sigma$

$$\frac{\begin{array}{rl} & \Gamma^{\mathbb{T}} \vdash mod_1 \Rightarrow \mathcal{R}^{\mathbb{T}} \uplus \mathcal{R}_1^{\mathbb{T}} \uplus \mathcal{L}_1^{\mathbb{T}}; \overline{\kappa_{\beta_1}}; \Sigma_1^{\mathbb{T}} \\ (\mathcal{R}_1 \uplus \mathcal{L}_1) \# (\mathcal{R}_2 \uplus \mathcal{L}_2) & \Gamma^{\mathbb{T}}, X : |\mathbb{S}_1| \vdash mod_2 \Rightarrow \mathcal{R}^{\mathbb{T}} \uplus \mathcal{R}_2^{\mathbb{T}} \uplus \mathcal{L}_2^{\mathbb{T}}; \overline{\kappa_{\beta_2}}; \Sigma_2^{\mathbb{T}} \\ \vdash \mathcal{L}_1 \text{ locates } \overline{\alpha_1} \quad \mathcal{R}_1 \# \Sigma_2 & \Gamma; \mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{L}_1; \overline{\beta_1} \vdash mod_1 : \Sigma_1 \\ \vdash \mathcal{L}_2 \text{ locates } \overline{\alpha_2} \quad \mathcal{R}_2 \# \Sigma_1 & \Gamma, X : |\Sigma_1|; \mathcal{R} \uplus \mathcal{R}_2 \uplus \mathcal{L}_2; \overline{\beta_2} \vdash_{\text{stat}} mod_2 : \Sigma_2' \\ \vdash (\mathcal{L}_1; \Sigma_1) \rightleftarrows (\mathcal{L}_2; \Sigma_2') \rightsquigarrow \delta & \Gamma, X : |\delta\Sigma_1|; \mathcal{R} \uplus \mathcal{R}_2 \uplus \delta\mathcal{L}_2; \overline{\beta_2} \vdash mod_2 : \Sigma_2 \\ \overline{\alpha_1}, \overline{\alpha_2} \text{ fresh} & \vdash \delta\Sigma_1 + \Sigma_2 \Rightarrow \Sigma \end{array}}{\Gamma; \mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{R}_2; \overline{\beta_1}, \overline{\beta_2} \vdash (X = mod_1) \texttt{ with } mod_2 : \Sigma} \text{ (LINK-DET)}$$

$$\frac{\begin{array}{rll} \Gamma^{\mathbb{T}} \vdash mod_1 \Rightarrow \mathcal{L}_1^{\mathbb{T}}; \overline{\kappa_{\beta_1}}; \mathbb{S}_1 & \vdash \mathcal{L}_1 \text{ locates } \overline{\alpha_1} & \Gamma; \mathcal{L}_1; \overline{\beta_1} \vdash mod_1 : \Sigma_1 \\ \Gamma^{\mathbb{T}}, X : |\mathbb{S}_1| \vdash mod_2 \Rightarrow \mathcal{L}_2^{\mathbb{T}}; \overline{\kappa_{\beta_2}}; \mathbb{S}_2 & \vdash \mathcal{L}_2 \text{ locates } \overline{\alpha_2} & \Gamma, X : |\Sigma_1|; \mathcal{L}_2; \overline{\beta_2} \vdash_{\text{stat}} mod_2 : \Sigma_2' \\ \vdash (\mathcal{L}_1; \Sigma_1) \rightleftarrows (\mathcal{L}_2; \Sigma_2') \rightsquigarrow \delta & & \delta\Gamma, X : |\delta\Sigma_1|; \delta\mathcal{L}_2; \overline{\beta_2} \vdash mod_2 : \Sigma_2 \\ \overline{\beta_2}, \overline{\alpha_2} \text{ fresh} & & \vdash \delta\Sigma_1 + \Sigma_2 \Rightarrow |\Sigma| \end{array}}{\Gamma; \{\!\}; \overline{\beta_1}, \overline{\alpha_1} \vdash (X = mod_1) \texttt{ seals } mod_2 : |\Sigma_1|} \text{ (SEAL-DET)}$$

Complete Modules: $\Gamma \vdash mod : \Sigma$

$$\frac{\Gamma^{\mathbb{T}} \vdash mod \Rightarrow \{\!\}; \overline{\kappa_\beta}; \mathbb{S} \qquad \Gamma; \{\!\}; \overline{\beta} \vdash mod : |\Sigma| \qquad \overline{\beta} \text{ fresh} \qquad \overline{\beta} \notin \text{fv}(\Sigma)}{\Gamma \vdash mod : |\Sigma|} \text{ (COMPL-DET)}$$

Units: $\Gamma \vdash mod : \Phi$

$$\frac{\Gamma^{\mathbb{T}} \vdash mod \Rightarrow \mathcal{L}^{\mathbb{T}}; \overline{\kappa_\beta}; \mathbb{S} \qquad \Gamma; \mathcal{L}; \overline{\beta} \vdash mod : \Sigma \qquad \vdash \mathcal{L} \text{ locates } \overline{\alpha} \qquad \overline{\alpha}, \overline{\beta} \text{ fresh}}{\Gamma \vdash mod : \forall \overline{\alpha}. \exists \overline{\beta}. (\mathcal{L}; \Sigma)} \text{ (UNIT-DET)}$$

Fig. 29. Deterministic Typing Rules for MixML

to pick $\mathcal{R}$, $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{L}_1$, and $\mathcal{L}_2$ such that:

$$\begin{array}{lll} \text{dom}(\mathcal{R}) & = \text{dom}(\mathbb{L}_1) \cap \text{dom}(\mathbb{L}_2) & \text{(common imports of both } mod_1 \text{ and } mod_2\text{)} \\ \text{dom}(\mathcal{R}_1) & = \text{dom}(\mathbb{L}_1) \setminus \text{dom}(\mathbb{S}_2) & (mod_1 \text{ imports not present in } mod_2\text{)} \\ \text{dom}(\mathcal{R}_2) & = \text{dom}(\mathbb{L}_2) \setminus \text{dom}(\mathbb{S}_1) & (mod_2 \text{ imports not present in } mod_1\text{)} \\ \text{dom}(\mathcal{L}_1) & = \text{dom}(\mathbb{L}_1) \cap \text{dom}(\mathbb{S}_2) \setminus \text{dom}(\mathbb{L}_2) & (mod_1 \text{ imports supplied by } mod_2\text{)} \\ \text{dom}(\mathcal{L}_2) & = \text{dom}(\mathbb{L}_2) \cap \text{dom}(\mathbb{S}_1) \setminus \text{dom}(\mathbb{L}_1) & (mod_2 \text{ imports supplied by } mod_1\text{)} \end{array}$$

These choices can be enforced locally by adding the premises $\Gamma^{\mathbb{T}} \vdash mod_1 \Rightarrow \mathcal{R}^{\mathbb{T}} \uplus \mathcal{R}_1^{\mathbb{T}} \uplus \mathcal{L}_1^{\mathbb{T}}; \overline{\kappa_{\beta_1}}; \Sigma_1^{\mathbb{T}}$ and $\Gamma^{\mathbb{T}}, X : |\Sigma_1^{\mathbb{T}}| \vdash mod_2 \Rightarrow \mathcal{R}^{\mathbb{T}} \uplus \mathcal{R}_2^{\mathbb{T}} \uplus \mathcal{L}_2^{\mathbb{T}}; \overline{\kappa_{\beta_2}}; \Sigma_2^{\mathbb{T}}$ to the rule, along with the side condition $(\mathcal{R}_1 \uplus \mathcal{L}_1) \# (\mathcal{R}_2 \uplus \mathcal{L}_2)$, resulting in rule LINK-DET from Figure 29. As for rule SEAL-DET, these premises also uniquely determine the split of the linear variables from the context into $\overline{\beta}_1$ and $\overline{\beta}_2$.

9.3. Decidability and Uniqueness

That is almost all. Let $\vdash^{\text{alg}}$ (and $\vdash^{\text{alg}}_{\text{stat}}$) stand for variants of our typing judgments where the rules LINK, SEAL, COMPL, and UNIT, and the static rule EVAL, have been replaced by their deterministic counterparts from Figures 23 and 29. When amended with template computation, the MixML typing rules describe a 3-pass type-checking algorithm

(template computation, static pass, and main pass) that is still sound and complete with respect to the declarative rules.

THEOREM 9.9 (COMPLETENESS OF ALGORITHMIC TYPE-CHECKING).
Suppose $\Gamma; \mathcal{R}; \overline{\beta}$ *is a well-formed context.*

(*1*) *If* $\Gamma \vdash \mathit{exp} : \mathrm{A}$, *then* $\Gamma \stackrel{\mathrm{alg}}{\vdash} \mathit{exp} : \mathrm{A}$.
(*2*) *If* $\Gamma \vdash \mathit{typ} \rightsquigarrow \mathrm{A}$, *then* $\Gamma \stackrel{\mathrm{alg}}{\vdash} \mathit{typ} \rightsquigarrow \mathrm{A}$.
(*3*) *If* $\Gamma; \mathcal{R}; \overline{\beta} \vdash \mathit{mod} : \Sigma$, *then* $\Gamma; \mathcal{R}; \overline{\beta} \stackrel{\mathrm{alg}}{\vdash} \mathit{mod} : \Sigma$.
(*4*) *If* $\Gamma \vdash \mathit{mod} : \Sigma$, *then* $\Gamma \stackrel{\mathrm{alg}}{\vdash} \mathit{mod} : \Sigma$.
(*5*) *If* $\Gamma \vdash \mathit{mod} : \Phi$, *then* $\Gamma \stackrel{\mathrm{alg}}{\vdash} \mathit{mod} : \Phi$.
(*6*) *If* $\Gamma \vdash \mathit{usig} \rightsquigarrow \Phi$, *then* $\Gamma \stackrel{\mathrm{alg}}{\vdash} \mathit{usig} \rightsquigarrow \Phi$.

(And similarly, for the static judgments.)

PROOF. By induction on the derivation, using Theorems 9.8 and 9.3. Once more, Parts 1 and 2 depend on the details of the core language, and we assume they hold for all additional cases. □

Again, the inverse, soundness, is straightforward, because the algorithmic rules are merely restrictions of the declarative ones:

THEOREM 9.10 (SOUNDNESS OF ALGORITHMIC TYPE-CHECKING).
Suppose $\Gamma; \mathcal{R}; \overline{\beta}$ *is a well-formed context.*

(*1*) *If* $\Gamma \stackrel{\mathrm{alg}}{\vdash} \mathit{exp} : \mathrm{A}$, *then* $\Gamma \vdash \mathit{exp} : \mathrm{A}$.
(*2*) *If* $\Gamma \stackrel{\mathrm{alg}}{\vdash} \mathit{typ} \rightsquigarrow \mathrm{A}$, *then* $\Gamma \vdash \mathit{typ} \rightsquigarrow \mathrm{A}$.
(*3*) *If* $\Gamma; \mathcal{R}; \overline{\beta} \stackrel{\mathrm{alg}}{\vdash} \mathit{mod} : \Sigma$, *then* $\Gamma; \mathcal{R}; \overline{\beta} \vdash \mathit{mod} : \Sigma$.
(*4*) *If* $\Gamma \stackrel{\mathrm{alg}}{\vdash} \mathit{mod} : \Sigma$, *then* $\Gamma \vdash \mathit{mod} : \Sigma$.
(*5*) *If* $\Gamma \stackrel{\mathrm{alg}}{\vdash} \mathit{mod} : \Phi$, *then* $\Gamma \vdash \mathit{mod} : \Phi$.
(*6*) *If* $\Gamma \stackrel{\mathrm{alg}}{\vdash} \mathit{usig} \rightsquigarrow \Phi$, *then* $\Gamma \vdash \mathit{usig} \rightsquigarrow \Phi$.

(And similarly, for the static judgments.)

Because we have a sound and complete algorithm, we have proved decidability of the MixML type system:

COROLLARY 9.11 (DECIDABILITY OF TYPE-CHECKING).
All MixML typing judgments are decidable.

Moreover, given this algorithm for computing the types of MixML modules that is both complete and deterministic, we obtain our final theorem as a direct consequence:

THEOREM 9.12 (UNIQUENESS OF TYPES).

(*1*) *If* $\Gamma \vdash \mathit{exp} \rightsquigarrow \mathrm{A}_1$ *and* $\Gamma \vdash \mathit{exp} \rightsquigarrow \mathrm{A}_2$, *then* $\mathrm{A}_1 = \mathrm{A}_2$.
(*2*) *If* $\Gamma \vdash \mathit{typ} \rightsquigarrow \mathrm{A}_1$ *and* $\Gamma \vdash \mathit{typ} \rightsquigarrow \mathrm{A}_2$, *then* $\mathrm{A}_1 = \mathrm{A}_2$.
(*3*) *If* $\Gamma; \mathcal{R}; \overline{\beta} \vdash \mathit{mod} : \Sigma_1$ *and* $\Gamma; \mathcal{R}; \overline{\beta} \vdash \mathit{mod} : \Sigma_2$, *then* $\Sigma_1 = \Sigma_2$.
(*4*) *If* $\Gamma \vdash \mathit{mod} : \Sigma_1$ *and* $\Gamma \vdash \mathit{mod} : \Sigma_1$, *then* $\Sigma_1 = \Sigma_2$.
(*5*) *If* $\Gamma \vdash \mathit{mod} : \Phi_1$ *and* $\Gamma \vdash \mathit{mod} : \Phi_2$, *then* $\Phi_1 = \Phi_2$.
(*6*) *If* $\Gamma \vdash \mathit{usig} \rightsquigarrow \Phi_1$ *and* $\Gamma \vdash \mathit{usig} \rightsquigarrow \Phi_2$, *then* $\Phi_1 = \Phi_2$.

PROOF. Assume two different types exist for a module under a given context. Because the type-checking algorithm defined by $\stackrel{\mathrm{alg}}{\vdash}$ is complete, it must be able to

compute both types. That is a contradiction because the algorithm is deterministic. □

10. RELATED AND FUTURE WORK

10.1. Module Systems

There is a large body of work on ML modules and mixin modules independently, some of which we cited in the introduction. We primarily 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 [1996] were the first to integrate a notion of mixin composition into ML modules with type components. They divide mixin modules into three sections. Of these, only the middle section is "mixable," and it may only contain datatype and function bindings. In addition, their focus lies mainly on merging datatype variants and function clauses in order to support extensible datatypes. They consider neither opaque sealing nor hierarchical structures, and expressly disallow separate compilation.

Recursive Modules for ML. As mentioned in the introduction, there are several proposals for extending ML with recursive modules [Crary et al. 1999; Russo 2001; Leroy 2003; Nakata and Garrigue 2006; Dreyer 2007b], but they do not handle separate compilation in the general case. Both Moscow ML [Russo 2001] and OCaml [Leroy 2003] support separate compilation for limited classes of recursive modules through the functor mechanism. For example, if recursive modules in these languages do not contain any internal uses of opaque sealing and only contain term components of *pointed* type (*i.e.,* functions or lazy suspensions), they can usually be separately compiled. This covers quite a few common cases, but is not a general solution. In particular, it cannot handle our separate compilation example from Section 2 (Figures 2 and 3).

The reason for the restrictions on sealing boils down to the double vision problem and the limitations of functor typing. First, none of these languages (with the exception of RMC [Dreyer 2007b]) properly handles double vision in general (see Dreyer [2007b] for details). Second, even in RMC, if we try to break up a recursive module $\texttt{rec}\,(\mathrm{X} : sig)\; mod$ into separately compiled functors of the form $\lambda(\mathrm{X} : sig).\; mod'$ (where mod' is a substructure of mod), then the connection between the abstract types defined by mod' and their forward declaration in sig is lost once more. Consequently, double vision again rears its ugly head when type-checking mod'.

Ignoring separate compilation, MixML's type system is closely based on RMC's, and our encoding of the `rec` construct for recursive modules yields essentially the same semantics as in RMC.[10] MixML's transparent linking generalizes RMC's `rec` construct, while opaque linking generalizes RMC's sealing operator. This generalization actually simplifies the semantics of the language: the typing rules for transparent and opaque linking are very similar—they match up premise for premise—whereas, in RMC, the rules for recursive and sealed modules differ significantly.

Nakata and Garrigue [2006] define a language of recursive modules with a direct type system in the style of Leroy [1994], which avoids elaboration of syntactic types and signatures into semantic ones. Like us, they support definitions of opaquely recursive types, but not transparently recursive ones. Their encoding of opaquely recursive

[10]We say "essentially" because MixML is more liberal than ML (and RMC) in certain respects. For example, when matching a structure against a signature with a transparent type spec `type t = int`, ML will require the structure to have a type component `t` equal to `int`; MixML will require only that, *if* the structure does have a `t` component, *then* it is equal to `int`. While we view this departure from ML as a potentially useful feature, it makes formal comparisons of expressiveness difficult.

types is simpler than ours in that they do not need to insert `new[·]` as described in Section 3. This is only because their type system does not attempt to handle the double vision problem in the first place, so there is no need to manually override it.

In a recent paper, Im et al. [2011] devise a refinement of Nakata and Garrigue's system that addresses double vision using a type equivalence relation that allows cyclic types and hence is not known to be decidable, similar to that of MixML's internal language LTG. They also give an algorithmically decidable variant of the relation for the purpose of external type-checking, which, like MixML's external type system, rejects transparent type cycles. Im et al. define the equivalence relation by resorting to a bisimulation, which amounts to constructing a greatest fixed-point. In contrast, our system sticks to the standard definition of type equivalence by the usual inference rules that yield a least fixed-point. While less general, this is sufficient for handling recursive modules that disallow transparent type cycles. Our type system also encompasses type constructors and respective $\beta\eta$-equivalences, which Im et al. do not consider.

Units. Units were originally proposed by Flatt and Felleisen [1998] as a recursive module extension to Scheme, which they extended with support for abstract type components. Later work by Owens and Flatt [2006] extended units with hierarchical namespaces (called *modules*) and translucent type components. 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 MixML's higher-order units, but also necessarily introducing subtyping into the core language (unlike our package extension, which confines unit signatures to package types, without infecting the rest of the language).

In OF, as in other mixin-based languages, units may be recursively linked with each other, but they are not hierarchically composable into other units. In contrast, MixML modules are both hierarchically composable and recursively linkable. MixML *units* (named in homage to Flatt's units), which are just suspended modules, are composable both hierarchically and recursively as well. For example, to recursively link units U_1 and U_2 we write `[new` U_1 `with new` U_2`]`, as seen at the end of Section 2.

OF requires 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 emulate ML-like modules, but in this approach modules require signature annotations on essentially every subterm (for example, each functor application involves three distinct signature annotations). Moreover, it is not clear how a general recursive module construct $\mathtt{rec}\,(\mathrm{X} : \mathit{sig})\ \mathit{mod}$ would be expressed in OF. In contrast, our MixML encoding of ML-style modules is simple and direct and includes recursive modules.

Recursive DLLs. Duggan [2002] presents a language modelling recursive *dynamically linked libraries* (DLLs). His units (called modules in his paper) are similar to OF, but enriched with explicit support for sealing and an orthogonal construct for dynamic typing. As in other mixin approaches, his modules are not hierarchically composable. In addition, his system does not support transparent type definitions, only opaque 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.

Signature Operators. Ramsey et al. [2005] propose a variety of extensions to the ML signature language. Some of them are expressible in MixML: signature composition (`andalso`) directly corresponds to linking, and the `adding` and `revealing` constructs for signature extension and refinement can also be encoded using linking. Moreover, they propose a binder (`as`) that plays a role similar to the variable binding in MixML's linking construct. Other extensions presented in their paper, such as renaming and removal of components, cannot be encoded directly in MixML. However, similar operators are present in classical CMS-style mixin modules [Ancona and Zucca 2002], and we believe these could be readily incorporated into MixML.

Scala. Scala [Odersky et al. 2003; Odersky and Zenger 2005; Cremet et al. 2006] is a language combining object-oriented mixin class composition with ML-style type components. Being OO, objects and classes take the place of modules and units, respectively. Moreover, objects are fully first-class citizens.

Scala's mixin composition is, in some regards, very similar to our transparent linking. However, there are also fundamental differences. Scala's mixin composition operates on classes, not on objects, and it is not hierarchical (although the underlying νObj calculus does define linking on objects [Odersky et al. 2003]). It allows overriding of values, but on the other hand, restricts specialization of abstract fields to be left-to-right.

Besides classes, which are nominal, Scala also provides structural object types. Being purely types, they are more restricted—*e.g.,* they can only contain abstract definitions (*i.e.,* imports), and they cannot be instantiated to create objects. They also cannot be recursive, nor can they be composed with each other without first being named. *Traits* are yet another kind of unit-like mechanism: like classes, they are nominal and can be composed, but like structural types, they cannot be instantiated.

Abstraction is only allowed over types, and while abstract types can be concretized with classes, they cannot actually be *treated* as classes. That is, composition or object instantiation is not possible anywhere where the concrete definition is not known. Consequently, Scala cannot express the equivalent of unit imports and thus higher-order units. Even first-order functors cannot be expressed directly in Scala, because the language currently lacks the ability to handle *dependent functions* whose result type depends on their argument *value*.[11] Instead, they have to be emulated through generic functions or classes, which requires manually separating the static and dynamic components of the argument.

Data abstraction can be achieved in at least two ways in Scala: either in the typical OO-style, via the class mechanism and its access modifiers, or by ascribing a structural type to an existing object. The latter mechanism is very similar to ML-style sealing, which we model using opaque linking. However, Scala provides this mechanism without addressing the double vision problem. (There are alternative constructs, like `object` declarations, that avoid double vision in specific cases.)

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

J&. Also in the context of object-oriented programming, Nystrom et al. argue that *nested intersection* (hierarchical composability) for nested classes is an essential feature for supporting compositional modular extensions [Nystrom et al. 2006; Nystrom et al. 2004]. They devise J&, a mixin extension to a Java-like language that supports this feature, and give a number of examples demonstrating its utility.

[11] Preliminary support is available at least for methods (which differ from functions), but as of version 2.8 of the language, this support is still flagged as experimental.

The language does not support type components in the same way MixML or Scala do. Nested and abstract class definitions (a.k.a. "virtual types"), which are only briefly mentioned in the papers, can simulate type components to a certain extent, but because they can also be overridden covariantly, they do not actually express type equivalences but merely upper bounds. Thus, it is unclear to us how J& could encode modular abstractions involving type sharing specifications. Lacking any form of dependent types, even simpler functor-based abstractions may be difficult to express.

Data abstraction in J& is expressed using private members. There is no equivalent to ML-style sealing and the compositional data abstraction it provides.

Newspeak. Newspeak is a more radical object-oriented language designed by Bracha et al. [2010] that emphasizes nested classes as an important modularity mechanism. It can express mixin composition through inheritance by making the "super" reference late-bound. Because classes are first-class entities, this feature can also be applied to encode hierarchical composition for nested classes, but by default, class members are just plain overridden, like any other component.

As the authors note, Newspeak's design strongly values "flexibility" over "semantic consistency" [Bracha et al. 2010]. Consequently, it is neither typed nor does it currently provide any form of data abstraction.

10.2. Elaboration and Internal Languages

Translation of Modules. The idea of defining language semantics, and especially the semantics of module systems, in terms of a translation into an internal language (IL) is well-established. Harper and Stone [2000] give a definition of full Standard ML via translation into a dependently-typed IL named XML [Harper and Mitchell 1993], extended with translucent sums [Harper and Lillibridge 1994]. A similar approach was taken by Dreyer in his thesis on module systems [Dreyer 2005], where he targets an even more expressive IL that features singleton kinds [Stone and Harper 2006; Dreyer et al. 2003].

Together with Russo, and based on his previous work [Russo 1999b; 1998], we recently demonstrated that, in fact, plain vanilla System F is sufficient as an IL for conventional ML modules, and a relatively straightforward, compositional translation is possible, in which structures are interpreted merely as existential packages and functors as polymorphic functions [Rossberg et al. 2010].

Dreyer's earlier work on RMC [Dreyer 2007b] already employed the same basic idea as this "F-ing" approach, but with the necessary extensions to both the IL and the translation to encompass recursive type abstraction. Dreyer observed that existential types are no longer sufficient to represent type abstraction in the presence of recursive structures. He devised the RTG calculus [Dreyer 2007a] to extend System F with constructs for recursive type generation, and proposed destination-passing style (cf. Section 8) for the translation of RMC. Our elaboration follows RMC's very closely.

A variation of RTG was put forth by Montagu and Rémy [2009]. They recast RTG's type generation and definition constructs as *open existential types* that generalize the usual System F notion of existential types. Similarly to LTG, they recast RTG's effect system using linear typing contexts with a splitting (zipping) meta-operator. The technical details differ greatly, though. In particular, they are focused primarily on developing the core of a direct type system for modules which requires fewer typing annotations than RTG or LTG. They do not employ general substructural types, and they do not support general destination-passing style functions as RTG and LTG do.

Linear Types and Kinds. Ignoring some stylistic differences, the type and term level of LTG, and its way of tracking linearity, is largely a subset of Ahmed, Fluet, & Morrisett's more comprehensive language λ^{URAL} [Ahmed et al. 2005], which in turn is based

on Walker's work on substructural type systems [Walker 2005]. LTG is limited to only two modes and has no mode polymorphism. However, the semantics of LTG references is different from the $\lambda^{\mathsf{refURAL}}$-calculus, the extension of λ^{URAL} with references: in the latter, a linear reference is one that is not aliased, whereas in LTG, linearity provides the one-shot write capability to an (uninitialized) reference, that may still be aliased for reading. We are not aware of other work employing linear typing of references in this particular way.

Mazurak et al. [2010] present System F$^\circ$, a linear version of System F that sports a kind $\circ$ of linear types. But the meaning of linear kindedness in F$^\circ$ is completely different than in our system: in F$^\circ$ it is merely an elegant way to make mode annotations on most types implicit, by deriving them as kind information. The kind $\circ$ does not induce any linearity on the use of a type itself, as it does in LTG.

The use of a substructural *kind* system seems to be a novelty of our system. It consolidates the *ad hoc* linearity in Dreyer's RTG [Dreyer 2007a], leading to a more compositional calculus, and rendering the treatment of undefined type names very similar to that of uninitialized references. RTG has no references and tracks definedness of type names by a simple linear "effect system" for defining types. These effects are confined to the typing judgment and are not expressible in the types themselves. Consequently, it is not possible to abstract over linear objects in a first-class manner, and RTG has to provide specialised constructs, like a type of "destination-passing-style functions" to make up for that. See Section 7 for more discussion of the differences from RTG.

Linearity in our kind system is degenerate in the sense that there are no actual introduction or elimination forms for linear kinds on the type level itself. We considered including substructural arrow kinds, so that linear kinds could be consumed by type constructors. But as already mentioned in Section 7.2, we had no actual use for such functionality in the context of MixML. That said, it would be a natural extension to LTG that would require no significant changes to its setup of moded kinds.

10.3. Future Work

We believe that MixML already provides a fairly complete basis for a practical module system. To further expand its utility, though, we are interested in extending it with support for OCaml-style applicative functors [Leroy 1995] and type classes [Wadler and Blott 1989; Dreyer et al. 2007], as well as dynamic units [Rossberg 2006]. Integrating "applicative units" is particularly challenging, because it most likely will involve a form of existential quantifier hoisting, as for applicative functor signatures [Russo 1998], that does not readily fit into our elaboration framework. Existing encodings of type classes into ML modules should be comparatively easy to adapt to MixML, based on the encodings of traditional module features that we have given (especially functors). Likewise, we do not expect any significant hurdle for extending the language with forms of dynamic type analysis, as required for expressing dynamic modules. In fact, the type generation facility of LTG and RTG can already be viewed as a refinement of the respective construct needed for supporting safe type abstraction in the presence of dynamic casts [Rossberg 2003; Neis et al. 2011].

On the practical side, we would like to get MixML implemented in one of the existing ML systems. So far, we only have built a prototype interpreter for a language based on MixML [Rossberg and Dreyer 2008], which includes built-in support for most of the encodings given in Section 2. A mature implementation will likely raise additional questions, particularly with respect to optimizations, as well as the concrete realization of separate compilation. For example, a realistic implementation should avoid the repeated recursive structure copying that is performed for each and every variable use in our rules, and also by our simplistic translation of merging. It should not be difficult to get rid of the copying for simple cases, *i.e.,* where no unit abstraction/instantiation

is involved: all that seems to be needed is suitable compile-time partial evaluation of the initializer terms that our translation produces. In particular, that would take care of our (otherwise very expensive) encoding of n-ary structures through repeated use of linking. However, such partial evaluation is not possible when linking units across compilation boundaries, unless the implementation also has suitable support for cross-compilation inlining. It remains to be seen how challenging these problems turn out in practice.

Finally, a frequent question raised about languages defined by elaboration semantics is whether one could define them also via a "direct" semantics and then prove the two semantics equivalent. Our use of an elaboration semantics is by now a standard approach to defining the semantics of an ML-like language (cf. Harper and Stone [2000] for Standard ML, or the "F-ing modules" approach [Rossberg et al. 2010]). While elaboration semantics means that one can only understand the behavior of MixML programs in terms of the behavior of their evidence translations, our feeling is that it is nevertheless clearer and offers more insight into the type-theoretic underpinnings of the language than an *ad hoc* direct semantics.

The fact of the matter, though, is that we have no idea how to provide a direct semantics for the MixML language—some amount of elaboration seems essential for both the static and the dynamic semantics. Statically, elaborating source-language signatures into richer internal types with some form of existential quantification is necessary for addressing the so-called *avoidance problem* for local types [Dreyer et al. 2003; Harper and Pierce 2005], at least if one wishes to avoid undesirable restrictions on module projection and functor application (which in MixML is encoded via projection). Dynamically, we believe that any suitable operational semantics of recursive linking will require allocating the skeleton of the fully linked module before evaluating the defining expression to initialize it. However, in a language as expressive as MixML, the shape of that skeleton cannot be determined without the environment and type information inferred by the static semantics (or at least some approximation thereof, such as the "templates" in Section 9.2). In short, we conjecture that a direct semantics for MixML is not possible, and that elaboration semantics is the only viable way of defining it.

Acknowledgments. We thank Scott Kilpatrick and the anonymous reviewers for many thoughtful comments and questions. We especially thank the reviewer who helped to uncover a serious flaw in the proof of consistency (Section 7.5) given in an earlier draft of this article.

A. PROOF OF SOUNDNESS OF THE TRANSLATION

The proof for Theorem 8.7 proceeds by simultaneous induction on the derivation. Throughout the proof, we take the notation $\tau^{|\overline{\beta}|}$ to mean τ^{U} if $\overline{\beta}$ is empty and τ^{L} otherwise, with the following auxiliary lemma:

— If $\Xi, \overline{\beta^{\mathrm{L}}} \vdash e : \tau^{|\overline{\beta}|}$ and $\Xi \preceq \mathrm{U}$, then $\Xi \vdash \lambda\overline{\beta^{\mathrm{L}}}.e : (\forall^{\mathrm{L}}\overline{\beta}.\tau)^{\mathrm{U}}$.
— If $\Xi \vdash e : (\forall^{\mathrm{L}}\overline{\beta}.\tau)^{\iota}$ and $\overline{\beta^{\mathrm{U}}} \subseteq \Delta$ of Ξ, then $\Xi * \overline{\beta^{\mathrm{L}}} \vdash e\ \overline{\beta} : \tau^{|\overline{\beta}|}$.

(See Figure 12 for the definition of the syntax $\forall^{\mathrm{L}}\overline{\beta}.\tau$.)

Most cases for the main proof are fairly straightforward, the interesting ones are the following.

— Rule $\mathsf{LINK}^{\rightharpoonup}$:
  — let $\Xi = \Delta * \overline{\beta_1^{\mathrm{L}}} * \overline{\beta_2^{\mathrm{L}}}; \epsilon; \Gamma^{\circ}$
  — to show: $\Xi \vdash e : (\Sigma^{\circ} \to \{\})^{|\overline{\beta}_1, \overline{\beta}_2|}$
  — since $\Delta \vdash (\Gamma; \mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{R}_2; \overline{\beta}_1, \overline{\beta}_2) \Uparrow$, we know that $\overline{\beta_1^{\mathrm{U}}}, \overline{\beta_2^{\mathrm{U}}} \in \Delta$

— let $\Delta' = \Delta, \overline{\alpha_1^{\mathrm{U}}}, \overline{\alpha_2^{\mathrm{U}}}$
— first show that $\Delta \vdash \delta^\circ : \Delta'$ and $\Delta * \overline{\beta_1^{\mathrm{L}}} \vdash \delta^\circ : \Delta' * \overline{\beta_1^{\mathrm{L}}}$
— by Lemma 8.3 (9, 1, 8), $\Delta' \vdash \mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{L}_1 \Uparrow$,
and thus $\Delta' \vdash (\Gamma; \mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{L}_1; \overline{\beta}_1) \Uparrow$
— by Theorem 8.1 and Lemma 8.4, $\Delta' \vdash \Sigma_1 \Uparrow$, and thus, $\Delta' \vdash \Gamma, \mathrm{X}_1{:}\Sigma_1 \Uparrow$
— by Lemma 8.3 (9, 1, 8), $\Delta' \vdash \mathcal{R} \uplus \mathcal{R}_2 \uplus \mathcal{L}_2 \Uparrow$,
and thus $\Delta' \vdash (\Gamma; \mathcal{R} \uplus \mathcal{R}_2 \uplus \mathcal{L}_2; \overline{\beta}_2) \Uparrow$
— by Lemma 8.4, $\Delta' \vdash \Sigma_2' \Uparrow$
— obviously, $\Delta \vdash (\Gamma; \mathcal{R}; \overline{\beta'}) \Uparrow$ for any $\overline{\beta'} \subseteq \overline{\beta}_1, \overline{\beta}_2$
— by induction (10), $\Delta * \overline{\beta'^{\mathrm{L}}} \vdash \delta^\circ : \Delta' * \overline{\beta'^{\mathrm{L}}}$ for any $\overline{\beta'} \subseteq \overline{\beta}_1, \overline{\beta}_2$
— now show that $\Delta * \overline{\beta_1^{\mathrm{L}}}; \epsilon; \Gamma^\circ \vdash \delta^\circ e_1 : ((\delta\Sigma_1)^\circ \to \{\})^{|\overline{\beta}_1|}$
— as above, $\Delta' \vdash \mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{L}_1 \Uparrow$, hence $\Delta' \vdash (\Gamma; \mathcal{R} \uplus \mathcal{R}_1 \uplus \mathcal{L}_1; \overline{\beta}_1) \Uparrow$
— by induction (3), $\Delta' * \overline{\beta_1^{\mathrm{L}}}; \epsilon; \Gamma^\circ \vdash e_1 : (\Sigma_1^\circ \to \{\})^{|\overline{\beta}_1|}$
— by substitution and Lemma 8.5, $\Delta * \overline{\beta_1^{\mathrm{L}}}; \epsilon; (\delta\Gamma)^\circ \vdash \delta^\circ e_1 : ((\delta\Sigma_1)^\circ \to \{\})^{|\overline{\beta}_1|}$
— since $\overline{\alpha}_1, \overline{\alpha}_2$ fresh, $\delta\Gamma = \Gamma$
— then show that $\Delta * \overline{\beta_2^{\mathrm{L}}}; \epsilon; (\Gamma, \mathrm{X}_1{:}|\delta\Sigma_1|)^\circ \vdash e_2 : (\Sigma_2^\circ \to \{\})^{|\overline{\beta}_2|}$
— like above, $\Delta' \vdash \Sigma_1 \Uparrow$
— by Lemma 8.3 (10, 3), $\Delta \vdash |\delta\Sigma_1| \Uparrow$, and hence, $\Delta \vdash \Gamma, \mathrm{X}_1{:}|\delta\Sigma_1| \Uparrow$
— by Lemma 8.3 (9, 1, 8, 10), $\Delta \vdash \mathcal{R} \uplus \mathcal{R}_2 \uplus \delta\mathcal{L}_2 \Uparrow$,
and hence, $\Delta \vdash (\Gamma, \mathrm{X}_1{:}|\delta\Sigma_1|; \mathcal{R} \uplus \mathcal{R}_2 \uplus \delta\mathcal{L}_2; \overline{\beta}_2) \Uparrow$
— by induction (3), $\Delta * \overline{\beta_2^{\mathrm{L}}}; \epsilon; (\Gamma, \mathrm{X}_1{:}|\delta\Sigma_1|)^\circ \vdash e_2 : (\Sigma_2^\circ \to \{\})^{|\overline{\beta}_2|}$
— now for the coercions f_1 and f_2:
— as above, $\Delta' \vdash \Sigma_1 \Uparrow$
— by Lemma 8.3 (10), $\Delta \vdash \delta\Sigma_1 \Uparrow$
— as above, $\Delta \vdash (\Gamma, \mathrm{X}_1{:}|\delta\Sigma_1|; \mathcal{R} \uplus \mathcal{R}_2 \uplus \delta\mathcal{L}_2; \overline{\beta}_2) \Uparrow$
— by Theorem 8.1 and Lemma 8.4, $\Delta \vdash \Sigma_2 \Uparrow$
— by induction (7), $\Delta; \epsilon; \epsilon \vdash f_1 : \Sigma_1''^\circ \to (\delta\Sigma_1)^\circ$ and $\Delta; \epsilon; \epsilon \vdash f_2 : \Sigma_2''^\circ \to \Sigma_2^\circ$ with $\Sigma^\circ = \Sigma_1''^\circ * \Sigma_2''^\circ$
— let $\Xi_1 = \Delta; \epsilon; \Gamma^\circ, x : \Sigma^\circ$
and $\Xi_{11} = \Delta; \epsilon; \Gamma^\circ, x : \Sigma_1''^\circ$ and $\Xi_{12} = \Delta; \epsilon; \Gamma^\circ, x : \Sigma_2''^\circ$
— by Lemma 8.5, $\Xi_{11} * \Xi_{12} = \Xi_1$
— by weakening and LTG typing rules, $\Xi_{11} \vdash f_1\, x : (\delta\Sigma_1)^\circ$ and $\Xi_{12} \vdash f_2\, x : \Sigma_2^\circ$
— let $\Xi_2 = \Delta * \overline{\beta_1^{\mathrm{L}}}, \overline{\beta_2^{\mathrm{L}}}; \epsilon; \Gamma^\circ, x : \Sigma^{-\circ}$
and $\Xi_{21} = \Delta * \overline{\beta_1^{\mathrm{L}}}; \epsilon; \Gamma^\circ, x : \Sigma^{-\circ},\ \mathrm{X}_1 : |\delta\Sigma_1|^\circ,\ \mathrm{X}_2 : \Sigma_2^{-\circ}$
and $\Xi_{22} = \Delta * \overline{\beta_2^{\mathrm{L}}}; \epsilon; \Gamma^\circ, x : \Sigma^{-\circ},\ \mathrm{X}_1 : |\delta\Sigma_1|^{-\circ},\ \mathrm{X}_2 : \Sigma_2^\circ$
— by weakening and LTG typing rules, $\Xi_{21} \vdash \delta^\circ e_1\, \mathrm{X}_1 : \{\}$ and $\Xi_{22} \vdash e_2\, \mathrm{X}_2 : \{\}$
— by Lemma 8.5, $\Xi_{21} * \Xi_{22} = \Xi_2, \mathrm{X}_1 : |\delta\Sigma_1|^\circ, \mathrm{X}_2 : \Sigma_2^\circ$
— and thus, $\Xi_2, \mathrm{X}_1 : |\delta\Sigma_1|^\circ, \mathrm{X}_2 : \Sigma_2^\circ \vdash \delta^\circ e_1\, \mathrm{X}_1; e_2\, \mathrm{X}_2 : \{\}$
— by Lemma 8.5, $\Xi_1 * \Xi_2 = \Xi, x : \Sigma^\circ$
— by LTG typing rules, $\Xi \vdash e : (\Sigma^\circ \to \{\})^{|\overline{\beta}_1, \overline{\beta}_2|}$
— Rule $\widetilde{\mathsf{SEAL}}$:
— let $\Xi = \Delta * \overline{\beta_1^{\mathrm{L}}} * \overline{\alpha_1^{\mathrm{L}}}; \epsilon; \Gamma^\circ$
— to show: $\Xi \vdash e : (|\Sigma_1|^\circ \to \{\})^{|\overline{\beta}_1, \overline{\alpha}_1|}$
— since $\Delta \vdash (\Gamma; \{\!|\}; \overline{\beta}_1, \overline{\alpha}_1) \Uparrow$, we know that $\overline{\beta_1^{\mathrm{U}}}, \overline{\alpha_1^{\mathrm{U}}} \in \Delta$
— let $\Delta_1 = \Delta, \overline{\beta_2^{\mathrm{U}}}$ and $\Delta_2 = \Delta_1, \overline{\alpha_2^{\mathrm{U}}}$ and $\Delta_0 = \Delta_1 - \overline{\alpha}_1$
— let $\delta_1 = \{\overline{\alpha_1 \mapsto \delta\alpha_1}\}$ and $\Psi = \overline{\alpha_1{:=}\delta^\circ\alpha_1}$
— first show that $\Delta_1 * \overline{\beta_1^{\mathrm{L}}}; \Psi; \Gamma^\circ \vdash e_1 : (\Sigma_1^\circ \to \{\})^{|\overline{\beta}_1|}$

— by Lemma 8.3 (9, 1), $\Delta_1 \vdash \mathcal{L}_1 \Uparrow$, and hence, $\Delta_1 \vdash (\Gamma; \mathcal{L}_1; \overline{\beta}_1) \Uparrow$
— by induction (3) and weakening, $\Delta_1 * \overline{\beta_1^{\mathrm{L}}}; \Psi; \Gamma^\circ \vdash e_1 : (\Sigma_1^\circ \to \{\})^{|\overline{\beta}_1|}$
— then show that $\Delta_1 * \overline{\beta_2^{\mathrm{L}}}; \Psi; (\Gamma, \mathrm{X}_1{:}|\Sigma_1|)^\circ \vdash e_2 : (\Sigma_2^\circ \to \{\})^{|\overline{\beta}_2|}$
— as above, $\Delta_1 \vdash (\Gamma; \mathcal{L}_1; \overline{\beta}_1) \Uparrow$
— by Theorem 8.1 and Lemma 8.4, $\Delta_1 \vdash \Sigma_1 \Uparrow$
— by Lemma 8.3 (10, 3, 12), $\Delta_0 \vdash |\delta_1\Sigma_1| \Uparrow$ and $\Delta_0 \vdash \delta_1\Gamma \Uparrow$,
and hence, $\Delta_0 \vdash \delta_1\Gamma, \mathrm{X}_1{:}|\delta_1\Sigma_1| \Uparrow$
— because $\overline{\alpha}_2$ fresh and $\Delta_1 \vdash \Sigma_1 \Uparrow$, we have $\delta\Gamma = \delta_1\Gamma$ and $\delta\Sigma_1 = \delta_1\Sigma_1$
— so, $\Delta_0 \vdash \delta\Gamma, \mathrm{X}_1{:}|\delta\Sigma_1| \Uparrow$
— by Lemma 8.3 (9, 1, 10), $\Delta_2 \vdash \mathcal{L}_2 \Uparrow$ and then $\Delta_0 \vdash \delta\mathcal{L}_2 \Uparrow$
— thus, $\Delta_0 \vdash (\delta\Gamma, \mathrm{X}_1{:}|\delta\Sigma_1|; \delta\mathcal{L}_2; \overline{\beta}_2) \Uparrow$
— by induction (3), $\Delta_0 * \overline{\beta_2^{\mathrm{L}}}; \epsilon; (\delta\Gamma, \mathrm{X}_1{:}|\delta\Sigma_1|)^\circ \vdash e_2 : (\Sigma_2^\circ \to \{\})^{|\overline{\beta}_2|}$
— that is, $\Delta_0 * \overline{\beta_2^{\mathrm{L}}}; \epsilon; (\delta_1\Gamma, \mathrm{X}_1{:}|\delta_1\Sigma_1|)^\circ \vdash e_2 : (\Sigma_2^\circ \to \{\})^{|\overline{\beta}_2|}$
— by Theorem 8.1 and Lemma 8.4, $\Delta_0 \vdash \Sigma_2 \Uparrow$, and hence, $\Sigma_2 = \delta_1\Sigma_2$
— by Lemma 7.4, $\mathrm{fv}(e_2) \cap \overline{\alpha}_1 = \emptyset$, and hence, $e_2 = \delta_1^\circ e_2$
— thus, $\Delta_0 * \overline{\beta_2^{\mathrm{L}}}; \epsilon; (\delta_1\Gamma, \mathrm{X}_1{:}|\delta_1\Sigma_1|)^\circ \vdash \delta_1^\circ e_2 : ((\delta_1\Sigma_2)^\circ \to \{\})^{|\overline{\beta}_2|}$
— obviously, $\Delta_0 * \overline{\beta_2^{\mathrm{L}}}; \epsilon \vdash \delta_1^\circ : \Delta_1 * \overline{\beta_2^{\mathrm{L}}}; \Psi$
— by Lemma 7.15, $\Delta_1 * \overline{\beta_2^{\mathrm{L}}}; \Psi; (\Gamma, \mathrm{X}_1{:}|\Sigma_1|)^\circ \vdash e_2 : (\Sigma_2^\circ \to \{\})^{|\overline{\beta}_2|}$
— now for the coercions f_1 and f_2:
— like above, $\Delta_0 \vdash \delta\Sigma_1 \Uparrow$ and $\Delta_0 \vdash \Sigma_2 \Uparrow$
— by induction (7), $\Delta_0; \epsilon; \epsilon \vdash f_1 : \Sigma_1''^\circ \to (\delta\Sigma_1)^\circ$ and $\Delta_0; \epsilon; \epsilon \vdash f_2 : \Sigma_2''^\circ \to \Sigma_2^\circ$ with $|\Sigma|^\circ = \Sigma_1''^\circ * \Sigma_2''^\circ$
— as above, $\delta\Sigma_1 = \delta_1\Sigma_1$ and $\Sigma_2 = \delta_1\Sigma_2$
— by Lemma 8.4, $\Delta_0 \vdash |\Sigma| \Uparrow$, and so $|\Sigma| = |\delta_1\Sigma|$ and $\Sigma_1'' = \delta_1\Sigma_1''$ and $\Sigma_2'' = \delta_1\Sigma_2''$
— by Lemma 7.4, $\mathrm{fv}(f_1) \cap \overline{\alpha}_1 = \mathrm{fv}(f_2) \cap \overline{\alpha}_1 = \emptyset$, and hence, $f_1 = \delta_1^\circ f_1$ and $f_2 = \delta_1^\circ f_2$
— obviously, $\Delta_0; \epsilon \vdash \delta_1^\circ : \Delta_1; \Psi$
— by Lemma 7.15, $\Delta_1; \Psi; \epsilon \vdash f_1 : \Sigma_1''^\circ \to \Sigma_1^\circ$ and $\Delta_1; \Psi; \epsilon \vdash f_2 : \Sigma_2''^\circ \to \Sigma_2^\circ$
— let $\Xi_1 = \Delta_1; \Psi; \Gamma^\circ, x : |\Sigma_1|^{-\circ}, x' : |\Sigma|^\circ$
and $\Xi_{11} = \Delta_1; \Psi; \Gamma^\circ, x : |\Sigma_1|^{-\circ}, x' : \Sigma_1''^\circ$
and $\Xi_{12} = \Delta_1; \Psi; \Gamma^\circ, x : |\Sigma_1|^{-\circ}, x' : \Sigma_2''^\circ$
— by Lemma 8.5, $\Xi_{11} * \Xi_{12} = \Xi_1$
— by weakening and LTG typing rules, $\Xi_{11} \vdash f_1\, x' : \Sigma_1^\circ$ and $\Xi_{12} \vdash f_2\, x' : \Sigma_2^\circ$
— let $\Xi_2 = \Delta_1 * \overline{\beta_1^{\mathrm{L}}} * \overline{\beta_2^{\mathrm{L}}}; \Psi; \Gamma^\circ, x : |\Sigma_1|^\circ, x' : |\Sigma|^{-\circ}$
and $\Xi_3 = \Xi_2, \mathrm{X}_1 : \Sigma_1^\circ, \mathrm{X}_2 : \Sigma_2{}^\circ$
and $\Xi_{31} = \Delta_1; \Psi; \Gamma^\circ, x : |\Sigma_1|^\circ,\ x' : |\Sigma|^{-\circ},\ \mathrm{X}_1 : \Sigma_1^{-\circ},\ \mathrm{X}_2 : \Sigma_2^{-\circ}$
and $\Xi_{32} = \Delta_1 * \overline{\beta_1^{\mathrm{L}}}; \Psi; \Gamma^\circ, x : |\Sigma_1|^{-\circ},\ x' : |\Sigma|^{-\circ},\ \mathrm{X}_1 : \Sigma_1^\circ,\ \mathrm{X}_2 : \Sigma_2^{-\circ}$
and $\Xi_{33} = \Delta_1 * \overline{\beta_2^{\mathrm{L}}}; \Psi; \Gamma^\circ, x : |\Sigma_1|^{-\circ},\ x' : |\Sigma|^{-\circ},\ \mathrm{X}_1 : \Sigma_1^{-\circ},\ \mathrm{X}_2 : \Sigma_2^\circ$
— by Lemma 8.5, $\Xi_3 = \Xi_{31} * \Xi_{32} * \Xi_{33}$
— by LTG typing rules, $\Xi_{31} \vdash \mathrm{X}_1 : \Sigma_1^{-\circ}$ and $\Xi_{31} \vdash x : |\Sigma_1|^\circ$
— by Lemma 8.5 (7), $\Xi_{31} \vdash \mathrm{X}_1 : |\Sigma_1|^{-\circ}$
— like above, $\Delta_1 \vdash |\Sigma_1| \Uparrow$
— by Lemma 8.6 (2), $\Xi_{31} \vdash \mathrm{Copy}(\mathrm{X}_1, x : |\Sigma_1|) : \{\}$
— by weakening and LTG typing rules, $\Xi_{32} \vdash e_1\, \mathrm{X}_1 : \{\}$ and $\Xi_{33} \vdash e_2\, \mathrm{X}_2 : \{\}$
— and thus, $\Xi_3 \vdash \mathrm{Copy}(\mathrm{X}_1, x : |\Sigma_1|); e_1\, \mathrm{X}_1; e_2\, \mathrm{X}_2 : \{\}$
— by Lemma 8.5, $\Xi_1 * \Xi_2 = \Delta_0 * \overline{\beta_1^{\mathrm{L}}} * \overline{\beta_2^{\mathrm{L}}}; \Psi; \Gamma^\circ, x : |\Sigma_1|^\circ, x' : |\Sigma|^\circ$
— by LTG typing rules, $\Xi_1 * \Xi_2 \vdash \mathsf{let}\, \mathrm{X}_1 = \ldots, \mathrm{X}_2 = \ldots\, \mathsf{in}\, \ldots : \{\}$
— because $\Delta_0 \vdash |\Sigma| \Uparrow$, also $\Delta_1 \vdash |\Sigma| \Uparrow$
— by Lemma 8.6 (1) and weakening, $\Delta_1; \Psi; \Gamma^\circ, x : |\Sigma_1|^{-\circ} \vdash \mathrm{Create}(|\Sigma|) : |\Sigma|^\circ$

— by LTG typing rules, $\Delta_1 * \overline{\beta_1^{\mathrm{L}}} * \overline{\beta_2^{\mathrm{L}}}; \Psi; \Gamma^\circ, x : |\Sigma_1|^\circ \vdash \mathsf{let}\, x' = \ldots \,\mathsf{in}\, \ldots : \{\}$
— by LTG typing rule, $\Delta_1 * \overline{\beta_1^{\mathrm{L}}} * \overline{\beta_2^{\mathrm{L}}} * \alpha_1^{\mathrm{L}}; \epsilon; \Gamma^\circ, x : |\Sigma_1|^\circ \vdash \mathsf{def}\, \overline{\alpha_1 := \delta^\circ \alpha_1}\, \mathsf{in}\, \ldots : \{\}$
— by LTG typing rule, $\Delta * \overline{\beta_1^{\mathrm{L}}} * \alpha_1^{\mathrm{L}}; \epsilon; \Gamma^\circ, x : |\Sigma_1|^\circ \vdash \mathsf{new}\, \overline{\beta_2}\, \mathsf{in}\, \ldots : \{\}$
— by LTG typing rule, $\Delta * \overline{\beta_1^{\mathrm{L}}} * \alpha_1^{\mathrm{L}}; \epsilon; \Gamma^\circ \vdash e : (|\Sigma_1|^\circ \to \{\})^{|\overline{\beta}_1, \overline{\alpha}_1|}$

— Rule NEW$^{\rightsquigarrow}$:
— let $\Xi = \Delta; \epsilon; \Gamma^\circ$
— to show: $\Xi * \overline{\delta^\circ \beta^{\mathrm{L}}} \vdash \lambda x{:}(\delta\Sigma)^\circ.\, (!\, e)\ \overline{\delta^\circ \alpha}\ \overline{\delta^\circ \beta}\ x : ((\delta\Sigma)^\circ \to \{\})^{|\overline{\beta}|}$
— by induction (4) and weakening, $\Xi, x{:}(\delta\Sigma)^{-\circ} \vdash e : (?(\forall^{\mathrm{U}} \overline{\alpha}.\forall^{\mathrm{L}} \overline{\beta}.\Sigma^\circ \to \{\}))^{\mathrm{U}}$
— by Theorem 8.1 and Lemma 8.4, $\Delta \vdash [\![\forall \overline{\alpha}.\, \exists \overline{\beta}.\, (\mathcal{L}; \Sigma)]\!]^+ \Uparrow$
— hence, $\vdash \mathcal{L}$ locates $\overline{\alpha}$
— consequently, $\mathrm{fv}(\delta\overline{\alpha}) = \mathrm{fv}(\delta\mathcal{L})$
— because $\Delta \vdash \delta\mathcal{L} \Uparrow$ and $\Delta \vdash \delta\overline{\beta} \Uparrow$, we know $\mathrm{fv}(\delta\mathcal{L}, \delta\overline{\beta}) \subseteq \mathrm{dom}(\Delta)$
— by implicit assumptions, $\Delta \preceq \upsilon$ and δ kind-preserving
— hence, $\overline{\Delta \vdash \delta^\circ \alpha : \kappa_\alpha^{\mathrm{U}}}$ and $\Delta * \overline{\delta^\circ \beta^{\mathrm{L}}} \vdash \delta^\circ \beta : \kappa_\beta^{\mathrm{L}}$
— by Lemma 8.5,
$(\Xi * \overline{\delta^\circ \beta^{\mathrm{L}}}, x : (\delta\Sigma)^\circ) = (\Xi, x : (\delta\Sigma)^{-\circ}) * \Delta * (\Delta * \overline{\delta^\circ \beta^{\mathrm{L}}}) * (\Xi, x : (\delta\Sigma)^\circ)$
— hence, by LTG typing rules and the auxiliary lemma,
$\Xi * \overline{\delta^\circ \beta^{\mathrm{L}}}, x{:}(\delta\Sigma)^\circ \vdash (!\, e)\ \overline{\delta^\circ \alpha}\ \overline{\delta^\circ \beta}\ x : \{\}$

— Rule UNPACK$^{\rightsquigarrow}$:
— Analogous to the previous case.

— Rule COMPL$^{\rightsquigarrow}$:
— let $\Xi = \Delta; \epsilon; \Gamma^\circ$
— to show: $\Xi \vdash \mathsf{new}\, \overline{\beta}\, \mathsf{in}\, \mathsf{let}\, x = \mathrm{Create}(|\Sigma|)\, \mathsf{in}\, e\ x; x : |\Sigma|^{-\circ}$
— obviously, $\Delta, \overline{\beta^{\mathrm{U}}} \vdash (\Gamma; \{\!|\}; \overline{\beta}) \Uparrow$ for fresh $\overline{\beta}$
— by Lemma 8.6 (1) and weakening, $\Xi, \overline{\beta^{\mathrm{U}}} \vdash \mathrm{Create}(|\Sigma|) : |\Sigma|^\circ$
— by induction (3) and weakening, $\Xi, \overline{\beta^{\mathrm{L}}}, x{:}|\Sigma|^{-\circ} \vdash e : (|\Sigma|^\circ \to \{\})^{|\overline{\beta}|}$
— by Lemma 8.5,
$\Xi, \overline{\beta^{\mathrm{L}}}, x{:}|\Sigma|^\circ = (\Xi, \overline{\beta^{\mathrm{L}}}, x{:}|\Sigma|^{-\circ}) * (\Xi, \overline{\beta^{\mathrm{U}}}, x{:}|\Sigma|^\circ) * (\Xi, \overline{\beta^{\mathrm{U}}}, x{:}|\Sigma|^{-\circ})$
— hence, by LTG typing rules, $\Xi, \overline{\beta^{\mathrm{L}}}, x{:}|\Sigma|^\circ \vdash e\, x; x : |\Sigma|^{-\circ}$
— the goal follows by further straightforward application of LTG typing rules

— Rule UNIT$^{\rightsquigarrow}$:
— to show: $\Delta; \epsilon; \Gamma \vdash \lambda\overline{\alpha^{\mathrm{U}}}.\lambda\overline{\beta^{\mathrm{L}}}.e : (\forall^{\mathrm{U}} \overline{\alpha}.\forall^{\mathrm{L}} \overline{\beta}.(\Sigma^\circ \to \{\}))^{\mathrm{U}}$
— by Lemma 8.3 (9, 1), $\Delta, \overline{\alpha}, \overline{\beta} \vdash \mathcal{L} \Uparrow$, and so $\Delta, \overline{\alpha}, \overline{\beta} \vdash (\Gamma; \mathcal{L}; \overline{\beta}) \Uparrow$
— by induction (3), $\Delta, \overline{\alpha^{\mathrm{U}}}, \overline{\beta^{\mathrm{L}}}; \epsilon; \Gamma \vdash e : (\Sigma^\circ \to \{\})^{|\overline{\beta}|}$
— by LTG typing rules and the auxiliary lemma,
$\Delta; \epsilon; \Gamma \vdash \lambda\overline{\alpha^{\mathrm{U}}}.\lambda\overline{\beta^{\mathrm{L}}}.e : (\forall\overline{\alpha^{\mathrm{U}}}.(\forall^{\mathrm{L}}\overline{\beta}.(\Sigma^\circ \to \{\}))^{\mathrm{U}})^{\mathrm{U}}$

— Rule MATCH$^{\rightsquigarrow}$:
— to show: $\Delta; \epsilon; \epsilon \vdash f : (\forall^{\mathrm{U}}\overline{\alpha_1}.\, \forall^{\mathrm{L}}\overline{\beta_1}.\, (\Sigma_1^\circ \to \{\}))^{\mathrm{U}} \to (\forall^{\mathrm{U}}\overline{\alpha_2}.\, \forall^{\mathrm{L}}\overline{\beta_2}.\, (\Sigma_2^\circ \to \{\}))^{\mathrm{U}}$
— let $\Delta' = \Delta, \overline{\alpha_2^{\mathrm{U}}}, \overline{\beta_1^{\mathrm{U}}}$ and $\Delta'' = \Delta', \overline{\alpha_1^{\mathrm{U}}}, \overline{\beta_2^{\mathrm{U}}}$
— let $\Gamma = y : (\forall^{\mathrm{U}}\overline{\alpha_1}.\, \forall^{\mathrm{L}}\overline{\beta_1}.\, (\Sigma_1^\circ \to \{\}))^{\mathrm{U}}$ and $\Xi = \Delta'; \epsilon; \delta^\circ\Gamma$
— first show $\Delta' \vdash \delta^\circ : \Delta''$ and $\Delta' * \overline{\beta_1^{\mathrm{L}}} \vdash \delta^\circ : \Delta'' * \overline{\beta_1^{\mathrm{L}}}$
— by inverting $\Delta \vdash \Phi_1 \Uparrow$, we know $\Delta, \overline{\alpha_1^{\mathrm{U}}}, \overline{\beta_1^{\mathrm{U}}} \vdash \Sigma_1 \Uparrow$ and $\mathcal{L}_1^-$ locates $\overline{\alpha}_1$ and $\mathcal{L}_1^- \subseteq \Sigma_1$
— by inverting $\Delta \vdash \Phi_2 \Downarrow$, we know $\Delta, \overline{\alpha_2^{\mathrm{U}}}, \overline{\beta_2^{\mathrm{U}}} \vdash \Sigma_2 \Downarrow$ and $\mathcal{L}_2^-$ locates $\overline{\alpha}_2$ and $\mathcal{L}_2^- \subseteq \Sigma_2$ and $\mathcal{L}_2^+$ locates $\overline{\beta}_2$ and $\mathcal{L}_2^+ \subseteq \Sigma_2$
— by Lemma 8.3 (1), $\Delta, \overline{\alpha_2^{\mathrm{U}}}, \overline{\beta_2^{\mathrm{U}}} \vdash \Sigma_2 \Uparrow$,
— consequently, also $\Delta'' \vdash \Sigma_1 \Uparrow$ and $\Delta'' \vdash \Sigma_2 \Uparrow$

— by induction (10), $\Delta' \vdash \delta^\circ : \Delta''$
— obviously, also $\Delta' * \overline{\beta_1^{\mathrm{L}}} \vdash \delta^\circ : \Delta'' * \overline{\beta_1^{\mathrm{L}}}$
— then consider the coercions f_1 and f_2:
— by Lemma 8.3 (10, 4), $\Delta' \vdash \delta\Sigma_1 \Uparrow$ and $\Delta' \vdash -\delta\Sigma_2 \Uparrow$
— by induction (7), $|\Sigma|^\circ = {\Sigma_1'}^\circ * {\Sigma_2'}^\circ$
with $\Delta'; \epsilon; \epsilon \vdash f_1 : {\Sigma_1'}^\circ \to (\delta\Sigma_1)^\circ$ and $\Delta'; \epsilon; \epsilon \vdash f_2 : {\Sigma_2'}^\circ \to (-\delta\Sigma_2)^\circ$
— let $\Xi_1 = \Xi, x_2 : (\delta\Sigma_2)^\circ, x : {\Sigma_2'}^\circ$
and $\Xi_{11} = \Xi, x_2 : (\delta\Sigma_2)^{-\circ}, x : {\Sigma_2'}^\circ$
and $\Xi_{12} = \Xi, x_2 : (\delta\Sigma_2)^\circ, x : {\Sigma_2'}^{-\circ}$,
and $\Xi_2 = \Xi * \overline{\beta_1^{\mathrm{L}}}, x_2 : (\delta\Sigma_2)^{-\circ}, x : {\Sigma_1'}^\circ$
and $\Xi_{21} = \Xi * \overline{\beta_1^{\mathrm{L}}}, x_2 : (\delta\Sigma_2)^{-\circ}, x : {\Sigma_1'}^{-\circ}$
and $\Xi_{22} = \Xi, x_2 : (\delta\Sigma_2)^{-\circ}, x : {\Sigma_1'}^\circ$,
so that $\Xi_1 = \Xi_{11} * \Xi_{12}$ and $\Xi_2 = \Xi_{21} * \Xi_{22}$ and $\Xi_1 * \Xi_2 = \Xi, x_2 : (\delta\Sigma_2)^\circ, x : |\Sigma|^\circ$
— by weakening and LTG typing rules, $\Xi_{11} \vdash f_2\ x : (-\delta\Sigma_2)^\circ$ and $\Xi_{12} \vdash x_2 : (\delta\Sigma_2)^\circ$
— by Lemma 8.6 (2), $\Xi_1 \vdash \mathrm{Copy}(f_2\ x, x_2 : \delta\Sigma_2) : \{\}^{\mathrm{U}}$
— by LTG typing rules, the auxiliary lemma, and weakening,
$\Xi_{21} \vdash y\ \overline{\delta^\circ\alpha_1}\ \overline{\beta_1} : ((\delta\Sigma_1)^\circ \to \{\})^{|\overline{\beta_1}|}$ and $\Xi_{22} \vdash f_1\ x : (\delta\Sigma_1)^\circ$
— by LTG typing rules, $\Xi_2 \vdash y\ \overline{\delta^\circ\alpha_1}\ \overline{\beta_1}\ (f_1\ x) : \{\}$
— by Lemma 8.4 (7), $\Delta' \vdash |\Sigma| \Uparrow$
— by Lemma 8.6 (1) and weakening, $\Xi, x_2 : (\delta\Sigma_2)^{-\circ} \vdash \mathrm{Create}(|\Sigma|) : |\Sigma|^\circ$
— by LTG typing rules, $\Xi * \overline{\beta_1^{\mathrm{L}}}, x_2 : (\delta\Sigma_2)^\circ \vdash \mathsf{let}\ x{=}\mathrm{Create}(|\Sigma|)\ \mathsf{in}\ \ldots : \{\}$
— by Lemma 7.4, $\mathrm{fv}(f_1) \cap \mathrm{dom}(\delta) = \mathrm{fv}(f_2) \cap \mathrm{dom}(\delta) = \emptyset$, and so $f_1 = \delta^\circ f_1$ and $f_2 = \delta^\circ f_2$
— by Lemma 8.6 (3), $\mathrm{Create}(|\Sigma|) = \mathrm{Create}(|\delta\Sigma|)$
— consequently, $\Xi * \overline{\beta_1^{\mathrm{L}}}, x_2 : (\delta\Sigma_2)^\circ \vdash \delta^\circ(\mathsf{let}\ x{=}\mathrm{Create}(|\Sigma|)\ \mathsf{in}\ \ldots) : \{\}$
— let $\Psi = \overline{\beta_2{:=}\delta^\circ\beta_2}$
— since $\mathrm{dom}(\delta^\circ) \cap \mathrm{dom}(\Delta') = \emptyset$, it holds that $\delta^\circ\delta^\circ\tau = \delta^\circ\tau$ for all τ
— consequently, obviously $\epsilon \vdash \delta^\circ : \Psi$, and so $\Delta' * \overline{\beta_1^{\mathrm{L}}}; \epsilon \vdash \delta^\circ : \Delta'' * \overline{\beta_1^{\mathrm{L}}}; \Psi$
— by Lemma 7.15, $\Delta' * \overline{\beta_1^{\mathrm{L}}}, \overline{\beta_2^{\mathrm{U}}}; \Psi; \Gamma, x_2 : \Sigma_2^\circ \vdash \mathsf{let}\ x{=}\mathrm{Create}(|\Sigma|)\ \mathsf{in}\ \ldots) : \{\}$
— by LTG typing rules, $\Delta' * \overline{\beta_1^{\mathrm{L}}}, \overline{\beta_2^{\mathrm{L}}}; \epsilon; \Gamma, x_2 : \Sigma_2^\circ \vdash \mathsf{def}\ \overline{\beta_2{:=}\delta^\circ\beta_2}\ \mathsf{in}\ \ldots : \{\}$
— by LTG typing rules, $\Delta, \overline{\alpha_2^{\mathrm{U}}}, \overline{\beta_2^{\mathrm{L}}}; \epsilon; \Gamma, x_2 : \Sigma_2^\circ \vdash \mathsf{new}\ \overline{\beta_1}\ \mathsf{in}\ \ldots : \{\}$
— by LTG typing rules and the auxiliary lemma,
$\Delta; \epsilon; \Gamma \vdash \lambda\overline{\alpha_2^{\mathrm{U}}}.\, \lambda\overline{\beta_2^{\mathrm{L}}}.\, \lambda x_2{:}\Sigma_2^\circ.\ \ldots : (\forall^{\mathrm{U}}\overline{\alpha}_2.\forall^{\mathrm{L}}\overline{\beta}_2.\Sigma_2^\circ \to \{\})^{\mathrm{U}}$
— by LTG typing rule,
$\Delta; \epsilon; \epsilon \vdash f : (\forall^{\mathrm{U}}\overline{\alpha_1}.\, \forall^{\mathrm{L}}\overline{\beta_1}.\, (\Sigma_1^\circ \to \{\}))^{\mathrm{U}} \to (\forall^{\mathrm{U}}\overline{\alpha_2}.\, \forall^{\mathrm{L}}\overline{\beta_2}.\, (\Sigma_2^\circ \to \{\}))^{\mathrm{U}}$

REFERENCES

ABADI, M. AND CARDELLI, L. 1996. *A Theory of Objects*. Springer-Verlag, New York, NY, USA.

AHMED, A. 2006. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming (ESOP)*. Springer-Verlag, Vienna, Austria.

AHMED, A., FLUET, M., AND MORRISETT, G. 2005. A step-indexed model for substructural state. In *International Conference on Functional Programming (ICFP)*. ACM Press, Tallinn, Estonia.

ANCONA, D., FAGORZI, S., MOGGI, E., AND ZUCCA, E. 2003. Mixin modules and computational effects. In *International Colloquium on Automata, Languages and Programming (ICALP)*. Springer-Verlag, Eindhoven, The Netherlands.

ANCONA, D. AND ZUCCA, E. 1998. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science 8,* 4, 401–446.

ANCONA, D. AND ZUCCA, E. 2002. A calculus of module systems. *Journal of Functional Programming 12,* 2, 91–132.

APPEL, A. AND MCALLESTER, D. 2001. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS 23,* 5, 657–683.

BRACHA, G., AHE, P., BYKOV, V., KASHAI, Y., MADDOX, W., AND MIRANDA, E. 2010. Modules as objects in Newspeak. In *European Conference on Object-Oriented Programming (ECOOP)*. ACM Press, Maribor, Slovenia.

BRACHA, G. AND COOK, W. 1990. Mixin-based inheritance. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, Ottawa, Canada.

BRACHA, G. AND LINDSTROM, G. 1992. Modularity meets inheritance. In *International Conference on Computer Languages (ICCL)*. IEEE, Oakland, California, USA.

CRARY, K., HARPER, R., AND PURI, S. 1999. What is a recursive module? In *Principles of Language Design and Implementation (PLDI)*. ACM Press, Atlanta, Georgia, USA.

CREMET, V., GARILLOT, F., LENGLET, S., AND ODERSKY, M. 2006. A core calculus for scala type checking. In *Mathematical Foundations of Computer Science (MFCS)*. Springer-Verlag, Stará Lesná, Slovakia.

DREYER, D. 2004. A type system for well-founded recursion. In *Principles of Programming Languages (POPL)*. ACM Press, Venice, Italy.

DREYER, D. 2005. Understanding and evolving the ML module system. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA.

DREYER, D. 2007a. Recursive type generativity. *Journal of Functional Programming 17,* 4&5, 433–471.

DREYER, D. 2007b. A type system for recursive modules. In *International Conference on Functional Programming (ICFP)*. ACM Press, Freiburg, Germany.

DREYER, D., CRARY, K., AND HARPER, R. 2003. A type system for higher-order modules. In *Principles of Programming Languages (POPL)*. ACM Press, New Orleans, Louisiana, USA.

DREYER, D., HARPER, R., AND CHAKRAVARTY, M. M. T. 2007. Modular type classes. In *Principles of Programming Languages (POPL)*. ACM Press, Nice, France.

DREYER, D. AND ROSSBERG, A. 2008. Mixin' up the ML module system. In *International Conference on Functional Programming (ICFP)*. ACM Press, Victoria, Canada.

DUGGAN, D. 2002. Type-safe linking with recursive DLLs and shared libraries. *ACM Transactions on Programming Languages and Systems 24,* 6, 711–804.

DUGGAN, D. AND SOURELIS, C. 1996. Mixin modules. In *International Conference on Functional Programming (ICFP)*. ACM Press, Philadelphia, Pennsylvania, USA.

FLATT, M. AND FELLEISEN, M. 1998. Units: Cool modules for HOT languages. In *Programming Language Design and Implementation (PLDI)*. ACM Press, Montreal, Canada.

HARPER, R. 2011. *Programming in Standard ML*. Carnegie Mellon University, Pittsburgh, Pennsylvania, USA. Working draft.

HARPER, R. AND LILLIBRIDGE, M. 1994. A type-theoretic approach to higher-order modules with sharing. In *Principles of Programming Languages (POPL)*. ACM Press, Portland, Oregon, USA.

HARPER, R. AND MITCHELL, J. C. 1993. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems 15,* 2, 211–252.

HARPER, R., MITCHELL, J. C., AND MOGGI, E. 1990. Higher-order modules and the phase distinction. In *Principles of Programming Languages (POPL)*. ACM Press, San Francisco, California, USA.

HARPER, R. AND PIERCE, B. C. 2005. Design considerations for ML-style module systems. In *Advanced Topics in Types and Programming Languages*, B. C. Pierce, Ed. MIT Press, Cambridge, Massachusetts, USA.

HARPER, R. AND STONE, C. 2000. A type-theoretic interpretation of Standard ML. In *Proof, Language, and Interaction: Essays in Honor of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, Cambridge, Massachusetts, USA.

HIRSCHOWITZ, T. AND LEROY, X. 2005. Mixin modules in a call-by-value setting. *ACM Transactions on Programming Languages and Systems 27,* 5, 857–881.

IM, H., NAKATA, K., GARRIGUE, J., AND PARK, S. 2011. A syntactic type system for recursive modules. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, Portland, Oregon, USA.

JONES, M. P. 1996. Using parameterized signatures to express modular structure. In *Principles of Programming Languages (POPL)*. ACM Press, St. Petersburg Beach, Florida, USA.

LEROY, X. 1994. Manifest types, modules, and separate compilation. In *Principles of Programming Languages (POPL)*. ACM Press, Portland, Oregon, USA.

LEROY, X. 1995. Applicative functors and fully transparent higher-order modules. In *Principles of Programming Languages (POPL)*. ACM Press, San Francisco, California, USA.

LEROY, X. 2000. A modular module system. *Journal of Functional Programming 10,* 3, 269–303.

LEROY, X. 2003. A proposal for recursive modules in Objective Caml. Available online at the following URL: http://caml.inria.fr/pub/papers/xleroy-recursive_modules-03.pdf.

MACQUEEN, D. 1984. Modules for Standard ML. In *LISP and Functional Programming (LFP)*. ACM Press, Austin, Texas, USA.

MAZURAK, K., ZHAO, J., AND ZDANCEWIC, S. 2010. Lightweight linear types in System F°. In *Types in Language Design and Implementation (TLDI)*. ACM Press, Madrid, Spain.

MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. 1997. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, USA.

MONTAGU, B. AND RÉMY, D. 2009. Modeling abstract types in modules with open existential types. In *Principles of Programming Languages (POPL)*. ACM Press, Savannah, GA, USA, 354–365.

MOON, D. A. 1986. Object-oriented programming with Flavors. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, Portland, Oregon, USA.

NAKATA, K. AND GARRIGUE, J. 2006. Recursive modules for programming. In *International Conference on Functional Programming (ICFP)*. ACM Press, Portland, Oregon, USA.

NEIS, G., DREYER, D., AND ROSSBERG, A. 2011. Non-parametric parametricity. *Journal of Functional Programming 21,* 4 & 5, 497–562.

NYSTROM, N., CHONG, S., AND MYERS, A. 2004. Scalable extensibility via nested inheritance. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, Vancouver, Canada.

NYSTROM, N., QI, X., AND MYERS, A. 2006. J&: Nested intersection for scalable software composition. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, Portland, Oregon, USA.

ODERSKY, M., CREMET, V., RÖCKL, C., AND ZENGER, M. 2003. A nominal theory of objects with dependent types. In *European Conference on Object-Oriented Programming (ECOOP)*. ACM Press, Darmstadt, Germany.

ODERSKY, M. AND ZENGER, M. 2005. Scalable component abstractions. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, San Diego, California, USA.

OWENS, S. AND FLATT, M. 2006. From structures and functors to modules and units. In *International Conference on Functional Programming (ICFP)*. ACM Press, Portland, Oregon, USA.

PEYTON JONES, S. ET AL. 2003. Haskell 98 language and libraries: the revised report. *Journal of Functional Programming 13,* 1, i–255.

RAMSEY, N., FISHER, K., AND GOVEREAU, P. 2005. An expressive language of signatures. In *International Conference on Functional Programming (ICFP)*. ACM Press, Tallinn, Estonia.

ROSSBERG, A. 2003. Generativity and dynamic opacity for abstract types. In *Principles and Practice of Declarative Programming (PPDP)*. ACM Press, Uppsala, Sweden.

ROSSBERG, A. 2006. The missing link – dynamic components for ML. In *International Conference on Functional Programming (ICFP)*. ACM Press, Portland, Oregon, USA.

ROSSBERG, A. AND DREYER, D. 2008. MixML (project website). http://www.mpi-sws.org/~rossberg/mixml/.

ROSSBERG, A., RUSSO, C. V., AND DREYER, D. 2010. F-ing modules. In *Types in Language Design and Implementation (TLDI)*. ACM Press, Madrid, Spain.

RUSSO, C. V. 1998. Types for modules. Ph.D. thesis, University of Edinburgh.

RUSSO, C. V. 1999a. First-class structures for Standard ML. In *International Conference on Functional Programming (ICFP)*. ACM Press, Paris, France.

RUSSO, C. V. 1999b. Non-dependent types for Standard ML modules. In *Principles and Practice of Declarative Programming (PPDP)*. Springer-Verlag, Paris, France.

RUSSO, C. V. 2001. Recursive structures for Standard ML. In *International Conference on Functional Programming (ICFP)*. ACM Press, Florence, Italy.

STONE, C. A. AND HARPER, R. 2006. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic 7,* 4, 676–722.

WADLER, P. 1990. Linear types can change the world! In *Programming Concepts and Methods*, M. Broy and C. Jones, Eds. North Holland, Sea of Galilee, Israel.

WADLER, P. AND BLOTT, S. 1989. How to make *ad-hoc* polymorphism less *ad hoc*. In *Principles of Programming Languages (POPL)*. ACM Press, Austin, Texas, USA.

WALKER, D. 2005. Substructural type systems. In *Advanced Topics in Types and Programming Languages*, B. C. Pierce, Ed. MIT Press, Cambridge, Massachusetts, USA.