Practical Type Theory for Recursive Modules

Derek Dreyer
Toyota Technological Institute at Chicago
dreyer@tti-c.org

Abstract

There has been much work in recent years on extending ML with recursive modules. We consider two problems with the typechecking of recursive modules that have proven to be serious stumbling blocks for existing proposals. Both problems involve the interaction of recursion and data abstraction. The first, more fundamental problem is that, inside a recursive module, one may wish to define an abstract data type in a context where a name for the type already exists. We call this the *double vision* problem because it has the effect that the programmer sees two distinct versions of the same type when they should only see one. The second, more superficial problem is that the use of data abstraction inside recursive modules often requires the programmer to write duplicate signature annotations for no clear reason. We call this the *repetitive stress* problem.

In this paper, we present a recursive module calculus called RMC that addresses both of these problems. We formalize RMC using an elaboration semantics that translates recursive ML-style modules into an internal module type system. The design of the internal language exploits previous work of ours on a type-theoretic foundation for solving the double vision problem. To remedy the repetitive stress problem, our elaboration algorithm employs a novel form of bidirectional typechecking. Although our approach to elaboration generally follows the framework of Harper and Stone, in certain key details it more closely resembles the Definition of Standard ML. The RMC design thus illustrates a viable hybrid of two approaches to defining ML that are commonly viewed as incompatible.

1. Introduction

While the ML module system [14] is esteemed for its strong support for data abstraction and code reuse, it has also been criticized for lacking a feature common to less sophisticated module systems—namely, *recursive modules*. The absence of recursive modules in ML means that programmers are forced to consolidate mutually recursive code and type definitions within a single module, often at the expense of modularity. Consequently, in recent years, language researchers have proposed and implemented a variety of recursive module extensions to ML in the interest of remedying this deficiency [23, 13, 3, 18].

The author's Ph.D. thesis [3] describes several problems in the design of a recursive module construct that all of the aforementioned proposals have had to deal with in one way or another. By far the most serious of these problems is one that involves the interaction of recursion and data abstraction. Inside a recursive module, one may wish to define an abstract data type in a context where a name for the type already exists, and there is no way in traditional accounts of ML-style type generativity to connect the old name with the new definition. We call this the *double vision* problem because it has the effect that the programmer sees two distinct versions of the same type when they should only see one. (An example of the problem is given in Section 2.1.)

Double vision has proven difficult to cure. Existing recursive module proposals address the problem either by imposing severe restrictions on the use of data abstraction within recursive module definitions, or else by implementing tricky typechecking maneuvers that are difficult to explain and do not always work. (See Section 5 for an overview of the existing proposals.) Neither of these approaches is really satisfactory.

Recently, Nakata and Garrigue [18] have called attention to another, related problem involving the interaction of data abstraction and recursive modules. Nearly all recursive module extensions require a recursive module definition to be accompanied by a *forward declaration* signature specifying the components of the module that may be referred to recursively. If the module also contains uses of data abstraction via opaque signature ascription (aka *sealing*), then the programmer ends up being forced to write down the same signature annotation twice. We refer to this as the *repetitive stress* problem. Ordinarily in ML, such repetitive stress could be mitigated by the use of signature definitions, which bind signatures to shorthand identifiers. However, as we argue in Section 2.2, this common technique does not scale well to handle the general case in which the signatures in question are themselves *recursively dependent* [1]. While the repetitive stress problem is a problem of convenience and thus not as fundamental as the double vision problem, in practice it is likely to prove a major source of programmer irritation.

In this paper, we tackle both of these problems head on. Concerning the double vision problem, we have recently proposed a cure for it at the level of a core type system for *recursive type generativity* (RTG) [2]. The key idea of this type system is to separate the creation of an abstract type name from the point at which it is defined. This allows new definitions to be given to existing as-yet-undefined names, thus avoiding double vision in a way that is simple to understand and explain to the programmer. (See Section 3.1 for details.)

In the present work, we show how to lift this solution to the level of a Standard ML-style module calculus, which we call RMC (for *Recursive Module Calculus*). The formalization of RMC follows Harper and Stone's approach of defining a programmer-level *external* language (EL) by elaboration into an *internal* language type system (IL) [9]. Instead of basing our IL on a traditional module type system in the style of Harper-Lillibridge [6] and Leroy [11], as Harper and Stone do, we base it on a variant of our RTG core calculus extended with a simple form of modules-as-namespaces.

As for the repetitive stress problem, we relieve it using a novel form of bidirectional typechecking, reminiscent of work on type inference such as Pierce and Turner's *local type inference* [21] and Pottier and Gianas' *stratified type inference* [22], but different in detail. The basic idea is as follows. If a module is annotated with a certain signature—as recursive modules are required to be—then the programmer should be able to reuse information from that signature *during* the typechecking of the module itself. In particular, in RMC, if the programmer wishes to *seal* a module mod with the same signature that was already used to specify

mod in a surrounding forward declaration signature, then they need only write `seal` mod, and the signature with which to seal the module can be inferred from context. This bidirectional approach to module typechecking is easily accommodated by the Harper-Stone elaboration framework.

Aside from solving two important problems for recursive modules, the design of RMC is interesting for two other reasons. First, it addresses a longstanding gripe about ML—namely, that when a module mod is ascribed a signature sig, any algebraic datatype definitions given in sig must be duplicated in the body of mod. We observe that this datatype duplication problem is just a special case of the more general repetitive stress problem. Correspondingly, RMC provides a keyword `canonical` that instructs the elaborator to generate the canonical implementation of a module component (in particular, a datatype definition) according to its specification in the surrounding sig.

Second, while RMC elaboration follows the architecture of Harper-Stone, certain key details of the RMC formalization more closely resemble the Definition of SML [15]. In particular, signatures in RMC's internal language are very similar to *semantic objects* in the Definition, and our treatment of abstract types via explicit type names follows the style of the Definition as well. On the other hand, like Harper-Stone, the RMC elaborator translates programs into a *type system*, for which type soundness is proven syntactically by progress and preservation. This makes proof of soundness more straightforward than it is for Definition-style formalisms [4, 26]. The RMC design thus illustrates a viable hybrid of two approaches to defining ML that are commonly viewed as incompatible.

A key feature that RMC does not account for in its present form is the ability to compile mutually recursive modules separately. While the RTG calculus that forms the semantic basis of RMC's internal language was designed originally with the goal of supporting separate compilation [2], it is still unclear how best to introduce this feature into the external language.[1] I leave this as an important direction for future work.

The rest of the paper is structured as follows. In Section 2, we present an example of mutually recursive modules in order to illustrate the double vision and repetitive stress problems as well as our solutions for them. In Section 3, we define the RMC internal language type system, and explain formally how it addresses the double vision problem. In Section 4, we present the design of the RMC external language and formalize an elaboration semantics for external language programs. Finally, in Section 5, we conclude with a comparison to related work.

2. Motivating Example

In this section, we motivate the double vision and repetitive stress problems by means of an illustrative example. We also use the example to describe at a high level how RMC addresses these problems. So that this may serve as a running example throughout the paper, the details of the example are concise to the point of being contrived. For more thorough expositions of the double vision and repetitive stress problems and more realistically detailed examples, see Dreyer [3] and Nakata and Garrigue [18].

2.1 The Double Vision Problem

Consider the example in Figure 1 of two mutually recursive modules `A` and `B`, with `A` providing a type component `t` and a value component `f`, and `B` providing a type component `u` and a value component `g`. In this example, the types of both value components, `A.f` and `B.g`, refer to both type components `A.t` and `B.u`. So that we

[1] None of the existing work on extending ML with recursive modules seriously addresses this issue either.

```
signature S_A = sig
                    type u; type t;
                    val f : t -> u * t ...
                end
signature S_B = sig
                    type t; type u;
                    val g : t -> u * t ...
                end
signature S =
    rec (X) sig
      structure A : S_A where type u = X.B.u
      structure B : S_B where type t = X.A.t
    end

structure AB = rec (X : S) struct
  structure A :> S_A where type u = X.B.u = struct
    type u = X.B.u
    type t = int
    fun f (x:t) : u * t =
       let val (y,z) = X.B.g(x+3)  (* Error 1 *)
       in (y,z+5) end              (* Error 2 *)
    ...
  end
  structure B :> S_B where type t = X.A.t = struct
    type t = X.A.t
    type u = bool
    fun g (x:t) : u * t = ...X.A.f(...)...
    ...
  end
end
```

Figure 1. Problematic Recursive Module Example

may write down the signature for each module independently and bind it to a signature identifier ($\mathtt{S_A}$ and $\mathtt{S_B}$), each of these signatures includes a specification of the type component from the other module. This is a standard technique in ML programming, which Harper and Pierce have recently dubbed *fibration* [8].

When we write down the forward declaration signature `S`, we need a way to connect the two copies of each type component. For this purpose, we employ a *recursively dependent signature* (or *rds*), written `rec (X)` *sig*. The concept of the rds is due to Crary *et al.* [1], and has been adopted (with minor variations) by subsequent recursive module proposals [23, 13]. The idea is that `X` here represents the module whose signature we are in the process of defining. Using ML's `where type` mechanism, we can reify the specification of `A.u` so that it is transparently equal to `X.B.u` (and similarly so that `B.t` is transparently equal to `X.A.t`).

Now we come to the recursive module definition itself. Within the definition of module `A`, the type `t` is defined to be `int`. The function `f` takes a value `x` of type `t` as an argument (*i.e.*, an integer) and calls `X.B.g` on `x+3`. Unfortunately, this is not well-typed, because `X.B.g` expects a value of type `X.A.t`, not `t`, and `X.A.t` is not known to equal `int`. Of course, to the programmer this seems bizarre, since `X.A.t` is merely a recursive reference to `t`, so the two types should be indiscernible. This is the first instance of the double vision problem. The second instance comes on the following line of code. The call to `X.B.g` has returned a value `z` of type `X.A.t`, which the function `f` then tries to add `5` to. The typechecker will prevent it from doing so, though, for the same reason as before—`X.A.t` does not equal `int`.

The only simple solution to this problem that we are aware of is to throw away the sealing of module `A` and expose the definition of `A.t` as `int` in the forward declaration signature `S`. However, we find this approach to be unsatisfactory because it negates one of the *raisons d'être* of recursive modules, namely the ability to introduce abstraction boundaries between mutually recursive pieces of code.

```
signature S =
    rec (X) sig
      structure A : sig
        type t
        val f : t -> X.B.u * t ...
      end
      structure B : sig
        type u
        val g : X.A.t -> u * X.A.t ...
      end
    end

structure AB = rec (X : S) struct
  structure A = seal struct
    (* Within A, X.A.t = int *)
    type t = int
    fun f (x:t) : X.B.u * t =
       let val (y,z) = X.B.g(x+3) (* No more *)
       in (y,z+5) end             (* double vision *)
    ...
  end
  (* From here on, X.A.t = A.t, but both are abstract *)
  structure B = seal struct
    (* Within B, X.B.u = bool *)
    type u = bool
    fun g (x:X.A.t) : u * X.A.t = ...X.A.f(...)...
    ...
  end
end
(* From here on, AB.A.t and AB.B.u are abstract *)
```

Figure 2. Reworking of Figure 1 in RMC

2.2 The Repetitive Stress Problem

In addition to the double vision problem, the recursive module example of Figure 1 also suffers from the repetitive stress problem. In particular, the signatures used to seal `A` and `B` must each be written down twice: once in the forward declaration signature, and once at the point of sealing in the body of the recursive module. Thanks to the signature bindings for $\mathtt{S}_\mathtt{A}$ and $\mathtt{S}_\mathtt{B}$, this is admittedly a very mild case of repetitive stress, so mild in fact that it does not seem like much of a problem.

However, as the number of mutually recursive modules we wish to define increases, the repetitive stress becomes more severe. Imagine an analogous example with n mutually recursive modules, whose signatures spin a web of recursive dependencies. In this general case, the signature fibration technique does not scale well. The signature of each module may have to include specifications of $O(n)$ type components from the other modules in order to be written down independently, and those $O(n)$ types will have to be reified (using `where type`) twice for each module—once in the forward declaration and once at the point of sealing. While the situation imagined here is clearly a worst-case scenario, the fact remains that the programmer who wishes to enforce data abstraction between recursive modules is forced to write down the same information multiple times without a clear explanation of why.

2.3 Reworking the Example in RMC

Before we present the design of RMC, we want to give the reader an intuitive feel for how RMC addresses the double vision and repetitive stress problems by showing how the recursive module example from Figure 1 would be written and typechecked in RMC.

Figure 2 shows the RMC version of this example. The first thing to notice is that we have not bothered to define the signatures of `A` and `B` independently—they are written together as subspecifications of the forward declaration `S`. As a result, `A`'s and `B`'s signatures need not include duplicate copies of each other's type components.

$$\begin{aligned}
\Sigma_\mathtt{A} &= [\![\mathtt{t} : [\![= \alpha : \mathbf{T}]\!], \mathtt{f} : [\![\alpha \to \beta \times \alpha]\!], \ldots]\!] \\
\Sigma_\mathtt{B} &= [\![\mathtt{u} : [\![= \beta : \mathbf{T}]\!], \mathtt{g} : [\![\alpha \to \beta \times \alpha]\!], \ldots]\!] \\
\Sigma &= [\![\mathtt{A} : \Sigma_\mathtt{A}, \mathtt{B} : \Sigma_\mathtt{B}]\!]
\end{aligned}$$

```
new α↑T, β↑T in
  let AB = rec (X : Σ)
    [A = set α := int in
           [t = int, f = ..., ...] : Σ_A,
     B = set β := bool in
           [u = bool, g = ..., ...] : Σ_B]
    in (* rest of program *)
```

Figure 3. RMC IL Translation of Figure 2

Secondly, within the recursive module definition, note that neither of the two submodules is sealed with any explicit signature annotation. Rather, as suggested in the introduction, we simply define `A` and `B` using a new module expression of the form `seal` mod, with the intention that each submodule be sealed with whatever signature has already been prescribed for it in the forward declaration signature `S` of the (nearest) enclosing recursive module definition. With these two changes, we have addressed the repetitive stress problem by eliminating both sources of duplication—the duplication of type components due to fibration, and the duplication of signature annotations due to the enforcement of data abstraction.

As for the double vision problem, the RMC solution is not visible in any changes to the code, but rather in how the recursive module is typechecked (as indicated by the comments in Figure 2). Within the definition of `A`, `t` is defined transparently to be `int`, and thus the RMC typechecker ensures that `X.A.t` and `int` are considered equivalent throughout the typechecking of `A`. (Similarly, `X.B.u` and `bool` are considered equivalent throughout the typechecking of `B`.) Outside of `A`, the sealing of `A` ensures that `A.t` will be treated as abstract, but it will still be considered equivalent to `X.A.t`. The important thing is that at no point during the typechecking of `AB` is a type component in the module body viewed as distinct from the corresponding component of the recursive variable `X`.

As this example illustrates, RMC's approach to avoiding the double vision problem requires an unusual form of typechecking in which the information that is available about the identity of `X`'s type components changes depending on context. In the next section, we explain how this is achieved by RMC's internal type system.

3. The RMC Internal Language

We begin in Section 3.1 with a high-level description of the approach to double vision taken by the RMC internal language (IL). Then, in Sections 3.2 through 3.4, we describe the formal details of the IL.

3.1 Separating Type Creation from Type Definition

In traditional accounts of data abstraction, including both existential types [16] and ML-style module systems, one can only create a new abstract type name if one supplies a definition along with it. In the context of recursive modules, this joining together of type creation and type definition engenders the double vision problem by preventing one from providing a definition for a pre-existing type name. The key idea of the RMC IL type system (derived directly from our previous RTG type system [2]) is to separate the generation of the name for an abstract type from the definition of the type, so that the type name may be created and referred to even if its definition is not yet available.

This approach is best illustrated by example. Consider Figure 3, which exhibits the RMC IL code that would result from elaborat-

$$
\begin{array}{lrl}
\text{Type Var.'s} & & \alpha, \beta \\
\text{Type Subst.'s} & \delta & ::= \{\overline{\alpha \mapsto \mathrm{A}}\} \\
\text{Kinds} & \mathrm{K}, \mathrm{L} & ::= \mathbf{T} \mid \mathbf{T}^n \to \mathbf{T} \\
\text{Constructors} & \mathrm{A}, \mathrm{B}, \tau & ::= \alpha \mid b \mid \lambda(\overline{\alpha}).\tau \mid \mathrm{A}(\overline{\tau}) \\
\text{Type Contexts} & \Delta & ::= \emptyset \mid \Delta, \alpha \uparrow \mathrm{K} \mid \Delta, \alpha \downarrow \mathrm{K} \mid \\
 & & \quad \Delta, \alpha : \mathrm{K} = \mathrm{A} \mid \Delta, \alpha : \mathrm{K} \approx \mathrm{A} \\
\text{Base Types} & b & ::= \forall[\overline{\alpha}].\, \tau_1 \Rightarrow \tau_2 \mid \ldots \\
\text{Terms} & e & ::= \mathtt{fold}_\alpha \mid \mathtt{unfold}_\alpha \mid e_1[\overline{\mathrm{A}}](e_2) \mid \ldots
\end{array}
$$

$$
\begin{array}{rcl}
\uparrow(\Delta) & \stackrel{\text{def}}{=} & \{\alpha \mid \alpha \uparrow \mathrm{K} \in \Delta\} \\
\mathsf{basis}_\Delta(\mathrm{A}) & \stackrel{\text{def}}{=} & \mathrm{FV}(\mathsf{norm}_\Delta(\mathrm{A})) \cap \uparrow(\Delta)
\end{array}
$$

Figure 4. IL Core-Level Syntax

ing the example of Figure 2. First, note that the definitions of IL signatures $\Sigma_{\mathtt{A}}$, $\Sigma_{\mathtt{B}}$, and Σ are merely meta-level shorthand—the IL does not support signature bindings—given here to make the code more legible. The important thing to observe about these IL signatures is that they are *transparent*. The opaque type specifications present in the source signature `S` have been replaced by transparent specifications that refer to the free type variables α and β. The signature $[\![= \alpha : \mathbf{T}]\!]$ for the `t` component of $\Sigma_{\mathtt{A}}$ indicates that `t` is transparently equivalent to α, which has base kind $\mathbf{T}$. The recursive references to `X.A.t` and `X.B.u` in the types of the `f` and `g` components have been replaced by references to α and β as well.

The reason for these changes is that, in the RMC IL, *all* signatures are transparent. Unlike traditional module type systems, abstract types may not be introduced as projections from module variables (*e.g.*, `X.A.t`). Rather, abstract types take the form of explicit type variables and are introduced via the `new` construct. The first line of actual code in Figure 3 invokes this construct to create the abstract type names α and β without defining them. (Undefined type names are bound in the type context using an $\uparrow$.) The forward declaration signature Σ then refers transparently to these names.

In the recursive module body, the `seal`ed modules have been replaced by the IL's own sealing construct, called the `set` construct. For `A`, what this `set` construct does is to provide the type name α with the definition `int`, but to only make that definition visible within the body of the `set`. Within `A`'s definition, α is considered equivalent to `int`, and thus `X.A.t` is also considered equivalent to `int` since `X.A.t` is transparently equal to α. This is the key to solving the double vision problem. Upon leaving the scope of `A`'s definition, however, the identity of α is returned to its abstract state, and `A` is added to the context with signature $\Sigma_{\mathtt{A}}$. In addition, so that no subsequent code may attempt to redefine α—a critical condition for type soundness—the context binding for α is changed from $\alpha \uparrow \mathbf{T}$ to $\alpha \downarrow \mathbf{T}$. The typechecking of `B` proceeds similarly.

3.2 Syntax

Figures 4 and 5 show the syntax of the core and module levels of the RMC IL, respectively.

Core Level As in ML, type constructors either have kind $\mathbf{T}$ (the base kind of types) or are functions from n arguments of kind $\mathbf{T}$ to a single result of kind $\mathbf{T}$. Regarding notation, we use the overbar syntax to denote a sequence of zero or more objects separated by commas. For example, $\overline{\alpha}$ represents $\alpha_1, \ldots, \alpha_n$ ($n \geq 0$). Also, as a convention, we will use τ to stand for type constructors of kind $\mathbf{T}$, and A and B to stand for type constructors of any kind.

As described in the previous section, type contexts Δ may either bind type variables α as defined ($\alpha \downarrow \mathrm{K}$) or undefined ($\alpha \uparrow \mathrm{K}$). Type variables may also be bound as transparently equal to a type constructor ($\alpha : \mathrm{K} = \mathrm{A}$), or as isomorphic to one ($\alpha : \mathrm{K} \approx \mathrm{A}$). The latter binding is used to encode the functionality of ML `datatype`'s, which are abstract but provide coercion functions that fold (and unfold) values into (and out of) the abstract type. As far as type equivalence is concerned, all bindings of α are considered abstract except $\alpha : \mathrm{K} = \mathrm{A}$. To avoid the complications of *equi-recursive* types [1], we prohibit cycles among transparent type definitions.

$$
\begin{array}{lrl}
 & \text{Structure Variables} & \mathrm{X}, \mathrm{Y} \\
 & \text{Functor Variables} & \mathrm{F} \\
\\
\text{Str. Sig's} & \Sigma & ::= [\![= \mathrm{A} : \mathrm{K}]\!] \mid [\![\tau]\!] \mid [\![\overline{\ell : \Sigma}]\!] \\
\text{Fun. Sig's} & \Phi & ::= \forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \to \exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).\Sigma_2 \\
\text{Terms} & e & ::= \ldots \mid \mathtt{Val}(\mathrm{M}) \\
\text{Structures} & \mathrm{M} & ::= \mathrm{X} \mid [\mathrm{A}] \mid [e] \mid [\overline{\ell \triangleright \mathrm{X} = \mathrm{M}}] \mid \mathrm{M}.\ell \mid \\
 & & \quad \mathcal{F}[\overline{\mathrm{A}}](\mathrm{M})[\overline{\alpha}] \mid \mathtt{let}\ \mathrm{F} = \mathcal{F}\ \mathtt{in}\ \mathrm{M} \mid \\
 & & \quad \mathtt{rec}\ (\mathrm{X} : \Sigma)\ \mathrm{M} \mid \mathtt{new}\ \alpha \uparrow \mathrm{K}\ \mathtt{in}\ \mathrm{M} \mid \\
 & & \quad \mathtt{set}\ \alpha := \mathrm{A}\ \mathtt{in}\ \mathrm{M} : \Sigma \mid \mathtt{set}\ \alpha :\approx \mathrm{A}\ \mathtt{in}\ \mathrm{M} : \Sigma \\
\text{Functors} & \mathcal{F} & ::= \mathrm{F} \mid \Lambda(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\lambda(\mathrm{X} : \Sigma_1).\Lambda(\overline{\alpha_2 \uparrow \mathrm{K}_2}).\mathrm{M} \\
\text{Contexts} & \Gamma & ::= \emptyset \mid \Gamma, \mathrm{X} : \Sigma \mid \Gamma, \mathrm{F} : \Phi
\end{array}
$$

$$
\begin{array}{rcl}
\mathtt{let}\ \mathrm{X} = \mathrm{M}_1\ \mathtt{in}\ \mathrm{M}_2 & \stackrel{\text{def}}{=} & [1 \triangleright \mathrm{X} = \mathrm{M}_1, 2 = \mathrm{M}_2].2 \\
\mathtt{new}\ \overline{\alpha \uparrow \mathrm{K}}\ \mathtt{in}\ \mathrm{M} & \stackrel{\text{def}}{=} & \mathtt{new}\ \alpha_1 \uparrow \mathrm{K}_1\ \mathtt{in}\ \ldots \\
 & & \quad \mathtt{new}\ \alpha_n \uparrow \mathrm{K}_n\ \mathtt{in}\ \mathrm{M} \\
\mathtt{set}\ \overline{\alpha := \mathrm{A}}\ \mathtt{in}\ \mathrm{M} : \Sigma & \stackrel{\text{def}}{=} & \mathtt{set}\ \alpha_1 := \mathrm{A}_1\ \mathtt{in}\ \ldots \\
 & & \quad \mathtt{set}\ \alpha_n := \mathrm{A}_n\ \mathtt{in}\ \mathrm{M} : \Sigma \ldots : \Sigma
\end{array}
$$

Figure 5. IL Module-Level Syntax

This brings us to base types and terms. In the interest of isolating orthogonal concerns and focusing on the module system, the language of base types and terms is utterly minimal. The only type included here is the polymorphic coercion type, which is used to classify the fold and unfold operations for `datatype` variables. For every variable bound in the context as $\alpha : \mathrm{K} \approx \mathrm{A}$, there are coercion values $\mathtt{fold}_\alpha$ and $\mathtt{unfold}_\alpha$ associated with it. The typing of these values is described below in Figure 6.

Figure 4 also defines two useful bits of syntax: $\uparrow(\Delta)$ is the subdomain of Δ corresponding to its undefined variables, and $\mathsf{basis}_\Delta(\mathrm{A})$ is the subset of $\uparrow(\Delta)$ that A depends on. The latter is computed by $\beta\eta$-normalizing A, which involves expanding out the definitions (from Δ) of any transparent variables that A refers to. The basis function is useful when typechecking `set` expressions, in order to make sure that the `set` does not introduce a transparent type cycle into the context.

Module Level Modules are either structures and functors. For simplicity, we follow Standard ML and restrict functors to be first-order (they take structures as arguments and return structures as results) and only definable at top level (they may not be components of structures). Generalizing RMC to handle higher-order functors should not be fundamentally difficult, but we leave it to future work.

Structure signatures are written Σ. There are two atomic signatures: $[\![= \mathrm{A} : \mathrm{K}]\!]$, which describes a module containing a single type component equal to A, and $[\![\tau]\!]$, which describes a module containing a single value component of type τ. Composite structures have record signature $[\![\overline{\ell : \Sigma}]\!]$. Unlike in traditional module type systems, this record signature is not a dependent type. Dependent record types are only necessary when signatures contain opaque type specifications.

The functor signature $\forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \to \exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).\Sigma_2$ is similar to the form that functor signatures take in the Definition of SML and related formalisms such as Russo's [24], in that the type variable sequences $\overline{\alpha_1}$ and $\overline{\alpha_2}$—which represent the abstract type components of the functor argument and result, respectively—are made

Type Variable Sets $\quad \rho ::= \{\overline{\alpha}\}$
Type Effects $\quad \varphi ::= \alpha := \mathrm{A} \mid \alpha :\approx \mathrm{A} \mid \rho\downarrow$

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

$$\frac{\alpha\uparrow \mathrm{K} \in \Delta \quad \Delta \vdash \mathrm{A} : \mathrm{K} \quad \alpha \notin \mathsf{basis}_\Delta(\mathrm{A})}{\Delta \vdash \alpha := \mathrm{A} \ \mathsf{ok}} \ (1)$$

$$\frac{\alpha\uparrow \mathrm{K} \in \Delta \quad \Delta \vdash \mathrm{A} : \mathrm{K}}{\Delta \vdash \alpha :\approx \mathrm{A} \ \mathsf{ok}} \ (2) \qquad \frac{\rho \subseteq \uparrow(\Delta)}{\Delta \vdash \rho\downarrow \ \mathsf{ok}} \ (3)$$

Application of a type effect: $\Delta \mathbin{@} \varphi$

$$\begin{aligned} \Delta \mathbin{@} \alpha := \mathrm{A} &\stackrel{\text{def}}{=} \Delta\backslash\{\alpha\uparrow\Delta(\alpha)\} \cup \{\alpha : \Delta(\alpha) = \mathsf{norm}_\Delta(\mathrm{A})\} \\ \Delta \mathbin{@} \alpha :\approx \mathrm{A} &\stackrel{\text{def}}{=} \Delta\backslash\{\alpha\uparrow\Delta(\alpha)\} \cup \{\alpha : \Delta(\alpha) \approx \mathrm{A}\} \\ \Delta \mathbin{@} \rho\downarrow &\stackrel{\text{def}}{=} \Delta\backslash\{\alpha\uparrow\Delta(\alpha) \mid \alpha \in \rho\} \cup \{\alpha\downarrow\Delta(\alpha) \mid \alpha \in \rho\} \end{aligned}$$

Well-formed structures: $\Delta; \Gamma \vdash \mathrm{M} : \Sigma$ with $\rho\downarrow$

We write $\Delta; \Gamma \vdash \mathrm{M} : \Sigma$ as shorthand for $\Delta; \Gamma \vdash \mathrm{M} : \Sigma$ with $\emptyset\downarrow$.

$$\frac{\mathrm{X} : \Sigma \in \Gamma}{\Delta; \Gamma \vdash \mathrm{X} : \Sigma} \ (4) \qquad \frac{\Delta \vdash \mathrm{A} : \mathrm{K}}{\Delta; \Gamma \vdash [\mathrm{A}] : [\![= \mathrm{A} : \mathrm{K}]\!]} \ (5) \qquad \frac{\Delta; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash [e] : [\![\tau]\!]} \ (6) \qquad \frac{}{\Delta; \Gamma \vdash [] : [\![]\!]} \ (7)$$

$$\frac{\Delta; \Gamma \vdash \mathrm{M}_1 : \Sigma_1 \text{ with } \rho_1\downarrow \quad \Delta \mathbin{@} \rho_1\downarrow; \Gamma, \mathrm{X}_1 : \Sigma_1 \vdash [\overline{\ell \triangleright \mathrm{X} = \mathrm{M}}] : [\![\overline{\ell : \Sigma}]\!] \text{ with } \rho\downarrow}{\Delta; \Gamma \vdash [\ell_1 \triangleright \mathrm{X}_1 = \mathrm{M}_1, \overline{\ell \triangleright \mathrm{X} = \mathrm{M}}] : [\![\ell_1 : \Sigma_1, \overline{\ell : \Sigma}]\!] \text{ with } \rho_1, \rho\downarrow} \ (8) \qquad \frac{\Delta; \Gamma \vdash \mathrm{M} : [\![\ldots, \ell : \Sigma, \ldots]\!] \text{ with } \rho\downarrow}{\Delta; \Gamma \vdash \mathrm{M}.\ell : \Sigma \text{ with } \rho\downarrow} \ (9)$$

$$\frac{\Delta; \Gamma \vdash \mathcal{F} : \forall(\overline{\alpha_1\downarrow \mathrm{K}_1}).\Sigma_1 \to \exists(\overline{\alpha_2\downarrow \mathrm{K}_2}).\Sigma_2 \quad \overline{\Delta \vdash \mathrm{A}\downarrow \mathrm{K}_1} \quad \Delta; \Gamma \vdash \mathrm{M} : \{\overline{\alpha_1 \mapsto \mathrm{A}}\}\Sigma_1 \quad \overline{\alpha\uparrow \mathrm{K}_2} \subseteq \Delta}{\Delta; \Gamma \vdash \mathcal{F}[\overline{\mathrm{A}}](\mathrm{M})[\overline{\alpha}] : \{\overline{\alpha_1 \mapsto \mathrm{A}}\}\{\overline{\alpha_2 \mapsto \alpha}\}\Sigma_2 \text{ with } \overline{\alpha}\downarrow} \ (10)$$

$$\frac{\Delta; \Gamma \vdash \mathcal{F} : \Phi \quad \Delta; \Gamma, \mathrm{F} : \Phi \vdash \mathrm{M} : \Sigma \text{ with } \rho\downarrow}{\Delta; \Gamma \vdash \mathtt{let}\ \mathrm{F} = \mathcal{F}\ \mathtt{in}\ \mathrm{M} : \Sigma \text{ with } \rho\downarrow} \ (11) \qquad \frac{\Delta \vdash \Sigma \ \mathsf{sig} \quad \Delta; \Gamma, \mathrm{X} : \Sigma \vdash \mathrm{M} : \Sigma \text{ with } \rho\downarrow}{\Delta; \Gamma \vdash \mathtt{rec}\,(\mathrm{X} : \Sigma)\,\mathrm{M} : \Sigma \text{ with } \rho\downarrow} \ (12)$$

$$\frac{\Delta, \alpha\uparrow \mathrm{K}; \Gamma \vdash \mathrm{M} : \Sigma \text{ with } \alpha, \rho\downarrow \quad \alpha \notin \mathrm{FV}(\Sigma)}{\Delta; \Gamma \vdash \mathtt{new}\ \alpha\uparrow \mathrm{K}\ \mathtt{in}\ \mathrm{M} : \Sigma \text{ with } \rho\downarrow} \ (13) \qquad \frac{\Delta \vdash \alpha :\approx \mathrm{A} \ \mathsf{ok} \quad \Delta \mathbin{@} \alpha :\approx \mathrm{A}; \Gamma \vdash \mathrm{M} : \Sigma \text{ with } \rho\downarrow}{\Delta; \Gamma \vdash (\mathtt{set}\ \alpha :\approx \mathrm{A}\ \mathtt{in}\ \mathrm{M} : \Sigma) : \Sigma \text{ with } \alpha, \rho\downarrow} \ (14)$$

$$\frac{\Delta \vdash \alpha := \mathrm{A} \ \mathsf{ok} \quad \Delta \mathbin{@} \alpha := \mathrm{A}; \Gamma \vdash \mathrm{M} : \Sigma \text{ with } \rho\downarrow \quad \mathsf{basis}_\Delta(\mathrm{A}) \subseteq \rho}{\Delta; \Gamma \vdash (\mathtt{set}\ \alpha := \mathrm{A}\ \mathtt{in}\ \mathrm{M} : \Sigma) : \Sigma \text{ with } \alpha, \rho\downarrow} \ (15) \qquad \frac{\Delta; \Gamma \vdash \mathrm{M} : \Sigma' \text{ with } \rho\downarrow \quad \Delta \vdash \Sigma' \equiv \Sigma}{\Delta; \Gamma \vdash \mathrm{M} : \Sigma \text{ with } \rho\downarrow} \ (16)$$

Well-formed functors: $\Delta; \Gamma \vdash \mathcal{F} : \Phi$

$$\frac{\mathrm{F} : \Phi \in \Gamma}{\Delta; \Gamma \vdash \mathrm{F} : \Phi} \ (17) \qquad \frac{\Delta, \overline{\alpha_1\downarrow \mathrm{K}_1} \vdash \Sigma_1 \ \mathsf{sig} \quad \Delta, \overline{\alpha_1\downarrow \mathrm{K}_1}, \overline{\alpha_2\uparrow \mathrm{K}_2}; \Gamma, \mathrm{X} : \Sigma_1 \vdash \mathrm{M} : \Sigma_2 \text{ with } \overline{\alpha_2}\downarrow}{\Delta; \Gamma \vdash \Lambda(\overline{\alpha_1\downarrow \mathrm{K}_1}).\lambda(\mathrm{X} : \Sigma_1).\Lambda(\overline{\alpha_2\uparrow \mathrm{K}_2}).\mathrm{M} : \forall(\overline{\alpha_1\downarrow \mathrm{K}_1}).\Sigma_1 \to \exists(\overline{\alpha_2\downarrow \mathrm{K}_2}).\Sigma_2} \ (18)$$

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

$$\frac{\Delta; \Gamma \vdash \mathrm{M} : [\![\tau]\!]}{\Delta; \Gamma \vdash \mathtt{Val}(\mathrm{M}) : \tau} \ (19) \qquad \frac{\alpha : \mathbf{T} \approx \tau \in \Delta}{\Delta; \Gamma \vdash \mathtt{fold}_\alpha : \forall[].\, \tau \Rightarrow \alpha} \ (20) \qquad \frac{\alpha : \mathbf{T} \approx \tau \in \Delta}{\Delta; \Gamma \vdash \mathtt{unfold}_\alpha : \forall[].\, \alpha \Rightarrow \tau} \ (21)$$

$$\frac{\alpha : \mathbf{T}^n \to \mathbf{T} \approx \mathrm{A} \in \Delta \quad \overline{\beta}(= \beta_1, \ldots, \beta_n) \cap \mathrm{dom}(\Delta) = \emptyset}{\Delta; \Gamma \vdash \mathtt{fold}_\alpha : \forall[\overline{\beta}].\, \mathrm{A}(\overline{\beta}) \Rightarrow \alpha(\overline{\beta})} \ (22) \qquad \frac{\alpha : \mathbf{T}^n \to \mathbf{T} \approx \mathrm{A} \in \Delta \quad \overline{\beta}(= \beta_1, \ldots, \beta_n) \cap \mathrm{dom}(\Delta) = \emptyset}{\Delta; \Gamma \vdash \mathtt{unfold}_\alpha : \forall[\overline{\beta}].\, \alpha(\overline{\beta}) \Rightarrow \mathrm{A}(\overline{\beta})} \ (23)$$

$$\frac{\Delta; \Gamma \vdash e_1 : \forall[\overline{\alpha}].\, \tau_2 \Rightarrow \tau \quad \overline{\Delta \vdash \mathrm{A} : \mathbf{T}} \quad \Delta; \Gamma \vdash e_2 : \{\overline{\alpha \mapsto \mathrm{A}}\}\tau_2}{\Delta; \Gamma \vdash e_1[\overline{\mathrm{A}}](e_2) : \{\overline{\alpha \mapsto \mathrm{A}}\}\tau} \ (24) \qquad \frac{\Delta; \Gamma \vdash e : \tau' \quad \Delta \vdash \tau' \equiv \tau : \mathbf{T}}{\Delta; \Gamma \vdash e : \tau} \ (25)$$

Figure 6. IL Static Semantics

explicit. What is unusual is that a functor of this signature actually expects *three* arguments. The first argument is a sequence of type constructors $\overline{\mathrm{A}}$ to be substituted for $\overline{\alpha_1}$. The second argument is a module of signature $\{\overline{\alpha_1 \mapsto \mathrm{A}}\}\Sigma_1$. The third argument is a sequence of *undefined type variables* $\overline{\alpha}$ that take the place of $\overline{\alpha_2}$.

In other words, before calling the functor, the client of the functor is charged with creating new names $\overline{\alpha}$ corresponding to the abstract types that the functor body is going to define. These are passed to the functor, which then returns a module of signature $\{\overline{\alpha_1 \mapsto \mathrm{A}}\}\{\overline{\alpha_2 \mapsto \alpha}\}\Sigma_2$ *and* defines $\overline{\alpha}$ in the process. This *destination-passing style* account of functors is useful in handling functor applications that appear inside recursive modules, where names for their abstract type components may already exist. (Imagine the example of Figure 3 with `A` defined by a functor application instead of a sealed structure.)

An important technical point: The $\downarrow$ notation on $\overline{\alpha_1}$ indicates that the first argument to the functor should consist of *fully defined* types, *i.e.*, types that do not depend on any undefined variables. This condition is in place because the definitions of the abstract result types may depend on the argument types, and we wish to ensure that a transparent type cycle (such as defining $\alpha := \alpha \times \alpha$) does not arise. For further discussion, we refer the reader to prior work, in which this account of functors is studied in detail [2].

Given the discussion so far, the module constructs in Figure 5 are fairly self-explanatory. In the record construct

$$[\ell_1 \triangleright \mathrm{X}_1 = \mathrm{M}_1, \ldots, \ell_n \triangleright \mathrm{X}_n = \mathrm{M}_n],$$

the *labels* ℓ_i correspond to the *external* names of the structure components, which may be projected out by the construct $\mathrm{M}.\ell$. The *variables* X_i correspond to the *internal* names of the components. While the ℓ_i's are immutable, each X_i is bound in the subsequent M_j's and may be alpha-varied. The ℓ_i's and X_i's are all assumed to be distinct. This follows the syntactic formulation of structures set out by Harper and Lillibridge [6].

Lastly, the module level extends the term language with the construct $\mathtt{Val}(\mathrm{M})$, which enables the value component to be projected out of a module of atomic signature $[\![\tau]\!]$. Note that there is no corresponding construct for projecting the *type* component out of a module of atomic signature $[\![= \mathrm{A} : \mathrm{K}]\!]$. The reason is that a module with this latter signature can only have one implementation modulo type equivalence, namely $[\mathrm{A}]$. At the level of the IL, therefore, there is no need to complicate the type language when one can instead write A directly.

3.3 Static Semantics

The typing judgment for structures is $\Delta; \Gamma \vdash \mathrm{M} : \Sigma$ with $\rho\downarrow$. Here, ρ represents the set of type variables that are undefined prior to the evaluation of module M but that will become defined after M has finished executing. In fact, $\rho\downarrow$ is one of three *type effects* φ, whose syntax is given in Figure 6. The other two type effects are $\alpha := \mathrm{A}$ and $\alpha :\approx \mathrm{A}$, which represent the setting of α to be equal (resp. isomorphic) to A. We call them type effects because they engender a modification to the type context during typechecking.

Figure 6 also defines $\Delta \mathbin{@} \varphi$—syntactic sugar for the result of applying effect φ to the context Δ—and a typing judgment $\Delta \vdash \varphi$ ok, which indicates when $\Delta \mathbin{@} \varphi$ will be guaranteed to be well-formed. In short, φ may only attempt to define variables that are undefined in Δ, and the effect $\alpha := \mathrm{A}$ is only valid if it does not introduce a transparent type cycle, *i.e.*, if A does not depend on α.

For the most part, the typing rules for the IL (Figure 6) straightforwardly formalize the informal descriptions of the IL constructs given in the previous section. When a module has no type effects, we leave off the "with $\emptyset\downarrow$" from its typing judgment. In addition, throughout these rules (and the rest of the paper), we adopt the Barendregt convention that all bound variables are chosen to be distinct. Contexts never bind the same variable twice, and commas (when applied to contexts and other sets) denote disjoint union.

A few points of note: First, Rule 13 for $\mathtt{new}\ \alpha \uparrow \mathrm{K}\ \mathtt{in}\ \mathrm{M}$ requires that α not appear in the signature Σ of M because `new` binds α. Since modules have unique signatures in this language, deciding whether such a Σ exists simply involves computing a signature for M and normalizing it. Second, Rule 15 for $\mathtt{set}\ \alpha := \mathrm{A}\ \mathtt{in}\ \mathrm{M} : \Sigma$ allows A to depend on undefined variables initially, but requires that those variables be defined by the time M has finished evaluating. This is necessary since, subsequent to the evaluation of the `set` expression, α will be considered a defined variable.

3.4 Dynamic Semantics and Type Soundness

For space reasons, we omit presentation of the IL's dynamic semantics and type soundness theorem. Full details are given in Appendix B, but they are generally very similar to those of the RTG calculus (the journal version [2]). One point worth noting is that recursive modules are evaluated according to a Scheme-style backpatching semantics. In particular, to evaluate $\mathtt{rec}\ (\mathrm{X} : \Sigma)\ \mathrm{M}$, X is mapped to a fresh location in the store whose contents are undefined. M is evaluated to a value V, which is then stored at the location represented by X. If the evaluation of M attempts to evaluate X, an exception is raised at run time.

Label Seq's	ℓs	$::= \epsilon \mid \ell.\ell s$
Paths	P	$::= \mathrm{X}.\ell s$
Type Con's	con	$::= \mathrm{P} \mid \alpha \mid \lambda(\overline{\alpha}).con \mid con_1(\overline{con}) \mid \ldots$
Terms	exp	$::= \mathrm{P} \mid \ldots$
Sig. Var's	S	
Signatures	sig	$::= \mathrm{S} \mid [\![\mathrm{K}]\!] \mid [\![con]\!] \mid [\![\approx con : \mathrm{K}]\!] \mid$ $[\![\overline{\ell \triangleright \mathrm{X} : sig}]\!] \mid \mathtt{rec}\ (\mathrm{X})\ sig \mid$ $sig\ \mathtt{where\ type}\ \ell s = con$
Modules	mod	$::= \mathrm{P} \mid [con] \mid [exp] \mid [\overline{\ell \triangleright \mathrm{X} = mod}] \mid$ $\mathtt{let}\ \mathrm{X} = mod_1\ \mathtt{in}\ mod_2 \mid \mathrm{F}(\mathrm{X}) \mid$ $\mathtt{rec}\ (\mathrm{X} : sig)\ mod \mid \mathtt{canonical} \mid$ $\mathtt{seal}\ mod \mid \mathtt{coerce}\ mod$
Top-Level	top	$::= \mathtt{signature}\ \mathrm{S} = sig \mid$ $\mathtt{structure}\ \mathrm{X} = mod \mid$ $\mathtt{functor}\ \mathrm{F}(\mathrm{X} : sig) = mod$
Programs	$prog$	$::= \overline{top}$

$$
\begin{array}{rcl}
[\![= con : \mathrm{K}]\!] & \stackrel{\text{def}}{=} & [\![\mathrm{K}]\!]\ \mathtt{where\ type}\ \epsilon = con \\
\mathrm{F}(mod) & \stackrel{\text{def}}{=} & \mathtt{let}\ \mathrm{X} = mod\ \mathtt{in}\ \mathrm{F}(\mathrm{X}) \\
mod\ \mathtt{:>}\ sig & \stackrel{\text{def}}{=} & \mathtt{rec}\ (\_ : sig)\ \mathtt{seal}\ mod \\
mod : sig & \stackrel{\text{def}}{=} & \mathtt{rec}\ (\_ : sig)\ \mathtt{coerce}\ mod
\end{array}
$$

Figure 7. External Language Syntax

4. The RMC External Language

4.1 Syntax

Figure 7 gives the syntax of the RMC external language (EL). The EL is intended to be representative of a Standard ML-like module language, but it does not directly support all features of SML (or this paper would be quite a bit longer!). We focus instead on supporting the most semantically interesting features, and leave formalization of a full-fledged ML extension to future work.

In the spirit of keeping the core language as underdetermined as possible, the only interesting type- and term-level constructs considered here are *paths* P, which are sequences of projections from module variables. As a matter of notation, we will typically drop the trailing "$.\epsilon$" from a path P or label sequence ℓs.

Unlike IL signatures, EL signatures may contain both abstract and transparent type specifications. $[\![\mathrm{K}]\!]$ denotes the atomic signature of a module containing a single type component of kind K. While there is no primitive transparent type signature $[\![= con : \mathrm{K}]\!]$, we do support ML's `where type` construct, and Figure 7 shows how to define transparent type specifications as a derived form. (This is how the Definition of SML defines them as well.) Since EL signatures contain abstract type components, EL record signatures $[\![\overline{\ell \triangleright \mathrm{X} : sig}]\!]$ *are* dependently-typed, with each internal name X_i bound in the subsequent sig_j's. Note that in SML there is no label/variable distinction. The distinction is maintained here to simplify the concerns of elaboration.

The signature $[\![\approx con : \mathrm{K}]\!]$ represents an SML `datatype` specification in a manner following the interpretation of Harper and Stone. A module of this signature is viewed as having an abstract type constructor `t` of kind K, together with a constructor `in` and destructor `out` that fold/unfold values of (polymorphic instantiations of) con into/out of (polymorphic instantiations of) `t`. To be able to *use* such a `datatype`, the term language needs a mechanism for data constructor application. For space reasons, we omit this feature, as the details would closely follow Harper and Stone [9].

Of course, one of the key features of SML `datatype` specifications is that they may be recursive. In the RMC EL, the recursive aspect of `datatype`'s is encodable by placing $[\![\approx con : \mathrm{K}]\!]$ inside

Well-formed signature denotations: $\Delta \vdash \mathcal{S}\ \mathsf{ok} \qquad \Delta \vdash (\mathcal{L};\Sigma)\ \mathsf{ok}$

$$\frac{\mathrm{dom}(\mathcal{L}) = \{\overline{\alpha}\} \quad \Delta, \overline{\alpha \uparrow \mathrm{K}} \vdash (\mathcal{L};\Sigma)\ \mathsf{ok}}{\Delta \vdash \exists(\overline{\alpha \uparrow \mathrm{K}}).(\mathcal{L};\Sigma)\ \mathsf{ok}}\ (26) \qquad \frac{\Delta \vdash \Sigma\ \mathsf{sig} \quad \forall \alpha \in \mathrm{dom}(\mathcal{L}).\ \Delta; \mathrm{X} : \Sigma \vdash \mathrm{X}.\mathcal{L}(\alpha) : [\![= \alpha : \Delta(\alpha)]\!]}{\Delta \vdash (\mathcal{L};\Sigma)\ \mathsf{ok}}\ (27)$$

Type constructor elaboration: $\Delta;\Gamma \vdash con \rightsquigarrow \mathrm{A} : \mathrm{K}$

$$\frac{\Delta \vdash \alpha : \mathbf{T}}{\Delta;\Gamma \vdash \alpha \rightsquigarrow \alpha : \mathbf{T}}\ (28) \qquad \frac{\Delta;\Gamma \vdash \mathrm{P} : [\![= \mathrm{A} : \mathrm{K}]\!]}{\Delta;\Gamma \vdash \mathrm{P} \rightsquigarrow \mathrm{A} : \mathrm{K}}\ (29) \qquad \frac{\Delta, \overline{\alpha \uparrow \mathbf{T}};\Gamma \vdash con \rightsquigarrow \tau : \mathbf{T} \quad \overline{\alpha} = \alpha_1, \ldots, \alpha_n}{\Delta;\Gamma \vdash \lambda(\overline{\alpha}).con \rightsquigarrow \lambda(\overline{\alpha}).\tau : \mathbf{T}^n \to \mathbf{T}}\ (30)$$

$$\frac{\Delta;\Gamma \vdash con' \rightsquigarrow \mathrm{A} : \mathbf{T}^n \to \mathbf{T} \quad \overline{\Delta;\Gamma \vdash con \rightsquigarrow \tau : \mathbf{T}} \quad \overline{\tau} = \tau_1, \ldots, \tau_n}{\Delta;\Gamma \vdash con'(\overline{con}) \rightsquigarrow \mathrm{A}(\overline{\tau}) : \mathbf{T}}\ (31)$$

Signature elaboration: $\Delta;\Gamma \vdash sig \rightsquigarrow \mathcal{S}$

$$\frac{\mathrm{S} = \mathcal{S} \in \Gamma}{\Delta;\Gamma \vdash \mathrm{S} \rightsquigarrow \mathcal{S}}\ (32) \qquad \frac{}{\Delta;\Gamma \vdash [\![\mathrm{K}]\!] \rightsquigarrow \exists(\alpha \uparrow \mathrm{K}).(\{\alpha \mapsto \epsilon\}; [\![= \alpha : \mathrm{K}]\!])}\ (33) \qquad \frac{\Delta;\Gamma \vdash con \rightsquigarrow \tau : \mathbf{T}}{\Delta;\Gamma \vdash [\![con]\!] \rightsquigarrow \exists().(\emptyset; [\![\tau]\!])}\ (34)$$

$$\frac{\Delta;\Gamma \vdash con \rightsquigarrow \tau : \mathbf{T}}{\Delta;\Gamma \vdash [\![\approx con : \mathbf{T}]\!] \rightsquigarrow \exists(\alpha \uparrow \mathbf{T}).(\{\alpha \mapsto \mathtt{t}\}; [\![\mathtt{t} : [\![= \alpha : \mathbf{T}]\!], \mathtt{in} : [\![\forall[].\ \tau \Rightarrow \alpha]\!], \mathtt{out} : [\![\forall[].\ \alpha \Rightarrow \tau]\!]])}\ (35)$$

$$\frac{\mathrm{K} = \mathbf{T}^n \to \mathbf{T} \quad \Delta;\Gamma \vdash con \rightsquigarrow \mathrm{A} : \mathrm{K} \quad \overline{\beta} = \beta_1, \ldots, \beta_n \quad \{\alpha, \overline{\beta}\} \cap \mathrm{FV}(\mathrm{A}) = \emptyset}{\Delta;\Gamma \vdash [\![\approx con : \mathrm{K}]\!] \rightsquigarrow \exists(\alpha \uparrow \mathrm{K}).(\{\alpha \mapsto \mathtt{t}\}; [\![\mathtt{t} : [\![= \alpha : \mathrm{K}]\!], \mathtt{in} : [\![\forall[\overline{\beta}].\ \mathrm{A}(\overline{\beta}) \Rightarrow \alpha(\overline{\beta})]\!], \mathtt{out} : [\![\forall[\overline{\beta}].\ \alpha(\overline{\beta}) \Rightarrow \mathrm{A}(\overline{\beta})]\!]])}\ (36)$$

$$\frac{}{\Delta;\Gamma \vdash [\![]\!] \rightsquigarrow \exists().(\emptyset; [\![]\!])}\ (37) \qquad \frac{\Delta;\Gamma \vdash sig_1 \rightsquigarrow \exists(\overline{\alpha_1 \uparrow \mathrm{K}_1}).(\mathcal{L}_1;\Sigma_1) \quad \Delta, \overline{\alpha_1 \uparrow \mathrm{K}_1};\Gamma, \mathrm{X}_1 : \Sigma_1 \vdash [\![\overline{\ell \triangleright \mathrm{X} : sig}]\!] \rightsquigarrow \exists(\overline{\alpha \uparrow \mathrm{K}}).(\mathcal{L}; [\![\overline{\ell : \Sigma}]\!])}{\Delta;\Gamma \vdash [\![\ell_1 \triangleright \mathrm{X}_1 : sig_1, \overline{\ell \triangleright \mathrm{X} : sig}]\!] \rightsquigarrow \exists(\overline{\alpha_1 \uparrow \mathrm{K}_1}, \overline{\alpha \uparrow \mathrm{K}}).(\ell_1.\mathcal{L}_1, \mathcal{L}; [\![\ell_1 : \Sigma_1, \overline{\ell : \Sigma}]\!])}\ (38)$$

$$\frac{\Delta;\Gamma \vdash \mathsf{Shal}(sig) \rightsquigarrow \exists(\overline{\alpha_0 \uparrow \mathrm{K}_0}).(\mathcal{L}_0;\Sigma_0) \quad \Delta, \overline{\alpha_0 \uparrow \mathrm{K}_0};\Gamma, \mathrm{X} : \Sigma_0 \vdash sig \rightsquigarrow \exists(\overline{\alpha \uparrow \mathrm{K}}).(\mathcal{L};\Sigma) \quad \Delta, \overline{\alpha_0 \uparrow \mathrm{K}_0}, \overline{\alpha \uparrow \mathrm{K}} \vdash \mathsf{solve}\ \mathcal{L}_0\ \mathsf{by}\ \Sigma \rightsquigarrow \delta}{\Delta;\Gamma \vdash \mathtt{rec}\ (\mathrm{X})\ sig \rightsquigarrow \exists(\overline{\alpha \uparrow \mathrm{K}}).(\mathcal{L}; \delta\Sigma)}\ (39)$$

$$\frac{\Delta;\Gamma \vdash sig \rightsquigarrow \exists(\overline{\alpha \uparrow \mathrm{K}}).(\mathcal{L};\Sigma) \quad \beta \mapsto \ell s \in \mathcal{L} \quad \overline{\alpha \uparrow \mathrm{K}} = \overline{\alpha_1 \uparrow \mathrm{K}_1}, \beta : \mathrm{K}, \overline{\alpha_2 \uparrow \mathrm{K}_2} \quad \Delta;\Gamma \vdash con \rightsquigarrow \mathrm{B} : \mathrm{K}}{\Delta;\Gamma \vdash sig\ \mathtt{where\ type}\ \ell s = con \rightsquigarrow \exists(\overline{\alpha_1 \uparrow \mathrm{K}_1}, \overline{\alpha_2 \uparrow \mathrm{K}_2}).(\mathcal{L} \backslash \{\beta \mapsto \ell s\}; \{\beta \mapsto \mathrm{B}\}\Sigma)}\ (40)$$

Shallow version of an EL signature: $\mathsf{Shal}(sig)$

$$\begin{array}{rcl} \mathsf{Shal}(\mathrm{S}) & \stackrel{\mathrm{def}}{=} & \mathrm{S} \\ \mathsf{Shal}([\![\mathrm{K}]\!]) & \stackrel{\mathrm{def}}{=} & [\![\mathrm{K}]\!] \\ \mathsf{Shal}([\![con]\!]) & \stackrel{\mathrm{def}}{=} & [\![]\!] \\ \mathsf{Shal}([\![\approx con : \mathrm{K}]\!]) & \stackrel{\mathrm{def}}{=} & [\![\mathtt{t} : [\![\mathrm{K}]\!]]\!] \end{array} \qquad \begin{array}{rcl} \mathsf{Shal}([\![\overline{\ell \triangleright \mathrm{X} : sig}]\!]) & \stackrel{\mathrm{def}}{=} & [\![\overline{\ell : \mathsf{Shal}(sig)}]\!] \\ \mathsf{Shal}(\mathtt{rec}\ (\mathrm{X})\ sig) & \stackrel{\mathrm{def}}{=} & \mathsf{Shal}(sig) \\ \mathsf{Shal}(sig\ \mathtt{where\ type}\ \ldots) & \stackrel{\mathrm{def}}{=} & \mathsf{Shal}(sig) \end{array}$$

Figure 8. Signature Elaboration

a recursively dependent signature ($\mathtt{rec}\ (\mathrm{X})\ sig$). For example, the datatype specification for the `list` type constructor

$$\mathtt{datatype}\ \alpha\ \mathtt{list = Nil\ |\ Cons\ of}\ \alpha \times \alpha\ \mathtt{list}$$

would be encoded as:

$$\mathtt{rec}\ (\mathrm{X})\ [\![\approx \lambda(\alpha).\ \mathtt{unit} + \alpha \times \mathrm{X}.\mathtt{t}(\alpha) : \mathbf{T} \to \mathbf{T}]\!]$$

Mutually recursive `datatype` specifications are encodable via an rds with multiple submodules of signature $[\![\approx con : \mathrm{K}]\!]$.

Turning to modules, let us consider the differences from the IL. The functor application construct $\mathrm{F}(\mathrm{X})$ only mentions the module argument X. The other (type) arguments required by IL functor application are inferred by the elaborator. In addition, note that the module argument is restricted to be a variable purely for simplicity. Figure 7 shows how to encode $\mathrm{F}(mod)$ using module-level `let`.

The recursive module construct, $\mathtt{rec}\ (\mathrm{X} : sig)\ mod$, is similar syntactically to its IL analogue, but different semantically. In the IL, $\mathtt{rec}\ (\mathrm{X} : \Sigma)\ \mathrm{M}$ has signature Σ (Rule 12). In the EL, the signature sig is used internally by mod in the manner described in Section 2.3, and mod is required to be coercible to sig, but the signature of the whole recursive module is the (principal) signature of mod. In essence, the forward declaration signature is just a forward declaration—it need not specify *all* the components exported from the module, only those that need to be referred to recursively. This approach is very similar to the one taken by Russo's recursive module extension to Moscow ML [17].

The remaining three constructs only make sense when used inside a recursive module, as they all rely on the existence of an external forward declaration. As described in the introduction, `canonical` computes the canonical implementation of whatever signature has been declared for it. Such an implementation only exists if the signature in question is some composition of transparent type specifications and datatype specifications. In fact, `canonical` is the only mechanism by which one can actually *create* a `datatype` definition. To create a `datatype` definition according to a spec $[\![\approx con : \mathrm{K}]\!]$ that has not already been

$$
\begin{array}{lll}
\text{Abstract Type Locators} & \mathcal{L} ::= & \{\overline{\alpha \mapsto \ell s}\} \\
\text{Signature Denotations} & \mathcal{S} ::= & \exists(\overline{\alpha \uparrow \mathrm{K}}).(\mathcal{L};\Sigma) \\
\text{Elaboration Contexts} & \Gamma ::= & \ldots \mid \Gamma, \mathrm{S} = \mathcal{S} \mid \Gamma, \mathrm{F} : \forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).(\mathcal{L};\Sigma_1) \to \exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).\Sigma_2 \\
\text{Partially Elaborated Modules} & \mathit{pemod} ::= & \mathrm{P} \mid [\mathrm{A}] \mid [\mathit{exp}] \mid [\overline{\ell \triangleright \mathrm{X} = \mathit{pemod}}] \mid \texttt{let}\ \mathrm{X} = \mathit{pemod}_1\ \texttt{in}\ \mathit{pemod}_2 \mid \mathrm{F}[\overline{\mathrm{A}}](\mathrm{X})[\overline{\alpha}] \mid \\
& & \texttt{rec}\,(\mathrm{X}:\Sigma)\ \mathit{pemod} \mid \texttt{canonical}(\mathcal{L};\Sigma) \mid \texttt{seal}\ \mathit{mod} : (\mathcal{L};\Sigma) \mid \texttt{coerce}\ \mathit{pemod} : \Sigma
\end{array}
$$

$$
\ell.\mathcal{L} \stackrel{\text{def}}{=} \{\alpha \mapsto \ell.\ell s \mid \alpha \in \mathrm{dom}(\mathcal{L}) \wedge \mathcal{L}(\alpha) = \ell s\} \qquad \mathcal{L}.\ell \stackrel{\text{def}}{=} \{\alpha \mapsto \ell s \mid \alpha \in \mathrm{dom}(\mathcal{L}) \wedge \mathcal{L}(\alpha) = \ell.\ell s\}
$$

$$
\Sigma.\ell \stackrel{\text{def}}{=} \begin{cases} \Sigma' & \text{if } \Sigma = [\![\ldots, \ell : \Sigma', \ldots]\!] \\ [\![]\!] & \text{otherwise} \end{cases}
$$

Figure 9. Elaborator-Level Syntax

forward-declared, one can write the forward declaration in place: `rec` $(\_ : [\![\approx \mathit{con} : \mathrm{K}]\!])$ `canonical`.

The construct "`seal` mod" seals mod opaquely with whatever signature has been declared for it, and "`coerce` mod" seals mod transparently. The latter is included in order to model SML's transparent signature ascription. Although the traditional SML signature ascriptions are not included in the syntax of RMC, Figure 7 shows how to encode them directly in terms of `rec`, `seal`, and `coerce`.

Finally, a program is a sequence of top-level bindings of signatures/structures/functors to signature/structure/functor variables.

4.2 Elaboration

Signatures EL signatures are elaborated into *signature denotations* $\mathcal{S}$ of the form $\exists(\overline{\alpha \uparrow \mathrm{K}}).(\mathcal{L};\Sigma)$. Here, $\overline{\alpha \uparrow \mathrm{K}}$ represent the abstract type components of the signature, and Σ represents the signature itself (with transparent references to $\overline{\alpha}$). The *abstract type locator* $\mathcal{L}$ is a mapping from each of the variables in $\overline{\alpha}$ to a label sequence ℓs that indicates which type component of Σ was the "source" of that abstract type in the original EL signature. For example, the signature `S` from Figure 2 elaborates to

$$\exists(\alpha \uparrow \mathbf{T}, \beta \uparrow \mathbf{T}).(\{\alpha \mapsto \texttt{A.t}, \beta \mapsto \texttt{B.u}\}; \Sigma)$$

where Σ is as defined in Figure 3. (Note that α and β are bound by the denotation.) This approach to signature elaboration is modeled closely after that of the Definition of SML. The main novelty is the presence of the abstract type locator $\mathcal{L}$; we include $\mathcal{L}$ because it makes the definition of signature matching (see below) more deterministic by telling the elaborator explicitly where to look to fill in the abstract type components of a signature.

The elaboration rules for signatures are given in Figure 8. They are straightforward with the exception of Rule 39 for rds's. To compute the denotation of `rec` (X) sig, we need to come up with some signature to bind X to when typechecking sig. To do this, the first premise computes a *shallow* denotation of sig, $\exists(\overline{\alpha_0 \uparrow \mathrm{K}_0}).(\mathcal{L}_0;\Sigma_0)$, in which its type components are treated as having opaque specifications and its value components are ignored. Given this signature for X, the second premise computes the actual denotation of sig: $\exists(\overline{\alpha \uparrow \mathrm{K}}).(\mathcal{L};\Sigma)$. These two premises set up a system of equations between the "temporary" variables $\overline{\alpha_0}$, which were created to represent the type components of X, and their definitions, which appear in Σ.

To solve this system of equations, the third premise uses the solve judgment defined in Figure 11. The solve judgment uses the abstract type locator $\mathcal{L}_0$ to look up the definitions of the $\overline{\alpha_0}$ in Σ and return a (non-recursive) type substitution δ that solves for them. If there is a type cycle among the definitions, the solve will fail. For example, `rec` (X) $[\![\texttt{t} : [\![= \mathrm{X}.\texttt{t} : \mathbf{T}]\!]]\!]$ will fail to elaborate because `t` is defined transparently in terms of itself. In contrast, the signature `S` from the original version of our running example in Figure 1 will elaborate successfully—even though it contains references to the recursive variable `X` in the specifications of `A.u` and `B.t`—because those recursive references are fundamentally acyclic.

Figure 8 also defines judgments for well-formedness of signature denotations that are used in some of the module elaboration rules. Rules 26 and 27 say that $\exists(\overline{\alpha \uparrow \mathrm{K}}).(\mathcal{L};\Sigma)$ is well-formed if (1) $\mathcal{L}$ is a locator for precisely the variables $\overline{\alpha}$, (2) Σ is well-formed, and (3) if $\mathcal{L}$ maps α to ℓs, then the ℓs component of Σ is indeed equal to α. The signature elaboration judgment guarantees that all denotations it returns are well-formed according to this definition.

Modules The handling of EL modules is the most novel and complex aspect of RMC elaboration because it involves a form of bidirectional typechecking, together with a transformation into an IL where the creation and definition of abstract type names is more explicit than in ML. The high-level picture is as follows.

An EL module mod is elaborated in two phases. The first phase is a prepass over mod that handles the bidirectional aspect of typechecking by rewriting mod into the form of a *partially elaborated module*, denoted pemod, whose syntax is given in Figure 9. A pemod is essentially an EL module that has explicit signature annotations on its `canonical`, `seal`, and `coerce` subexpressions, and is thus amenable to more standard elaboration techniques. The second phase translates pemod's to IL modules.

The first phase of elaboration is represented by the judgment

$$\Delta; \Gamma; (\mathcal{L}_0;\Sigma_0) \vdash \mathit{mod} \rightsquigarrow \exists(\overline{\alpha \uparrow \mathrm{K}}).(\Sigma; \mathit{pemod})$$

This judgment looks a bit scary because it performs several interdependent functions at once, but each of the individual functions is in fact straightforward. Let us consider them each in turn.

The first function of the judgment is to produce a list of abstract type names $\overline{\alpha}$ (of kinds $\overline{\mathrm{K}}$) that mod wants to define. These are assumed to be fresh variables that are not bound in Δ. In elaborating the module from Figure 2, this judgment would produce two variables, say α and β, corresponding to `AB.A.t` and `AB.B.u`.

The second function of the judgment is to drive signature information from forward declarations down to the context-sensitive module expressions within mod that are dependent on such information in order to make sense. The input $(\mathcal{L}_0;\Sigma_0)$ represents the signature information for mod that was written down in mod's nearest enclosing forward declaration. The output pemod is essentially the original mod with the information from $(\mathcal{L}_0;\Sigma_0)$ propagated to its `canonical`, `seal`, and `coerce` subexpressions.

For example, phase-1 elaboration of the module from Figure 2 would produce the following pemod:

```
rec (X : Σ)
  [A = seal mod_A : (L_A; Σ_A),
   B = seal mod_B : (L_B; Σ_B)]
```

where $\mathcal{L}_{\texttt{A}} = \{\alpha \mapsto \texttt{t}\}$, $\mathcal{L}_{\texttt{B}} = \{\beta \mapsto \texttt{u}\}$, and Σ, $\Sigma_{\texttt{A}}$, and $\Sigma_{\texttt{B}}$ are as defined in Figure 3. Note that the bodies of `A`'s and `B`'s `seal` expressions are the original EL mod's, not pemod's. For reasons explained below, phase-1 elaboration does not proceed underneath abstraction boundaries.

The third function of the phase-1 judgment is to generate a *shallow* IL signature Σ containing IL translations of all the type com-

Elaboration of EL modules to PE modules (Phase 1): $\Delta;\Gamma;(\mathcal{L}_0;\Sigma_0)\vdash \mathit{mod}\rightsquigarrow \exists(\overline{\alpha\uparrow \mathrm{K}}).(\Sigma;\mathit{pemod})$

$$\frac{\Delta;\Gamma\vdash \mathrm{P}:\Sigma}{\Delta;\Gamma;(\mathcal{L}_0;\Sigma_0)\vdash \mathrm{P}\rightsquigarrow \exists().(\Sigma;\mathrm{P})}\ (41)\qquad \frac{\Delta;\Gamma\vdash \mathit{con}\rightsquigarrow \mathrm{A}:\mathrm{K}}{\Delta;\Gamma;(\mathcal{L}_0;\Sigma_0)\vdash [\mathit{con}]\rightsquigarrow \exists().([\![=\mathrm{A}:\mathrm{K}]\!];[\mathrm{A}])}\ (42)$$

$$\frac{}{\Delta;\Gamma;(\mathcal{L}_0;\Sigma_0)\vdash [\mathit{exp}]\rightsquigarrow \exists().([];[\mathit{exp}])}\ (43)\qquad \frac{}{\Delta;\Gamma;(\mathcal{L}_0;\Sigma_0)\vdash []\rightsquigarrow \exists().([\![]\!];[])}\ (44)$$

$$\frac{\begin{array}{c}\Delta;\Gamma;(\mathcal{L}_0.\ell_1;\Sigma_0.\ell_1)\vdash \mathit{mod}_1\rightsquigarrow \exists(\overline{\alpha_1\uparrow \mathrm{K}_1}).(\Sigma_1;\mathit{pemod}_1)\\ \Delta,\overline{\alpha_1\uparrow \mathrm{K}_1};\Gamma,\mathrm{X}_1:\Sigma_1;(\mathcal{L}_0;\Sigma_0)\vdash [\overline{\ell\triangleright \mathrm{X}=\mathit{mod}}]\rightsquigarrow \exists(\overline{\alpha\uparrow \mathrm{K}}).([\![\overline{\ell:\Sigma}]\!];[\overline{\ell\triangleright \mathrm{X}=\mathit{pemod}}])\end{array}}{\Delta;\Gamma;(\mathcal{L}_0;\Sigma_0)\vdash [\ell_1\triangleright \mathrm{X}_1=\mathit{mod}_1,\overline{\ell\triangleright \mathrm{X}=\mathit{mod}}]\rightsquigarrow \exists(\overline{\alpha_1\uparrow \mathrm{K}_1},\overline{\alpha\uparrow \mathrm{K}}).([\![\ell_1:\Sigma_1,\overline{\ell:\Sigma}]\!];[\ell_1\triangleright \mathrm{X}_1=\mathit{pemod}_1,\overline{\ell\triangleright \mathrm{X}=\mathit{pemod}}])}\ (45)$$

$$\frac{\Delta;\Gamma;(\emptyset;[\![]\!])\vdash \mathit{mod}_1\rightsquigarrow \exists(\overline{\alpha_1\uparrow \mathrm{K}_1}).(\Sigma_1;\mathit{pemod}_1)\quad \Delta,\overline{\alpha_1\uparrow \mathrm{K}_1};\Gamma,\mathrm{X}:\Sigma_1;(\mathcal{L}_0;\Sigma_0)\vdash \mathit{mod}_2\rightsquigarrow \exists(\overline{\alpha_2\uparrow \mathrm{K}_2}).(\Sigma_2;\mathit{pemod}_2)}{\Delta;\Gamma;(\mathcal{L}_0;\Sigma_0)\vdash \mathtt{let}\ \mathrm{X}=\mathit{mod}_1\ \mathtt{in}\ \mathit{mod}_2\rightsquigarrow \exists(\overline{\alpha_1\uparrow \mathrm{K}_1},\overline{\alpha_2\uparrow \mathrm{K}_2}).(\Sigma_2;\mathtt{let}\ \mathrm{X}=\mathit{pemod}_1\ \mathtt{in}\ \mathit{pemod}_2)}\ (46)$$

$$\frac{\mathrm{F}:\forall(\overline{\alpha_1\downarrow \mathrm{K}_1}).(\mathcal{L};\Sigma_1)\to \exists(\overline{\alpha_2\downarrow \mathrm{K}_2}).\Sigma_2\in\Gamma\quad \mathrm{X}:\Sigma\in\Gamma\quad \Delta,\overline{\alpha_1\uparrow \mathrm{K}_1}\vdash \mathsf{solve}\ \mathcal{L}\ \mathsf{by}\ \Sigma\rightsquigarrow\delta}{\Delta;\Gamma;(\mathcal{L}_0;\Sigma_0)\vdash \mathrm{F}(\mathrm{X})\rightsquigarrow \exists(\overline{\alpha_2\uparrow \mathrm{K}_2}).(\delta\Sigma_2;\mathrm{F}[\overline{\delta\alpha_1}](\mathrm{X})[\overline{\alpha_2}])}\ (47)$$

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash \mathit{sig}\rightsquigarrow \exists(\overline{\alpha_1\uparrow \mathrm{K}_1}).(\mathcal{L};\Sigma_1)\\ \Delta,\overline{\alpha_1\uparrow \mathrm{K}_1};\Gamma,\mathrm{X}:\Sigma_1;(\mathcal{L};\Sigma_1)\vdash \mathit{mod}\rightsquigarrow \exists(\overline{\alpha\uparrow \mathrm{K}}).(\Sigma;\mathit{pemod})\qquad \Delta,\overline{\alpha_1\uparrow \mathrm{K}_1},\overline{\alpha\uparrow \mathrm{K}}\vdash \mathsf{solve}\ \mathcal{L}\ \mathsf{by}\ \Sigma\rightsquigarrow\delta\end{array}}{\Delta;\Gamma;(\mathcal{L}_0;\Sigma_0)\vdash \mathtt{rec}\,(\mathrm{X}:\mathit{sig})\ \mathit{mod}\rightsquigarrow \exists(\overline{\alpha\uparrow \mathrm{K}}).(\delta\Sigma;\mathtt{rec}\,(\mathrm{X}:\delta\Sigma_1)\,\delta\mathit{pemod})}\ (48)$$

$$\frac{\mathcal{L}_0=\{\overline{\alpha\mapsto \ell s}\}\quad \overline{\alpha\uparrow \mathrm{K}}\subseteq\Delta\quad \overline{\beta}\cap\mathrm{dom}(\Delta)=\emptyset\quad \mathcal{L}=\{\overline{\beta\mapsto \ell s}\}\quad \Sigma=\{\overline{\alpha\mapsto\beta}\}\Sigma_0}{\Delta;\Gamma;(\mathcal{L}_0;\Sigma_0)\vdash \mathtt{canonical}\rightsquigarrow \exists(\overline{\beta\uparrow \mathrm{K}}).(\Sigma;\mathtt{canonical}(\mathcal{L};\Sigma))}\ (49)$$

$$\frac{\mathcal{L}_0=\{\overline{\alpha\mapsto \ell s}\}\quad \overline{\alpha\uparrow \mathrm{K}}\subseteq\Delta\quad \overline{\beta}\cap\mathrm{dom}(\Delta)=\emptyset\quad \mathcal{L}=\{\overline{\beta\mapsto \ell s}\}\quad \Sigma=\{\overline{\alpha\mapsto\beta}\}\Sigma_0}{\Delta;\Gamma;(\mathcal{L}_0;\Sigma_0)\vdash \mathtt{seal}\ \mathit{mod}\rightsquigarrow \exists(\overline{\beta\uparrow \mathrm{K}}).(\Sigma;\mathtt{seal}\ \mathit{mod}:(\mathcal{L};\Sigma))}\ (50)$$

$$\frac{\Delta;\Gamma;(\mathcal{L}_0;\Sigma_0)\vdash \mathit{mod}\rightsquigarrow \exists(\overline{\alpha\uparrow \mathrm{K}}).(\Sigma;\mathit{pemod})\quad \Delta,\overline{\alpha\uparrow \mathrm{K}}\vdash \mathsf{solve}\ \mathcal{L}_0\ \mathsf{by}\ \Sigma\rightsquigarrow\delta}{\Delta;\Gamma;(\mathcal{L}_0;\Sigma_0)\vdash \mathtt{coerce}\ \mathit{mod}\rightsquigarrow \exists(\overline{\alpha\uparrow \mathrm{K}}).(\delta\Sigma_0;\mathtt{coerce}\ \mathit{pemod}:\delta\Sigma_0)}\ (51)$$

Elaboration of PE modules to IL modules (Phase 2): $\Delta;\Gamma\vdash \mathit{pemod}\rightsquigarrow \mathrm{M}:\Sigma\ \mathsf{with}\ \rho\downarrow$

$$\frac{\Delta;\Gamma\vdash \mathrm{P}:\Sigma}{\Delta;\Gamma\vdash \mathrm{P}\rightsquigarrow \mathrm{P}:\Sigma}\ (52)\qquad \frac{\Delta\vdash \mathrm{A}:\mathrm{K}}{\Delta;\Gamma\vdash [\mathrm{A}]\rightsquigarrow [\mathrm{A}]:[\![=\mathrm{A}:\mathrm{K}]\!]}\ (53)\qquad \frac{\Delta;\Gamma\vdash \mathit{exp}\rightsquigarrow e:\tau}{\Delta;\Gamma\vdash [\mathit{exp}]\rightsquigarrow [e]:[\![\tau]\!]}\ (54)\qquad \frac{}{\Delta;\Gamma\vdash []\rightsquigarrow []:[]}\ (55)$$

$$\frac{\Delta;\Gamma\vdash \mathit{pemod}_1\rightsquigarrow \mathrm{M}_1:\Sigma_1\ \mathsf{with}\ \rho_1\downarrow\quad \Delta@\rho_1\downarrow;\Gamma,\mathrm{X}_1:\Sigma_1\vdash [\overline{\ell\triangleright \mathrm{X}=\mathit{pemod}}]\rightsquigarrow [\overline{\ell\triangleright \mathrm{X}=\mathrm{M}}]:[\![\overline{\ell:\Sigma}]\!]\ \mathsf{with}\ \rho\downarrow}{\Delta;\Gamma\vdash [\ell_1\triangleright \mathrm{X}_1=\mathit{pemod}_1,\overline{\ell\triangleright \mathrm{X}=\mathit{pemod}}]\rightsquigarrow [\ell_1\triangleright \mathrm{X}_1=\mathrm{M}_1,\overline{\ell\triangleright \mathrm{X}=\mathrm{M}}]:[\![\ell_1:\Sigma_1,\overline{\ell:\Sigma}]\!]\ \mathsf{with}\ \rho_1,\rho\downarrow}\ (56)$$

$$\frac{\Delta;\Gamma\vdash \mathit{pemod}_1\rightsquigarrow \mathrm{M}_1:\Sigma_1\ \mathsf{with}\ \rho_1\downarrow\quad \Delta@\rho_1\downarrow;\Gamma,\mathrm{X}:\Sigma_1\vdash \mathit{pemod}_2\rightsquigarrow \mathrm{M}_2:\Sigma_2\ \mathsf{with}\ \rho_2\downarrow}{\Delta;\Gamma\vdash \mathtt{let}\ \mathrm{X}=\mathit{pemod}_1\ \mathtt{in}\ \mathit{pemod}_2\rightsquigarrow \mathtt{let}\ \mathrm{X}=\mathrm{M}_1\ \mathtt{in}\ \mathrm{M}_2:\Sigma_2\ \mathsf{with}\ \rho_1,\rho_2\downarrow}\ (57)$$

$$\frac{\mathrm{F}:\forall(\overline{\alpha_1\downarrow \mathrm{K}_1}).(\mathcal{L};\Sigma_1)\to \exists(\overline{\alpha_2\downarrow \mathrm{K}_2}).\Sigma_2\in\Gamma\quad \overline{\Delta\vdash \mathrm{A}\downarrow \mathrm{K}_1}\quad \Delta;\Gamma\vdash \mathrm{X}\preceq\{\overline{\alpha_1\mapsto \mathrm{A}}\}\Sigma_1\rightsquigarrow \mathrm{M}\quad \overline{\alpha\uparrow \mathrm{K}_2}\subseteq\Delta}{\Delta;\Gamma\vdash \mathrm{F}[\overline{\mathrm{A}}](\mathrm{X})[\overline{\alpha}]\rightsquigarrow \mathrm{F}[\overline{\mathrm{A}}](\mathrm{M})[\overline{\alpha}]:\{\overline{\alpha_1\mapsto \mathrm{A}}\}\{\overline{\alpha_2\mapsto\alpha}\}\Sigma_2\ \mathsf{with}\ \overline{\alpha}\downarrow}\ (58)$$

$$\frac{\begin{array}{c}\Delta\vdash \Sigma_i\ \ \mathsf{sig}\quad \Delta;\Gamma,\mathrm{X}_i:\Sigma_i\vdash \mathit{pemod}\rightsquigarrow \mathrm{M}_e:\Sigma_e\ \mathsf{with}\ \rho\downarrow\quad \Delta@\rho\downarrow;\Gamma,\mathrm{X}_e:\Sigma_e\vdash \mathrm{X}_e\preceq\Sigma_i\rightsquigarrow \mathrm{M}_i\\ \Sigma=[\![\mathtt{ext}:\Sigma_e,\mathtt{int}:\Sigma_i]\!]\quad \mathrm{M}=[\mathtt{ext}\triangleright \mathrm{X}_e=\{\mathrm{X}_i\mapsto \mathrm{X}.\mathtt{int}\}\mathrm{M}_e,\ \mathtt{int}=\mathrm{M}_i]\end{array}}{\Delta;\Gamma\vdash \mathtt{rec}\,(\mathrm{X}_i:\Sigma_i)\ \mathit{pemod}\rightsquigarrow (\mathtt{rec}\,(\mathrm{X}:\Sigma)\,\mathrm{M}).\mathtt{ext}:\Sigma_e\ \mathsf{with}\ \rho\downarrow}\ (59)$$

$$\frac{\mathrm{dom}(\mathcal{L})=\{\overline{\alpha}\}\subseteq\uparrow(\Delta)\quad \Delta\vdash(\mathcal{L};\Sigma)\ \ \mathsf{ok}\quad \vdash_{\mathrm{can}}(\mathcal{L};\Sigma)\rightsquigarrow \mathrm{M}}{\Delta;\Gamma\vdash \mathtt{canonical}(\mathcal{L};\Sigma)\rightsquigarrow \mathrm{M}:\Sigma\ \mathsf{with}\ \overline{\alpha}\downarrow}\ (60)$$

$$\frac{\begin{array}{c}\mathrm{dom}(\mathcal{L}_0)=\{\overline{\alpha}\}\subseteq\uparrow(\Delta)\quad \Delta\vdash(\mathcal{L}_0;\Sigma_0)\ \ \mathsf{ok}\quad \Delta;\Gamma;(\mathcal{L}_0;\Sigma_0)\vdash \mathit{mod}\rightsquigarrow \exists(\overline{\beta\uparrow \mathrm{L}}).(\Sigma_1;\mathit{pemod})\\ \Delta,\overline{\beta\uparrow \mathrm{L}}\vdash \mathsf{solve}\ \mathcal{L}_0\ \mathsf{by}\ \Sigma_1\rightsquigarrow\delta\quad (\Delta,\overline{\beta\uparrow \mathrm{L}})@\overline{\alpha:=\delta\alpha};\Gamma\vdash \mathit{pemod}\rightsquigarrow \mathrm{M}:\Sigma\ \mathsf{with}\ \overline{\beta}\downarrow\\ (\Delta,\overline{\beta\downarrow \mathrm{L}})@\overline{\alpha:=\delta\alpha};\Gamma,\mathrm{X}:\Sigma\vdash \mathrm{X}\preceq\Sigma_0\rightsquigarrow \mathrm{M}_0\quad \forall\alpha\in\{\overline{\alpha}\}.\ \mathsf{basis}_{\Delta,\overline{\beta\uparrow \mathrm{L}}}(\delta\alpha)\subseteq\{\overline{\beta}\}\end{array}}{\Delta;\Gamma\vdash \mathtt{seal}\ \mathit{mod}:(\mathcal{L}_0;\Sigma_0)\rightsquigarrow (\mathtt{new}\ \overline{\beta\uparrow \mathrm{L}}\ \mathtt{in}\ \mathtt{set}\ \overline{\alpha:=\delta\alpha}\ \mathtt{in}\ \mathtt{let}\ \mathrm{X}=\mathrm{M}\ \mathtt{in}\ \mathrm{M}_0:\Sigma_0):\Sigma_0\ \mathsf{with}\ \overline{\alpha}\downarrow}\ (61)$$

$$\frac{\Delta;\Gamma\vdash \mathit{pemod}\rightsquigarrow \mathrm{M}:\Sigma\ \mathsf{with}\ \rho\downarrow\quad \Delta@\rho\downarrow;\Gamma,\mathrm{X}:\Sigma\vdash \mathrm{X}\preceq\Sigma_0\rightsquigarrow \mathrm{M}_0}{\Delta;\Gamma\vdash \mathtt{coerce}\ \mathit{pemod}:\Sigma_0\rightsquigarrow \mathtt{let}\ \mathrm{X}=\mathrm{M}\ \mathtt{in}\ \mathrm{M}_0:\Sigma_0\ \mathsf{with}\ \rho\downarrow}\ (62)$$

Figure 10. Module Elaboration

Abstract type lookup and equation solver: $\Delta \vdash \mathsf{solve}\ \mathcal{L}\ \mathsf{by}\ \Sigma \rightsquigarrow \delta$

$$\frac{\mathrm{dom}(\mathcal{L}) = \{\alpha_1, \ldots, \alpha_n\} \quad \forall i \in 1..n:\ \Delta; \mathrm{X} : \Sigma \vdash \mathrm{X}.\mathcal{L}(\alpha_i) : [\![= \mathrm{A}_i : \Delta(\alpha_i)]\!] \quad \mathrm{FV}(\mathsf{norm}_\Delta(\mathrm{A}_i)) \cap \mathrm{dom}(\mathcal{L}) \subseteq \{\alpha_1, \ldots, \alpha_{i-1}\} \\ \delta_0 = \mathbf{id} \quad \forall i \in 1..n:\ \delta_i = \delta_{i-1}, \alpha_i \mapsto \delta_{i-1}(\mathsf{norm}_\Delta(\mathrm{A}_i))}{\Delta \vdash \mathsf{solve}\ \mathcal{L}\ \mathsf{by}\ \Sigma \rightsquigarrow \delta_n} \quad (63)$$

Canonical module generation: $\vdash_{\mathrm{can}} (\mathcal{L}; \Sigma) \rightsquigarrow \mathrm{M}$

$$\frac{}{\vdash_{\mathrm{can}} (\emptyset; [\![= \mathrm{A} : \mathrm{K}]\!]) \rightsquigarrow [\mathrm{A}]} \quad (64) \qquad \frac{\Sigma = [\![\mathtt{t} : [\![= \alpha : \mathbf{T}]\!], \mathtt{in} : [\![\forall[]. \tau \Rightarrow \alpha]\!], \mathtt{out} : [\![\forall[]. \alpha \Rightarrow \tau]\!]]\!]}{\vdash_{\mathrm{can}} (\{\alpha \mapsto \mathtt{t}\}; \Sigma) \rightsquigarrow \mathtt{set}\ \alpha :\approx \tau\ \mathtt{in}\ [\mathtt{t} = [\alpha], \mathtt{in} = [\mathtt{fold}_\alpha], \mathtt{out} = [\mathtt{unfold}_\alpha]] : \Sigma} \quad (65)$$

$$\frac{\Sigma = [\![\mathtt{t} : [\![= \alpha : \mathbf{T}^n \to \mathbf{T}]\!], \mathtt{in} : [\![\forall[\overline{\beta}]. \mathrm{A}(\overline{\beta}) \Rightarrow \alpha(\overline{\beta})]\!], \mathtt{out} : [\![\forall[\overline{\beta}]. \alpha(\overline{\beta}) \Rightarrow \mathrm{A}(\overline{\beta})]\!]]\!]}{\vdash_{\mathrm{can}} (\{\alpha \mapsto \mathtt{t}\}; \Sigma) \rightsquigarrow \mathtt{set}\ \alpha :\approx \mathrm{A}\ \mathtt{in}\ [\mathtt{t} = [\alpha], \mathtt{in} = [\mathtt{fold}_\alpha], \mathtt{out} = [\mathtt{unfold}_\alpha]] : \Sigma} \quad (66) \qquad \frac{\overline{\vdash_{\mathrm{can}} (\mathcal{L}.\ell; \Sigma) \rightsquigarrow \mathrm{M}}}{\vdash_{\mathrm{can}} (\mathcal{L}; [\![\overline{\ell : \Sigma}]\!]) \rightsquigarrow [\overline{\ell = \mathrm{M}}]} \quad (67)$$

Signature coercions: $\Delta; \Gamma \vdash \mathrm{P} \preceq \Sigma \rightsquigarrow \mathrm{M}$

$$\frac{\Delta; \Gamma \vdash \mathrm{P} : \Sigma}{\Delta; \Gamma \vdash \mathrm{P} \preceq \Sigma \rightsquigarrow \mathrm{P}} \quad (68) \qquad \frac{\overline{\Delta; \Gamma \vdash \mathrm{P}.\ell \preceq \Sigma \rightsquigarrow \mathrm{M}}}{\Delta; \Gamma \vdash \mathrm{P} \preceq [\![\overline{\ell : \Sigma}]\!] \rightsquigarrow [\overline{\ell = \mathrm{M}}]} \quad (69)$$

Core term elaboration: $\Delta; \Gamma \vdash exp \rightsquigarrow e : \tau$

$$\frac{\Delta; \Gamma \vdash \mathrm{P} : [\![\tau]\!]}{\Delta; \Gamma \vdash \mathrm{P} \rightsquigarrow \mathtt{Val}(\mathrm{P}) : \tau} \quad (70)$$

Program elaboration: $\Delta; \Gamma \vdash prog \rightsquigarrow \mathrm{M}$

$$\frac{}{\Delta; \Gamma \vdash \emptyset \rightsquigarrow []} \quad (71) \qquad \frac{\Delta; \Gamma \vdash sig \rightsquigarrow \mathcal{S} \quad \Delta; \Gamma, \mathrm{S} = \mathcal{S} \vdash prog \rightsquigarrow \mathrm{M}_0}{\Delta; \Gamma \vdash \mathtt{signature}\ \mathrm{S} = sig,\ prog \rightsquigarrow \mathrm{M}_0} \quad (72)$$

$$\frac{\Delta; \Gamma; (\emptyset; [\![]\!]) \vdash mod \rightsquigarrow \exists(\overline{\alpha \uparrow \mathrm{K}}).(\_ ; pemod) \quad \Delta, \overline{\alpha \uparrow \mathrm{K}}; \Gamma \vdash pemod \rightsquigarrow \mathrm{M} : \Sigma\ \mathsf{with}\ \overline{\alpha}\downarrow \quad \Delta, \overline{\alpha \downarrow \mathrm{K}}; \Gamma, \mathrm{X} : \Sigma \vdash prog \rightsquigarrow \mathrm{M}_0}{\Delta; \Gamma \vdash \mathtt{structure}\ \mathrm{X} = mod,\ prog \rightsquigarrow \mathtt{new}\ \overline{\alpha \uparrow \mathrm{K}}\ \mathtt{in}\ \mathtt{let}\ \mathrm{X} = \mathrm{M}\ \mathtt{in}\ \mathrm{M}_0} \quad (73)$$

$$\frac{\Delta; \Gamma \vdash sig \rightsquigarrow \exists(\overline{\alpha_1 \uparrow \mathrm{K}_1}).(\mathcal{L}; \Sigma_1) \quad \Delta, \overline{\alpha_1 \downarrow \mathrm{K}_1}; \Gamma, \mathrm{X} : \Sigma_1; (\emptyset; [\![]\!]) \vdash mod \rightsquigarrow \exists(\overline{\alpha_2 \uparrow \mathrm{K}_2}).(\_ ; pemod) \\ \Delta, \overline{\alpha_1 \downarrow \mathrm{K}_1}, \overline{\alpha_2 \uparrow \mathrm{K}_2}; \Gamma, \mathrm{X} : \Sigma_1 \vdash pemod \rightsquigarrow \mathrm{M} : \Sigma_2\ \mathsf{with}\ \overline{\alpha_2}\downarrow \quad \Delta; \Gamma, \mathrm{F} : \forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).(\mathcal{L}; \Sigma_1) \to \exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).\Sigma_2 \vdash prog \rightsquigarrow \mathrm{M}_0}{\Delta; \Gamma \vdash \mathtt{functor}\ \mathrm{F}(\mathrm{X} : sig) = mod,\ prog \rightsquigarrow \mathtt{let}\ \mathrm{F} = \Lambda(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\lambda(\mathrm{X} : \Sigma_1).\Lambda(\overline{\alpha_2 \uparrow \mathrm{K}_2}).\mathrm{M}\ \mathtt{in}\ \mathrm{M}_0} \quad (74)$$

Figure 11. Program Elaboration and Other Judgments

ponents of mod but ignoring its value components. This shallow signature is used at several places in the elaborator. One critical place is the (phase-2) elaboration for `seal` expressions (Rule 61 in Figure 10). For instance, consider the elaboration of `seal` $mod_{\mathtt{A}}$, the definition of `A` in the example $pemod$ shown above. In order to avoid the double vision problem when typechecking $mod_{\mathtt{A}}$, it is necessary to determine first that the definition of `t` in $mod_{\mathtt{A}}$ is `int` and to install this knowledge in the context by setting $\alpha := \mathtt{int}$. The knowledge that `t` is defined as `int` comes precisely from examining the shallow signature that is returned from the phase-1 elaboration of $mod_{\mathtt{A}}$. We discuss Rule 61 in more detail below.

The second phase of elaboration is represented by the judgment

$$\Delta; \Gamma \vdash pemod \rightsquigarrow \mathrm{M} : \Sigma\ \mathsf{with}\ \rho\downarrow$$

As one might expect, it translates a $pemod$ into an IL module M with signature Σ and type effect $\rho\downarrow$. The type variables in ρ, *i.e.*, the variables that M defines, are precisely the same $\overline{\alpha}$ that the module requested to be created in the first phase of elaboration.

The rules for both module elaboration judgments are given in Figure 10. We now step through some of the more interesting ones.

Beginning with phase 1 of elaboration: Rule 45 for records is fairly self-explanatory. However, note that in the first premise, which elaborates the first component of the record (ℓ_1), the input forward declaration is pared down from $(\mathcal{L}_0; \Sigma_0)$ to $(\mathcal{L}_0.\ell_1; \Sigma_0.\ell_1)$. The latter, which relies on meta-level macros defined in Figure 9, represents the forward declaration corresponding to the ℓ_1 component of the original forward declaration, if one exists. If one does not exist, $(\mathcal{L}_0.\ell_1; \Sigma_0.\ell_1)$ will be the empty signature $(\emptyset; [\![]\!])$.

Rule 46 for module-level `let` is similar, but observe that the input forward declaration is only applied to the *body* of the `let`. The `let`-bound variable X does not have an associated label (external name), so there can be no forward declaration for it.

Rule 47 for functor applications $\mathrm{F}(\mathrm{X})$ infers the first type argument to F by using the solve judgment to look up the appropriate type components in the signature of X. In order to do this, we need an abstract type locator $\mathcal{L}$ for the argument signature of the functor. We therefore instrument functor bindings in elaboration contexts so as to include such an $\mathcal{L}$ (see the extended syntax of Γ in Figure 9).

Rule 48 for `rec` $(\mathrm{X} : sig)\ mod$ is strikingly similar to the elaboration rule for rds's (Rule 39). In particular, $\overline{\alpha_1}$, the abstract type components of sig, serve as temporary representatives of the actual type components of mod. The latter are then computed by the phase-1 elaboration of mod, which is performed using sig as the forward declaration. Finally, the solve judgment is used to solve the system of type equations between $\overline{\alpha_1}$ and their definitions.

This rule has the effect of making the forward declaration of the output recursive module transparent (possibly with respect to some fresh abstract type variables). In particular, if a type component `t` is specified abstractly in the original sig, but is defined transparently as τ in mod, then the new forward declaration ($\delta\Sigma_1$ in Rule 48) will fill in `t`'s specification so that it is transparently equal to τ. This is important for avoiding double vision—if `t` is transparently equal to `int` in mod, then X.`t` should equal `int` as well.

Rules 49 and 50 for `canonical` and `seal` mod are cutoff points for phase-1 elaboration. There is no need to proceed further for these constructs because their signatures are determined completely

from context. Thus, in both rules, a fresh set of type variables ($\overline{\beta}$) is created by which to rename the abstract type components of the given forward declaration signature, and the construct is turned into a $pemod$ by annotating it with said signature.

In contrast, Rule 51 for `coerce` mod is not a cutoff point because the signature of `coerce` mod may depend on type information in mod that is not in the forward declaration $(\mathcal{L}_0; \Sigma_0)$. The first premise collects that type information in the signature Σ. The second premise then uses the solve judgment to fill in the abstract type components of Σ_0 with their definitions in Σ. The resulting signature $\delta\Sigma_0$ thus allows the identity of mod's type components to leak through the opaque type specs in the forward declaration.

Continuing on to phase 2 of elaboration: Rule 58 for functor applications $\mathrm{F}[\overline{\mathrm{A}}](\mathrm{X})[\overline{\alpha}]$ checks that the type arguments $\overline{\mathrm{A}}$ are fully defined (have basis $\emptyset$), and employs the signature coercion judgment to coerce X to the (instantiated) argument signature of F. The signature coercion judgment is defined in Figure 11; it copies components from a module path (in this case, the variable X) in order to match a given transparent target signature, and checks that the module it returns is well-formed with that signature.

Rule 59 implements the semantics for EL recursive modules that we described informally in Section 4.1. In particular, the signature of the resulting IL module exports all the components in the body of the recursive module, not just those that were forward-declared.

Rule 60 computes the canonical module of the given signature by invoking the canonical module generation judgment defined in Figure 11. The rules defining this judgment are straightforward.

Rule 61 for `seal` $mod : (\mathcal{L}_0; \Sigma_0)$ is a veritable melting pot of judgments. The first premise checks that the externally-created variables $\overline{\alpha}$ that the module is supposed to define are in fact currently undefined. (This is a redundant check; we include it simply to make the proof of soundness completely obvious.) Similarly, the second premise checks the validity of the forward declaration that has been supplied for mod. The third premise performs phase-1 elaboration on mod, returning fresh abstract variables $\overline{\beta}$ that mod wants to define, the annotated $pemod$, and Σ_1, which specifies mod's type components.

The fourth premise then uses Σ_1 to solve for the $\overline{\alpha}$ that the `seal` is supposed to define. (In our running example, this step in the elaboration of `seal` $mod_{\mathtt{A}}$ would return the substitution $\{\alpha \mapsto \mathtt{int}\}$.) After applying the definitions for $\overline{\alpha}$ to the type context, the fifth premise proceeds to phase-2 elaborate $pemod$, producing IL module M with signature Σ. Coherence of the two phases of elaboration ensures that M will always define the variables $\overline{\beta}$ that mod requested to be created. Finally, we coerce M to the sealing signature and check that the definitions for $\overline{\alpha}$ are fully defined at the point of sealing, as demanded by the IL typing rule for sealing (Rule 15).

Lastly, Rule 62 for `coerce` $pemod : \Sigma_0$ is much simpler than the previous rule because it does not enforce any data abstraction. It simply elaborates $pemod$ and coerces the result to the transparent target signature Σ_0.

Programs Figure 11 shows the rules for elaboration of programs, which consist of a sequence of top-level bindings. The program elaboration judgment has the form $\Delta; \Gamma \vdash prog \rightsquigarrow \mathrm{M}$, where the output M is always a module of unit signature $[\![]\!]$.

Rule 73 handles the binding `structure` $\mathrm{X} = mod$ by first phase-1 elaborating mod to $pemod$, then creating the variables $\overline{\alpha}$ that mod requested, and finally phase-2 elaborating $pemod$ to M. Note that the first phase of elaboration is not given any forward declaration because mod is at top level and does not have one.

Rule 74 handles the binding `functor` $\mathrm{F}(\mathrm{X} : sig) = mod$. It is similar to the previous rule, except that instead of creating fresh abstract types for mod using `new`, it follows destination-passing style and asks the client of the functor to provide them.

Soundness Proving soundness of elaboration—*i.e.*, that if elaboration of a program $prog$ succeeds and outputs an IL module M, then M is well-formed—is completely straightforward. We have specified most invariants of elaboration informally in the above discussion; for space reasons, we omit any further reiteration of them. Full formal details are given in Appendix C.

5. Related Work

Flatt and Felleisen [5] describe a language of *units*, recursive modules for Scheme. They show how to extend them with type components, and their solution successfully avoids the double vision problem, but the unit constructs are syntactically heavyweight and awkward to use. In more recent work, Owens and Flatt [20] invest the unit language with features of ML modules (*e.g.*, translucent signatures), introduce a distinction between first-class recursive units and second-class hierarchical modules, and show how to encode a subset of the ML module system in their revised unit language. Their units remain verbose, however, and they do not provide any concrete proposal for extending ML with recursive modules.

Crary, Harper and Puri [1] give a foundational type theory for recursive modules, in which they observe the double vision problem and set forth several important concepts, including the idea of a recursively dependent signature. However, their language, which is based on Harper, Mitchell and Moggi's phase-distinction calculus [7], does not support data abstraction via sealing, nor is typechecking for it clearly decidable. Their solution to the double vision problem is to require forward declarations to be transparent.

Russo [23] defines a recursive module extension to Standard ML, which he has implemented in the Moscow ML compiler [17]. As discussed in Section 4.1, we follow Russo in allowing (EL) recursive modules to export components that are not forward-declared. (This is a feature not shared by other recursive module proposals, such as Leroy's [13] or the author's own previous proposal [3], discussed below.) Russo formalizes his extension in the style of his thesis [24], which is itself in the tradition of the Definition of SML. While Russo's formalism is very concise, his Definition-style framework is not amenable to standard syntactic techniques for proving type soundness. Russo describes the difficulties in proving soundness, but leaves the proof to future work.

Although Russo does not explicitly require forward declarations to be transparent, other restrictions of his system implicitly do. In particular, his typing rule for `rec` $(\mathrm{X} : sig)\ mod$ demands that, if a type component `t` is forward-declared abstractly in sig, then mod must define `t` to *be* `X.t`, thus clearly avoiding double vision. In the case that `t` is specified by a `datatype` specification, this means that `t` must be defined in mod by SML's `datatype` copying construct. (Thus, Russo's recursive modules offer a different solution to the `datatype` replication problem than the one provided by our `canonical` construct.) However, if `t` is specified in sig by an ordinary opaque specification (`type t`), then the only way to define it in mod is to write `type t = X.t`, in which case `t` never gets defined. As a result, one can never forward-declare a type component and also hold its definition abstract within the recursive module. In particular, our running example is not encodable in Moscow ML.

Leroy [13] presents informally a recursive module extension that he implemented for OCaml [12]. Unlike Russo's extension, Leroy's uses the forward declaration as the signature with which the whole module is sealed. To permit opaque type specifications in said signature, he describes a typechecking algorithm that is superficially similar to the approach to avoiding double vision taken by RMC elaboration. However, his algorithm only attempts to avoid double vision for type components that are defined (underneath the sealing) as `datatype`'s. His avoidance of double vision does not extend to types defined by transparent bindings (such as `type t = int`) or types that arise from nested uses of sealing.

The author's Ph.D. thesis [3] presents a recursive module extension to SML in the style of Harper and Stone [9] that is implemented in the TILT compiler [25]. The extension is similar to Leroy's in treating the forward declaration signature as a sealing signature. Concerning double vision, elaboration attempts to circumvent it through an extremely complex typechecking algorithm. The algorithm constructs a "meta-signature" representing the type information known at different points in the body of the recursive module, and then switches between public and private versions of this meta-signature as the typechecker crosses abstraction boundaries. We believe this algorithm fully avoids double vision, but it is difficult to know for sure. In contrast, RMC elaboration is much simpler to follow. Moreover, in RMC, the IL translation of an EL recursive module preserves uses of abstraction within it, whereas the elaboration of recursive modules in the author's thesis throws away all uses of sealing after typechecking. The RMC elaborator thus provides a more faithful interpretation of the source program, as evidenced by the IL code of Figure 3. Lastly, the TILT extension does not address the repetitive stress problem.

Nakata and Garrigue [18] propose a recursive module extension to ML, called *Traviata*, that is significantly different from other proposals in that it does not require recursive modules to have any forward declaration at all. Instead, the typechecker for *Traviata* performs two passes: a "reconstruction" pass, followed by a "type-correctness" check. The former traverses the whole program, collecting type information about all program identifiers, and apparently (we believe) giving globally unique names to all bound variables. Given this information about the program, the latter pass does relatively ordinary typechecking. In order for the first pass of their algorithm to collect type information about terms before they have been typechecked, terms are explicitly annotated with their types. The authors briefly sketch a type inference algorithm they have implemented to avoid requiring explicitly-typed core terms in practice, but it cannot succeed in all cases due to the undecidability of inference in the presence of polymorphic recursion [10].

One of the main reasons other proposals, including ours, employ forward declarations is to avoid imposing any demands on type inference for the core language. Nakata and Garrigue claim that the absence of forward declarations makes recursive modules easier to use because it provides a clear solution to the repetitive stress problem. We believe that forward declarations are helpful, though, in making recursive module code more readable.

As for the double vision problem, Nakata and Garrigue's typechecker does not solve it. They sketch a workaround involving type coercions that coerce between "double visions" of the same type (*e.g.*, between `X.A.t` and `int` in our running example). They admit that such type coercions are not a completely satisfactory solution.

The lack of expressiveness in *Traviata* that is engendered by the double vision problem is counterbalanced by an increase in expressiveness with respect to recursive type definitions. In particular, type definitions in their recursive modules are permitted to be cyclic, as long as the cycle is broken by an abstraction boundary. For example, consider the recursive module `rec` `(X)` $(mod$ `:>` $sig)$. In *Traviata*, if sig specifies `t` abstractly, then a definition in mod such as `type t = X.t * X.t` would be considered legal.

Nakata and Garrigue's ability to express such recursive type definitions is intricately tied to their failure to solve double vision. If they were to avoid the double vision problem, then `X.t` in the above example would be transparently equal to `X.t * X.t`, and *Traviata* treats such a *transparent* type cycle as illegal. In contrast, we cure the double vision problem but do not permit any potential cycles in type definitions, even if they are hidden by abstraction.

One consequence of this is that the typechecking of certain constructs in RMC is somewhat conservative. For instance, inside a recursive module, we only permit functor applications where the type components of the argument module are fully defined. As demonstrated in [2], our approach is sufficient to typecheck common uses of functors in recursive modules (*e.g.,* Okasaki's *bootstrapped heap* example [19]), but is not as liberal as possible. Finding a unifying compromise between our approach and Nakata and Garrigue's is a worthwhile avenue for future work.

References

[1] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *PLDI '99*.

[2] Derek Dreyer. Recursive type generativity. To appear in Journal of Functional Programming. Original version appeared in ICFP '05. Draft of journal version available from the author's website.

[3] Derek Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 2005.

[4] Martin Elsman. *Program Modules, Separate Compilation, and Intermodule Optimisation*. PhD thesis, University of Copenhagen, 1999.

[5] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *PLDI '98*.

[6] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL '94*.

[7] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *POPL '90*.

[8] Robert Harper and Benjamin C. Pierce. Design considerations for ml-style module systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[9] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.

[10] Fritz Henglein. Type inference with polymorphic recursion. *TOPLAS*, 15(2):253–289, 1993.

[11] Xavier Leroy. Manifest types, modules, and separate compilation. In *POPL '94*.

[12] Xavier Leroy. The Objective Caml system: Documentation and user's manual. `http://caml.inria.fr/ocaml/htmlman/`.

[13] Xavier Leroy. A proposal for recursive modules in Objective Caml, May 2003. Available from the author's website.

[14] David MacQueen. Modules for Standard ML. In *LFP '84*.

[15] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[16] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *TOPLAS*, 10(3):470–502, 1988.

[17] Moscow ML. `http://www.dina.dk/~sestoft/mosml.html`.

[18] Keiko Nakata and Jacques Garrigue. Recursive modules for programming. In *ICFP '06*.

[19] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[20] Scott Owens and Matthew Flatt. From structures and functors to modules and units. In *ICFP '06*.

[21] Benjamin C. Pierce and David N. Turner. Local type inference. *TOPLAS*, 22(1):1–44, 2000.

[22] François Pottier and Yann-Régis Gianas. Stratified type inference for generalized algebraic data types. In *POPL '06*.

[23] Claudio V. Russo. Recursive structures for Standard ML. In *ICFP '01*.

[24] Claudio V. Russo. *Types for Modules*. PhD thesis, University of Edinburgh, 1998.

[25] The TILT Compiler for SML. `http://www.tilt.cs.cmu.edu`.

[26] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, 1988.

A. IL Meta-Theory

For the purpose of proving type soundness, we extend the syntax of type contexts with a binding of the form $\alpha : \mathrm{K}$, which indicates that α has kind K but its definedness is unknown. Although we have no use for this binding in our IL, the meta-theory of the RTG calculus (on which the IL's meta-theory is closely based) makes use of it at certain points (*e.g.*, the "use it or lose it" lemma in [2]).

The new binding requires two minor changes to the static semantics of the IL. First, the definition of $\mathsf{basis}_\Delta(\mathrm{A})$ becomes

$$\mathsf{basis}_\Delta(\mathrm{A}) \stackrel{\text{def}}{=} \mathrm{FV}(\mathsf{norm}_\Delta(\mathrm{A})) \cap \{\alpha \mid \alpha \uparrow \mathrm{K} \in \Delta \vee \alpha : \mathrm{K} \in \Delta\}$$

Second, for technical reasons explained in [2], it is necessary to rewrite Rule 1 as follows:

$$\frac{\alpha \uparrow \mathrm{K} \in \Delta \quad \Delta \vdash \mathrm{A} : \mathrm{K} \quad \mathsf{basis}_\Delta(\mathrm{A}) \subseteq \uparrow(\Delta) \backslash \{\alpha\}}{\Delta \vdash \alpha := \mathrm{A}\ \ \mathsf{ok}} \quad (1)$$

These changes do not affect the well-formedness of programs written in the IL absent the $\alpha : \mathrm{K}$ binding. That is, IL programs that typecheck under the static semantics described in the main body of this paper will continue to typecheck under the new definition of $\mathsf{basis}_\Delta(\mathrm{A})$ and Rule 1 given here.

The meta-theory of the IL follows the meta-theory of the RTG calculus described in [2] very closely. The major difference is that the IL has a module level and a term level, whereas the RTG calculus only has one (term) level. This is a largely superficial difference, however, since the complexity of the type system derives from the treatment of abstract types via type effects, which is essentially the same in both languages. We therefore restrict attention to the few minor differences between the meta-theory of the two languages, and refer the reader to [2] for further details.

First, in the body of this paper, we refer to two signature judgments: a well-formedness judgment, $\Delta \vdash \Sigma\ \ \mathsf{sig}$, and an equivalence judgment, $\Delta \vdash \Sigma_1 \equiv \Sigma_2$. These are both defined in the obvious way: Σ is well-formed in Δ if all its constituent type constructor parts are well-formed (with the appropriate kinds) in Δ, and two signatures are equivalent if they have the same structure and their constituent type constructors are all equivalent. Although we did not refer to it in the IL typing rules, there is also a notion of well-formedness for functor signatures, $\Delta \vdash \Phi\ \ \mathsf{sig}$, defined as follows:

$$\frac{\Delta, \overline{\alpha_1 \downarrow \mathrm{K}_1} \vdash \Sigma_1\ \ \mathsf{sig} \quad \Delta, \overline{\alpha_1 \downarrow \mathrm{K}_1}, \overline{\alpha_2 \downarrow \mathrm{K}_2} \vdash \Sigma_2\ \ \mathsf{sig}}{\Delta \vdash \forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \to \exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).\Sigma_2\ \ \mathsf{sig}}$$

Second, we write $\Delta \vdash \Gamma\ \ \mathsf{ok}$ to mean that (1) $\vdash \Delta\ \ \mathsf{ok}$, (2) for all $\mathrm{X} : \Sigma \in \Gamma$, $\Delta \vdash \Sigma\ \ \mathsf{sig}$, and (3) for all $\mathrm{F} : \Phi \in \Gamma$, $\Delta \vdash \Phi\ \ \mathsf{sig}$. (The definition of $\vdash \Delta\ \ \mathsf{ok}$, which is given formally in [2], ensures that there are no cyclic dependencies between the definitions of transparent type variables, although cycles *are* permitted between the definitions of datatype variables.) In the sequel, we will write "$\Delta \Vdash \mathcal{J}$" to mean $\vdash \Delta\ \ \mathsf{ok}$ and $\Delta \vdash \mathcal{J}$, and "$\Delta; \Gamma \Vdash \mathcal{J}$" to mean $\Delta \vdash \Gamma\ \ \mathsf{ok}$ and $\Delta; \Gamma \vdash \mathcal{J}$.

Third, in RTG, variables are treated as values. In the IL, we make no such assumption, so there is no need to restrict substitutions for module variables to be value substitutions. Otherwise, the definition of well-formedness for module substitutions is similar to the one for value substitutions in RTG:

Definition A.1 (Well-Formed Module Substitutions)
We say that a module substitution γ maps Γ to Γ' under Δ, written $\Delta; \Gamma' \vdash \gamma : \Gamma$, if:

1. $\mathrm{dom}(\gamma) \subseteq \mathrm{dom}(\Gamma)$
2. $\Delta \vdash \Gamma\ \ \mathsf{ok}$ and $\Delta \vdash \Gamma'\ \ \mathsf{ok}$
3. $\forall \mathrm{X} : \Sigma \in \Gamma.\ \Delta; \Gamma' \vdash \gamma\mathrm{X} : \Sigma$
4. $\forall \mathrm{F} : \Phi \in \Gamma.\ \Delta; \Gamma' \vdash \gamma\mathrm{F} : \Phi$

The important point here is that the modules in the substitution must have the same type effects as the variables for which they are being substituted. Thus, since variables do not have any effects (Rule 4), the modules in the substitution may not either.

Finally, the meta-theory of RTG includes various lemmas concerning judgments of the form $\Delta; \Gamma \vdash e : \tau$ with $\rho\downarrow$. Since the IL has structures, functors *and* terms, all such lemmas are effectively divided into three parts, concerning judgments of the form $\Delta; \Gamma \vdash \mathrm{M} : \Sigma$ with $\rho\downarrow$, $\Delta; \Gamma \vdash \mathcal{F} : \Phi$, and $\Delta; \Gamma \vdash e : \tau$, respectively. (For the latter two judgment forms, $\rho = \emptyset$.) The proofs of all these lemmas are completely analogous to the proofs of the corresponding lemmas in [2].

B. IL Dynamic Semantics and Type Soundness

Figure 12 gives the dynamic semantics of the IL in the style of an abstract machine. Machine states consist of a type context Δ (containing the information about type names that have been created), a value store ω (containing the potentially-undefined values of recursive module variables), a continuation stack $\mathbb{C}$, and lastly either a term e or module M representing the entity currently being evaluated. We write $\omega \mathbin{@} \mathrm{X} := \mathrm{V}$ to represent the store resulting from replacing $\mathrm{X} \mapsto \mathbf{?}$ in ω with $\mathrm{X} \mapsto \mathrm{V}$.

The structure of the type soundness proof is very similar to the one given in [2]. Figure 13 defines well-formedness judgments for the various entities in the dynamic semantics, based on the corresponding judgments formalized in [2]. The relevant theorems and lemmas are as follows.

Theorem B.1 (Preservation)
If $\vdash \Omega\ \ \mathsf{ok}$ and $\Omega \rightarrowtail \Omega'$, then $\vdash \Omega'\ \ \mathsf{ok}$.

Definition B.2 (Terminal States)
A machine state Ω is *terminal* if it is of the form $\mathsf{BlackHole}$ or $(\Delta;\ \omega;\ \bullet;\ \mathrm{V})$ or $(\Delta;\ \omega;\ \bullet;\ v)$.

Definition B.3 (Stuck States)
A machine state Ω is *stuck* if it is not terminal and there is no state Ω' such that $\Omega \rightarrowtail \Omega'$.

Lemma B.4 (Canonical Forms)
Suppose $\Delta; \Gamma \Vdash v : \tau$ and $\Delta; \Gamma \Vdash \mathrm{V} : \Sigma$.

1. If $\tau = \forall[\overline{\alpha}].\tau_1 \Rightarrow \tau_2$, then v is of the form $\mathtt{fold}_\beta$ or $\mathtt{unfold}_\beta$.
2. If $\tau = \alpha$ or $\tau = \alpha(\overline{\mathrm{A}})$, where $\alpha : \mathrm{K} \approx \mathrm{A}' \in \Delta$, then v is of the form $\mathtt{fold}_\beta[\overline{\mathrm{B}}](v')$.
3. If $\Sigma = [\![= \mathrm{A} : \mathrm{K}]\!]$, then V is of the form $[\mathrm{B}]$.
4. If $\Sigma = [\![\tau']\!]$, then V is of the form $[v']$.
5. If $\Sigma = [\![\overline{\ell : \Sigma'}]\!]$, then V is of the form $[\overline{\ell = \mathrm{V}'}]$.

Theorem B.5 (Progress)
If $\vdash \Omega\ \ \mathsf{ok}$, then Ω is not stuck.

Corollary B.6 (Type Soundness)
If $\emptyset; \emptyset \vdash \mathrm{M} : \Sigma$, then for all Ω, $(\emptyset;\ \emptyset;\ \bullet;\ \mathrm{M}) \rightarrowtail^* \Omega$ implies that Ω is not stuck.

C. EL Soundness

Elaboration contexts Γ (as defined in Figure 9) extend IL contexts with two new bindings, so we must extend the context well-formedness judgment $\Delta \vdash \Gamma\ \ \mathsf{ok}$ accordingly. If Γ is an elaboration context, then $\Delta \vdash \Gamma\ \ \mathsf{ok}$ iff (1) $\vdash \Delta\ \ \mathsf{ok}$, (2) all IL bindings in Γ are well-formed according to the definition in Appendix A,

$$
\begin{array}{lll}
\text{Core Values} & v ::= & \mathtt{fold}_\alpha \mid \mathtt{unfold}_\alpha \mid \mathtt{fold}_\alpha[\overline{\mathrm{A}}](v) \\
\text{Module Values} & \mathrm{V} ::= & [\mathrm{A}] \mid [v] \mid [\overline{\ell = \mathrm{V}}] \\
\text{Machine States} & \Omega ::= & (\Delta;\, \omega;\, \mathbb{C};\, e) \mid (\Delta;\, \omega;\, \mathbb{C};\, \mathrm{M}) \mid \mathsf{BlackHole} \\
\text{Machine Stores} & \omega ::= & \emptyset \mid \omega, \mathrm{X} \mapsto \mathrm{V} \mid \omega, \mathrm{X} \mapsto \mathbf{?} \\
\text{Continuations} & \mathbb{C} ::= & \bullet \mid \mathbb{C} \circ \mathbb{F} \\
\text{Continuation Frames} & \mathbb{F} ::= & \bullet[\overline{\mathrm{A}}](e) \mid v[\overline{\mathrm{A}}](\bullet) \mid \mathtt{Val}(\bullet) \mid \\
 & & [\bullet] \mid [\overline{\ell_1 = \mathrm{V}_1}, \ell \triangleright \mathrm{X} = \bullet, \overline{\ell_2 \triangleright \mathrm{X}_2 = \mathrm{M}_2}] \mid \bullet.\ell \mid \mathcal{F}[\overline{\mathrm{A}}](\bullet)[\overline{\alpha}] \mid \mathtt{rec}\,(\mathrm{X} : \Sigma)\,\bullet
\end{array}
$$

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

$$
\frac{e = e_1[\overline{\mathrm{A}}](e_2) \quad e \text{ not a value}}{(\Delta;\, \omega;\, \mathbb{C};\, e) \rightarrowtail (\Delta;\, \omega;\, \mathbb{C} \circ \bullet[\overline{\mathrm{A}}](e_2);\, e_1)} \qquad \frac{}{(\Delta;\, \omega;\, \mathbb{C} \circ \bullet[\overline{\mathrm{A}}](e);\, v) \rightarrowtail (\Delta;\, \omega;\, \mathbb{C} \circ v[\overline{\mathrm{A}}](\bullet);\, e)}
$$

$$
\frac{}{(\Delta;\, \omega;\, \mathbb{C} \circ \mathtt{fold}_\alpha[\overline{\mathrm{A}}](\bullet);\, v) \rightarrowtail (\Delta;\, \omega;\, \mathbb{C};\, \mathtt{fold}_\alpha[\overline{\mathrm{A}}](v))} \qquad \frac{}{(\Delta;\, \omega;\, \mathbb{C} \circ \mathtt{unfold}_\alpha[\overline{\mathrm{A}}](\bullet);\, \mathtt{fold}_\beta[\overline{\mathrm{B}}](v))) \rightarrowtail (\Delta;\, \omega;\, \mathbb{C};\, v)}
$$

$$
\frac{}{(\Delta;\, \omega;\, \mathbb{C};\, \mathtt{Val}(\mathrm{M})) \rightarrowtail (\Delta;\, \omega;\, \mathbb{C} \circ \mathtt{Val}(\bullet);\, \mathrm{M})} \qquad \frac{}{(\Delta;\, \omega;\, \mathbb{C} \circ \mathtt{Val}(\bullet);\, [v]) \rightarrowtail (\Delta;\, \omega;\, \mathbb{C};\, v)}
$$

$$
\frac{\mathrm{X} \mapsto \mathrm{V} \in \omega}{(\Delta;\, \omega;\, \mathbb{C};\, \mathrm{X}) \rightarrowtail (\Delta;\, \omega;\, \mathbb{C};\, \mathrm{V})} \qquad \frac{\mathrm{X} \mapsto \mathbf{?} \in \omega}{(\Delta;\, \omega;\, \mathbb{C};\, \mathrm{X}) \rightarrowtail \mathsf{BlackHole}} \qquad \frac{e \text{ not a value}}{(\Delta;\, \omega;\, \mathbb{C};\, [e]) \rightarrowtail (\Delta;\, \omega;\, \mathbb{C} \circ [\bullet];\, e)}
$$

$$
\frac{}{(\Delta;\, \omega;\, \mathbb{C} \circ [\bullet];\, v) \rightarrowtail (\Delta;\, \omega;\, \mathbb{C};\, [v])} \qquad \frac{\mathrm{M} = [\ell_1 \triangleright \mathrm{X}_1 = \mathrm{M}_1, \overline{\ell_2 \triangleright \mathrm{X}_2 = \mathrm{M}_2}] \quad \mathrm{M} \text{ not a value}}{(\Delta;\, \omega;\, \mathbb{C};\, \mathrm{M}) \rightarrowtail (\Delta;\, \omega;\, \mathbb{C} \circ [\ell_1 \triangleright \mathrm{X}_1 = \bullet, \overline{\ell_2 \triangleright \mathrm{X}_2 = \mathrm{M}_2}];\, \mathrm{M}_1)}
$$

$$
\frac{}{\begin{array}{l}(\Delta;\, \omega;\, \mathbb{C} \circ [\overline{\ell_1 = \mathrm{V}_1}, \ell_2 \triangleright \mathrm{X}_2 = \bullet, \ell_3 \triangleright \mathrm{X}_3 = \mathrm{M}_3, \overline{\ell_4 \triangleright \mathrm{X}_4 = \mathrm{M}_4}];\, \mathrm{V}_2) \rightarrowtail \\ (\Delta;\, \omega;\, \mathbb{C} \circ [\overline{\ell_1 = \mathrm{V}_1}, \ell_2 = \mathrm{V}_2, \ell_3 \triangleright \mathrm{X}_3 = \bullet, \overline{\ell_4 \triangleright \mathrm{X}_4 = \{\mathrm{X}_2 \mapsto \mathrm{V}_2\}\mathrm{M}_4}];\, \{\mathrm{X}_2 \mapsto \mathrm{V}_2\}\mathrm{M}_3)\end{array}}
$$

$$
\frac{}{(\Delta;\, \omega;\, \mathbb{C} \circ [\overline{\ell_1 = \mathrm{V}_1}, \ell_2 = \bullet];\, \mathrm{V}_2) \rightarrowtail (\Delta;\, \omega;\, \mathbb{C};\, [\overline{\ell_1 = \mathrm{V}_1}, \ell_2 = \mathrm{V}_2])} \qquad \frac{}{(\Delta;\, \omega;\, \mathbb{C};\, \mathrm{M}.\ell) \rightarrowtail (\Delta;\, \omega;\, \mathbb{C} \circ \bullet.\ell;\, \mathrm{M})}
$$

$$
\frac{\mathrm{V} = [\ldots, \ell = \mathrm{V}_\ell, \ldots]}{(\Delta;\, \omega;\, \mathbb{C} \circ \bullet.\ell;\, \mathrm{V}) \rightarrowtail (\Delta;\, \omega;\, \mathbb{C};\, \mathrm{V}_\ell)} \qquad \frac{}{(\Delta;\, \omega;\, \mathbb{C};\, \mathcal{F}[\overline{\mathrm{A}}](\mathrm{M})[\overline{\alpha}]) \rightarrowtail (\Delta;\, \omega;\, \mathbb{C} \circ \mathcal{F}[\overline{\mathrm{A}}](\bullet)[\overline{\alpha}];\, \mathrm{M})}
$$

$$
\frac{\mathcal{F} = \Lambda(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\lambda(\mathrm{X} : \Sigma_1).\Lambda(\overline{\alpha_2 \uparrow \mathrm{K}_2}).\mathrm{M}}{(\Delta;\, \omega;\, \mathbb{C} \circ \mathcal{F}[\overline{\mathrm{A}}](\bullet)[\overline{\alpha}];\, \mathrm{V}) \rightarrowtail (\Delta;\, \omega;\, \mathbb{C};\, \{\overline{\alpha_1 \mapsto \mathrm{A}}\}\{\mathrm{X} \mapsto \mathrm{V}\}\{\overline{\alpha_2 \mapsto \alpha}\}\mathrm{M})}
$$

$$
\frac{\mathrm{X} \notin \mathrm{dom}(\omega)}{(\Delta;\, \omega;\, \mathbb{C};\, \mathtt{rec}\,(\mathrm{X} : \Sigma)\,\mathrm{M}) \rightarrowtail (\Delta;\, \omega, \mathrm{X} \mapsto \mathbf{?};\, \mathbb{C} \circ \mathtt{rec}\,(\mathrm{X} : \Sigma)\,\bullet;\, \mathrm{M})} \qquad \frac{\mathrm{X} \in \mathrm{dom}(\omega)}{(\Delta;\, \omega;\, \mathbb{C} \circ \mathtt{rec}\,(\mathrm{X} : \Sigma)\,\bullet;\, \mathrm{V}) \rightarrowtail (\Delta;\, \omega \mathbin{@} \mathrm{X} := \mathrm{V};\, \mathbb{C};\, \mathrm{V})}
$$

$$
\frac{}{(\Delta;\, \omega;\, \mathbb{C};\, \mathtt{let}\ \mathrm{F} = \mathcal{F}\ \mathtt{in}\ \mathrm{M}) \rightarrowtail (\Delta;\, \omega;\, \mathbb{C};\, \{\mathrm{F} \mapsto \mathcal{F}\}\mathrm{M})} \qquad \frac{\alpha \notin \mathrm{dom}(\Delta)}{(\Delta;\, \omega;\, \mathbb{C};\, \mathtt{new}\ \alpha \uparrow \mathrm{K}\ \mathtt{in}\ \mathrm{M}) \rightarrowtail (\Delta, \alpha \uparrow \mathrm{K};\, \omega;\, \mathbb{C};\, \mathrm{M})}
$$

$$
\frac{\alpha \in \uparrow(\Delta)}{(\Delta;\, \omega;\, \mathbb{C};\, \mathtt{set}\ \alpha := \mathrm{A}\ \mathtt{in}\ \mathrm{M} : \Sigma) \rightarrowtail (\Delta \mathbin{@} \alpha := \mathrm{A};\, \omega;\, \mathbb{C};\, \mathrm{M})} \qquad \frac{\alpha \in \uparrow(\Delta)}{(\Delta;\, \omega;\, \mathbb{C};\, \mathtt{set}\ \alpha :\approx \mathrm{A}\ \mathtt{in}\ \mathrm{M} : \Sigma) \rightarrowtail (\Delta \mathbin{@} \alpha :\approx \mathrm{A};\, \omega;\, \mathbb{C};\, \mathrm{M})}
$$

Figure 12. IL Dynamic Semantics

(3) for all signature bindings $\mathrm{S} = \mathcal{S}$ in Γ, $\Delta \vdash \mathcal{S}$ ok, and (4) for all functor bindings $\mathrm{F} : \forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).(\mathcal{L}; \Sigma_1) \to \exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).\Sigma_2$ in Γ, we have $\Delta \vdash \forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \to \exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).\Sigma_2$ sig and $\Delta \vdash \exists(\overline{\alpha_1 \uparrow \mathrm{K}_1}).(\mathcal{L}; \Sigma_1)$ ok.

The inference rules defining elaboration (Section 4.2) implicitly erase elaboration contexts into IL contexts whenever they refer to IL judgments in their premises. The erasure is defined in the obvious way: all signature definitions are dropped, and all functor bindings have their abstract type locators $\mathcal{L}$ erased. It is easy to see that valid elaboration contexts erase to valid IL contexts.

Proposition C.1 (Correctness of Equation Solver)
Suppose $\vdash \Delta$ ok and $\Delta \vdash$ solve $\mathcal{L}$ by $\Sigma \rightsquigarrow \delta$ and $\Delta \vdash \Sigma$ sig. Then:

1. $\{\overline{\alpha}\} = \mathrm{dom}(\delta) = \mathrm{dom}(\mathcal{L}) \subseteq \mathrm{dom}(\Delta)$.
2. $\forall \alpha \in \{\overline{\alpha}\}.\ \Delta \vdash \delta\alpha : \Delta(\alpha)$.
3. $\forall \alpha \in \{\overline{\alpha}\}.\ (\mathrm{FV}(\delta\alpha) \cup \mathsf{basis}_\Delta(\delta\alpha)) \cap \{\overline{\alpha}\} = \emptyset$.
4. If $\{\overline{\alpha}\} \subseteq \uparrow(\Delta)$, then $\vdash \Delta \mathbin{@} \overline{\alpha := \delta\alpha}$ ok and $\Delta \mathbin{@} \overline{\alpha := \delta\alpha} \vdash (\mathcal{L}; \Sigma)$ ok.

Regarding Proposition C.1: Part 1 says that the output substitution δ solves precisely for the abstract types in the domain of $\mathcal{L}$. Part 2 says that the resulting solutions are well-formed in Δ. Part 3 says that the solutions in δ do not refer to or depend on any of the $\overline{\alpha}$ in $\mathrm{dom}(\mathcal{L})$. In other words, the judgment returns a *non-recursive* solution. Part 4 says that if the $\overline{\alpha}$ are undefined in Δ, then patching Δ with the solutions for $\overline{\alpha}$ found in δ results in a valid context, and one in which the definitions of $\overline{\alpha}$ match up with their definitions in Σ (as located by $\mathcal{L}$). The proof of Proposition C.1 is completely straightforward.

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

$$\frac{}{\vdash \mathsf{BlackHole\ ok}} \qquad \frac{\Delta \vdash \omega : \Gamma \quad \Delta;\Gamma \vdash e : \tau \quad \Delta;\Gamma \vdash \mathbb{C} : \tau\ \mathsf{cont}}{\vdash (\Delta;\ \omega;\ \mathbb{C};\ e)\ \mathsf{ok}} \qquad \frac{\Delta \vdash \omega : \Gamma \quad \Delta;\Gamma \vdash \mathrm{M} : \Sigma\ \mathsf{with}\ \rho\downarrow \quad \Delta \mathbin{@} \rho\downarrow;\Gamma \vdash \mathbb{C} : \Sigma\ \mathsf{cont}}{\vdash (\Delta;\ \omega;\ \mathbb{C};\ \mathrm{M})\ \mathsf{ok}}$$

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

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

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

$$\frac{\Delta \vdash \tau : \mathbf{T}/\Delta \vdash \Sigma\ \mathsf{sig}}{\Delta;\Gamma \vdash \bullet : \tau/\Sigma\ \mathsf{cont}} \qquad \frac{\Delta;\Gamma \vdash \mathbb{F} : \tau_1/\Sigma_1 \rightsquigarrow \tau_2/\Sigma_2\ \mathsf{with}\ \rho\downarrow \quad \Delta \mathbin{@} \rho\downarrow;\Gamma \vdash \mathbb{C} : \tau_2/\Sigma_2\ \mathsf{cont}}{\Delta;\Gamma \vdash \mathbb{C} \circ \mathbb{F} : \tau_1/\Sigma_1\ \mathsf{cont}}$$

$$\frac{\Delta;\Gamma \vdash \mathbb{C} : \tau'/\Sigma'\ \mathsf{cont} \quad \Delta \vdash \tau' \equiv \tau : \mathbf{T}/\Delta \vdash \Sigma' \equiv \Sigma}{\Delta;\Gamma \vdash \mathbb{C} : \tau/\Sigma\ \mathsf{cont}}$$

Well-formed continuation frames: $\Delta;\Gamma \vdash \mathbb{F} : \tau_1/\Sigma_1 \rightsquigarrow \tau_2/\Sigma_2\ \mathsf{with}\ \rho\downarrow$

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

$$\frac{\Delta \vdash \tau : \mathbf{T}}{\Delta;\Gamma \vdash \mathtt{Val}(\bullet) : [\![\tau]\!] \rightsquigarrow \tau} \qquad \frac{\Delta \vdash \tau : \mathbf{T}}{\Delta;\Gamma \vdash [\bullet] : \tau \rightsquigarrow [\![\tau]\!]}$$

$$\frac{\overline{\Delta;\Gamma \vdash \mathrm{V}_1 : \Sigma_1} \quad \Delta \vdash \Sigma\ \mathsf{sig} \quad \Delta;\Gamma, \mathrm{X} : \Sigma \vdash \overline{[\ell_2 \triangleright \mathrm{X}_2 = \mathrm{M}_2]} : [\![\overline{\ell_2 : \Sigma_2}]\!]\ \mathsf{with}\ \rho\downarrow}{\Delta;\Gamma \vdash [\overline{\ell_1 = \mathrm{V}_1}, \ell \triangleright \mathrm{X} = \bullet, \overline{\ell_2 \triangleright \mathrm{X}_2 = \mathrm{M}_2}] : \Sigma \rightsquigarrow [\![\overline{\ell_1 : \Sigma_1}, \ell : \Sigma, \overline{\ell_2 : \Sigma_2}]\!]\ \mathsf{with}\ \rho\downarrow} \qquad \frac{\Delta \vdash \Sigma\ \mathsf{sig} \quad \Sigma = [\![\ldots, \ell : \Sigma_\ell, \ldots]\!]}{\Delta;\Gamma \vdash \bullet.\ell : \Sigma \rightsquigarrow \Sigma_\ell}$$

$$\frac{\Delta;\Gamma \vdash \mathcal{F} : \forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \to \exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).\Sigma_2 \quad \overline{\Delta \vdash \mathrm{A} : \mathrm{K}_1} \quad \overline{\alpha \uparrow \mathrm{K}_2} \subseteq \Delta}{\Delta;\Gamma \vdash \mathcal{F}[\overline{\mathrm{A}}](\bullet)[\overline{\alpha}] : \{\overline{\alpha_1 \mapsto \mathrm{A}}\}\Sigma_1 \rightsquigarrow \{\overline{\alpha_1 \mapsto \mathrm{A}}\}\{\overline{\alpha_2 \mapsto \alpha}\}\Sigma_2\ \mathsf{with}\ \overline{\alpha}\downarrow} \qquad \frac{\Delta;\Gamma \vdash \mathrm{X} : \Sigma}{\Delta;\Gamma \vdash \mathtt{rec}\ (\mathrm{X} : \Sigma)\ \bullet : \Sigma \rightsquigarrow \Sigma}$$

Figure 13. Typing Judgments for Machine States, Machine Stores, and Continuations

Theorem C.2 (Properties of Elaboration Judgments)
Suppose $\Delta \vdash \Gamma$ ok. Then:

1. If $\Delta;\Gamma \vdash \mathit{con} \rightsquigarrow \mathrm{A} : \mathrm{K}$, then $\Delta \vdash \mathrm{A} : \mathrm{K}$.
2. If $\Delta;\Gamma \vdash \mathit{exp} \rightsquigarrow e : \tau$, then $\Delta;\Gamma \vdash e : \tau$.
3. If $\Delta;\Gamma \vdash \mathit{sig} \rightsquigarrow \mathcal{S}$, then $\Delta \vdash \mathcal{S}$ ok.
4. If $\Delta;\Gamma;(\mathcal{L}_0;\Sigma_0) \vdash \mathit{mod} \rightsquigarrow \exists(\overline{\alpha \uparrow \mathrm{K}}).(\Sigma;\mathit{pemod})$ and $\Delta \vdash (\mathcal{L}_0;\Sigma_0)$ ok, then $\Delta, \overline{\alpha \uparrow \mathrm{K}} \vdash \Sigma$ sig.
5. If $\Delta;\Gamma \vdash \mathit{pemod} \rightsquigarrow \mathrm{M} : \Sigma\ \mathsf{with}\ \rho\downarrow$, then $\Delta;\Gamma \vdash \mathrm{M} : \Sigma\ \mathsf{with}\ \rho\downarrow$.
6. If $\vdash_{\mathrm{can}} (\mathcal{L};\Sigma) \rightsquigarrow \mathrm{M}$ and $\mathrm{dom}(\mathcal{L}) \subseteq \uparrow(\Delta)$ and $\Delta \vdash (\mathcal{L};\Sigma)$ ok, then $\Delta;\Gamma \vdash \mathrm{M} : \Sigma\ \mathsf{with}\ \mathrm{dom}(\mathcal{L})\downarrow$.
7. If $\Delta;\Gamma \vdash \mathrm{P} \preceq \Sigma \rightsquigarrow \mathrm{M}$, then $\Delta;\Gamma \vdash \mathrm{M} : \Sigma$.
8. If $\Delta;\Gamma \vdash \mathit{prog} \rightsquigarrow \mathrm{M}$, then $\Delta;\Gamma \vdash \mathrm{M} : [\![]\!]$.

Proof: By straightforward induction on elaboration. ■

Soundness of the external language follows from Part 8 of Theorem C.2, together with type soundness of the IL.