A Type System for Recursive Modules

Derek Dreyer
Toyota Technological Institute
Chicago, Illinois, USA
dreyer@tti-c.org

Abstract

There has been much work in recent years on extending ML with recursive modules. One of the most difficult problems in the development of such an extension is the *double vision* problem, which concerns the interaction of recursion and data abstraction. In previous work, I defined a type system called RTG, which solves the double vision problem at the level of a System-F-style core calculus. In this paper, I scale the ideas and techniques of RTG to the level of a recursive ML-style module calculus called RMC, thus establishing that no tradeoff between data abstraction and recursive modules is necessary. First, I describe RMC's typing rules for recursive modules informally and discuss some of the design questions that arose in developing them. Then, I present the formal semantics of RMC, which is interesting in its own right. The formalization synthesizes aspects of both the Definition and the Harper-Stone interpretation of Standard ML, and includes a novel two-pass algorithm for recursive module typechecking in which the coherence of the two passes is emphasized by their representation in terms of the same set of inference rules.

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

General Terms Design, Languages, Theory

Keywords Type systems, modules, recursion, abstract data types

1. Introduction

The ML module system (MacQueen 1984), while esteemed for its strong support for data abstraction and code reuse, 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 (Russo 2001; Leroy 2003; Dreyer 2005; Nakata and Garrigue 2006).

My Ph.D. thesis (Dreyer 2005) examines 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. I 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. (A motivating example of the problem is given in Section 2.1.)

Double vision has proven difficult to cure. To the extent that existing recursive module proposals address the problem, they do so either by imposing severe restrictions on the use of data abstraction within recursive module definitions (Crary et al. 1999; Russo 2001), or else by implementing tricky typechecking maneuvers that are difficult to formalize cleanly and only work in certain cases (Leroy 2003; Dreyer 2005). Neither of these approaches is satisfactory. (An overview of existing proposals is given in Section 2.2.)

In recent work (Dreyer 2007b), I argued that the reason double vision is a difficult problem is that the classical type-theoretic interpretation of abstract data types—namely, as packages of existential type (Mitchell and Plotkin 1988)—is inadequate for expressing the kinds of recursive abstract data types that arise naturally in the context of recursive modules. I defined a type system called RTG (for Recursive Type Generativity) that addresses this deficiency at the level of a System-F-style core calculus. (The basic idea of RTG is presented in Section 2.3.) Although I gave several examples to suggest how recursive modules may be encoded in RTG, I left the development of a general recursive module semantics to future work.

The primary contribution of this paper is to fulfill the promise of RTG by scaling its ideas and techniques to the level of a recursive ML-style module calculus, which I call RMC. The semantics of RMC successfully avoids the double vision problem without placing any undue restrictions on the use of data abstraction in recursive modules. While RMC's approach to solving double vision is based closely on that of RTG, the typechecking of RMC programs is more complex than that of RTG programs (much as the typechecking of ML modules is more complex than that of System F). In Section 3, I describe recursive module typechecking at an informal level, and provide a number of examples and exercises to demonstrate its subtleties. I also discuss a number of the technical issues that arose (aside from the handling of double vision) in working out a semantics for a general recursive module language.

The formalization of the RMC type system (presented in full detail in Section 4) is interesting on several levels. First, it exhibits a hybrid of the two main approaches to defining the semantics of ML-style modules: namely, the Definition of Standard ML (Milner et al. 1997) and the type-theoretic interpretation of Harper and

ICFP'07, October 1–3, 2007, Freiburg, Germany.
Copyright © 2007 ACM 978-1-59593-815-2/07/0010. . . $5.00

Stone (2000). On one hand, I define the dynamic semantics of RMC by means of a Harper-Stone-style *elaboration* relation (aka *evidence translation*) into an "internal" type system. This internal type system is just RTG extended with a primitive module system. The main benefit of this approach is that it enables us to establish the type soundness of RMC as a corollary of the type soundness of RTG. In contrast, while Definition-style formalisms typically employ a "direct" big-step evaluation semantics, type soundness of these formalisms is more difficult to prove (Tofte 1988).

On the other hand, the interpretations of modules and signatures in RMC are much closer in detail to the *semantic objects* of the Definition than to the *translucent-sums/manifest-types* formalism (Harper and Lillibridge 1994; Leroy 1994) employed by Harper and Stone. Moreover, like the Definition's typing rules, the RMC typing rules are completely self-contained and can be explained to the programmer independently of the evidence translation into the internal RTG type system. In fact, that is precisely how I will present the rules in this paper, leaving most of the details of evidence translation to the companion technical report (Dreyer 2007a). Thus, RMC's formalization combines the benefits of both definitional approaches.

A second interesting feature of RMC's static semantics is its streamlined presentation of recursive module typechecking. The proper handling of double vision seems to demand the use of a two-pass algorithm for typechecking certain kinds of modules—the first ("static") pass computes only the type components of the module, while the second pass typechecks the full module. As a way of demonstrating the semantic coherence of these passes, I formalize both of them using a single set of typing rules—the only difference is that the static pass omits some of the premises of the rules for the full pass. I believe the built-in coherence of these judgments makes the semantics of RMC easier to understand than other recursive module formalisms that involve multiple typechecking passes (Nakata and Garrigue 2006; Dreyer 2005, 2006).

A key feature that RMC does not account for in its present form is the ability to compile mutually recursive modules separately and link them dynamically. None of the related work on recursive ML-style modules supports this feature either. In prior work (Dreyer 2007b), I demonstrated that the RTG calculus is capable of encoding separately compiled recursive modules, so I believe it will be possible in the future to scale RMC to support separate compilation. At the moment, however, it is unclear how best to introduce this feature syntactically into a programmable module language, and I consider it separable from the focus of the present work.

Detailed comparisons with related work on recursive modules appear throughout the paper, particularly in Sections 2.2 and 3. I conclude in Section 5 with further discussion of related work, as well as directions for future work.

2. The Double Vision Problem

Crary et al. (1999) were the first to attempt to establish a type-theoretic foundation for recursive ML-style modules. Perhaps the most influential aspect of their work is that they set forth the two main syntactic extensions to the ML module system that appear (with minor variations) in every subsequent recursive module proposal, including the present one.

The first is the recursive module construct itself, which has the form `rec` $(\mathrm{X}{:}\,\mathit{sig})\ \mathit{mod}$. Here, mod is the module being recursively defined, X is the module identifier by which mod refers to itself, and sig is the *forward declaration* signature, which is used as the signature of X during the typechecking of mod. Mutually recursive modules are definable as a single recursive module with multiple substructures.

The second extension is the *recursively dependent signature* (or *rds*), which has the form `rec` $(\mathrm{X})\ \mathit{sig}$. The idea is that X is a variable by which sig can refer recursively to the *module* whose signature sig is intended to describe. This functionality is critical if we wish to describe the signatures of mutually recursive modules with abstract type components, such as those in the motivating example (below). Although several authors refer to rds's as "recursive signatures", I concur with Crary et al. that this is misleading, as it gives one the impression that the signatures can refer recursively to themselves (rather than to the modules that inhabit them).

```
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. Motivating Recursive Module Example

2.1 Motivating Example

Figure 1 presents a motivating example of a recursive module that exhibits the double vision problem. So that this motivating example may serve as a running example throughout the paper, it is concise to the point of being contrived. For more realistically detailed examples of recursive modules, see Dreyer (2005) and Nakata and Garrigue (2006).

The recursive module in Figure 1 comprises two mutually recursive substructures `A` and `B`, with `A` providing an abstract type component `t` and a value component `f`, and `B` providing an abstract 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 may write down the signature for each module independently and bind it to a signature identifier ($\mathtt{S}_\mathtt{A}$ and $\mathtt{S}_\mathtt{B}$), 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 (2005) recently dubbed *fibration*.

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. 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. While typechecking the body of the definition, we assume that the recursive variable X has the forward declaration signature S. 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. To the programmer, however, this may seem bizarre, since X.A.t is merely a recursive reference to t, so the two types should be indiscernible, shouldn't they? 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.

Intuitively, the problem here is obvious. The bodies of A and B should have access to different privileged information about the type components of X. Specifically, A should know that X.A.t is int, but it ought not know anything about X.B.u. Conversely, B should know that X.B.u is bool, but it ought not know anything about X.A.t. However, it is far from obvious how to define a general typing rule for recursive modules so that X has different signatures when typechecking different parts of the recursive module definition.

2.2 Existing Approaches to Double Vision

Under all of the existing recursive module proposals, the program in Figure 1 is rejected as ill-typed.

Crary et al. (1999) observe the double vision problem in their original paper, although they do not refer to it as such. (Instead, they call it the "trouble with opacity".) Their response to the problem is simply to restrict the forward declaration signature of a recursive module to be transparent. In the case of our motivating example, this means that the definitions of A.t and B.u would need to be exposed, effectively prohibiting either module from hiding its internal data representation from the other. Aware of this, Crary et al. discuss informally several ideas for how this restriction might be lifted in practice, but to my knowledge none of these ideas has been formally fleshed out.

Russo (2001) defines a recursive module extension to Standard ML, which he has implemented in the Moscow ML compiler. 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 $(\mathtt{X} : sig)\ mod$ demands that, if a type component t is forward-declared abstractly in sig, then mod must define t to be equal to X.t (*i.e.,* by writing type t = X.t). While this clearly has the effect of avoiding double vision, it also means that t never gets defined anywhere!

This restriction makes it essentially impossible to forward-declare abstract data types. It is worth noting that Russo makes an exception for types that are defined by an algebraic datatype definition. If a type is forward-declared using a datatype specification, then the body of the recursive module must define the corresponding type via SML's datatype copying construct, *e.g.,* datatype t = datatype X.t. In some sense, though, this is the exception that proves the rule—while t in this case is technically an abstract type, the datatype specification of t in the forward declaration signature exposes all of t's data constructors, so t's internal representation is all but transparent.

Leroy (2003) describes informally a recursive module extension that he implemented for OCaml. To permit abstract type specifications in forward declaration signatures, he sketches a typechecking algorithm that typechecks different mutually recursive modules under different typing contexts. However, while his algorithm successfully avoids double vision in certain cases, it only works for type components that are defined internally by datatype definitions. It does not work for types that are defined internally by transparent bindings (such as A.t and B.u in our motivating example) or for types that are defined under more than one level of opaque signature ascription. Moreover, there remains no formal account of his algorithm.

Nakata and Garrigue (2006) 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. Nevertheless, as the authors freely admit, their approach still suffers from the double vision problem. (In fact, as I explain in Section 3.4, some of their examples only typecheck *because* their type system suffers from the double vision problem.) The authors mention the existence of a workaround by which the programmer may manually coerce values from one "double vision" of a type component to the other (*e.g.,* from X.A.t to int), but they do not describe this workaround in any detail.

In my thesis (Dreyer 2005), I formally defined a recursive module extension to ML, which I implemented in the TILT compiler. My typechecking algorithm was an attempt to generalize the approach taken by Leroy's OCaml extension into a more complete solution to the double vision problem. Figure 1 does not typecheck under my TILT extension, but only because my semantics for recursively dependent signatures was overly restrictive and did not permit one to write the signature S in its fibered form. A slight variant of this example—using an unfibered forward declaration signature—*does* typecheck in TILT.

Nevertheless, since my thesis does not contain a clean type-theoretic account of the double vision problem, the formalization of my TILT extension is extremely long and complex. It employs a variety of *ad hoc* tricks, such as "meta-signatures" containing both "public" and "private" interfaces for subcomponents, and inference rules that make critical use of graphical boxes drawn around chunks of the typing context. In short, while my TILT extension successfully avoids the double vision problem (as far as I know), its formalization is incomprehensible. The desire for a simpler solution was the primary motivation for my work on the RTG type system.

2.3 The RTG Type System

In traditional accounts of data abstraction, including both existential types (Mitchell and Plotkin 1988) and ML-style module systems (MacQueen 1984), 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. For instance, in the case of our motivating example, double vision arises because module A wants to define an abstract type t in a scope where a name for that type—X.A.t—already exists. The key idea of my RTG type system (Dreyer 2007b) is to separate the generation of the name for an abstract type from the definition of the type, so that a 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 2, which demonstrates how the motivating example from Figure 1 would be encoded in RTG. (Actually, the encoding is in a variant of RTG that includes a primitive module system. This variant is defined formally in the companion technical report (Dreyer 2007a).)

Here, Σ is the RTG representation of the forward declaration signature S from Figure 1. The key difference is that the type components A.t and B.u of Σ are not abstract like those of S—rather, their specifications, written as $[\![= \alpha : \mathbf{T}]\!]$ and $[\![= \beta : \mathbf{T}]\!]$, denote that they are transparently equal to the free type variables α and β. These type variables are created and bound, before the recursive module AB is defined, by invoking a new construct. Since this new

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

where

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

Figure 2. Encoding of Figure 1 in a Variant of RTG

construct does not actually supply a definition for α and β, they are considered *undefined*, and are marked as such in the type context during typechecking using an ↑ binding.

In the recursive module body, the uses of *opaque signature ascription* (aka *sealing*) have been replaced by RTG's own sealing construct, called the `def` construct.[1] For `A`, what this `def` 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 `def`. 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.

In short, RTG provides a simple way of typechecking different parts of the recursive module definition under typing contexts that expose different privileged information concerning the identities of `X.A.t` and `X.B.u`. This corresponds to the programmer's natural intuition about how a recursive module should be typechecked.

3. An Informal Overview of RMC Typechecking

In this section, I explain what is involved in scaling the ideas and techniques of the RTG type system from the level of a System-F-style core calculus to the level of an ML-style module system. I begin in Section 3.1 with a simple informal explanation of how recursive modules and sealed modules (the two main language features on display in our running example) are typechecked. Then, in the following sections, I explore some of the interesting design questions that arise when one goes to work out the details.

Along the way, I test the reader's understanding of the exposition by offering some simple ***exercises*** concerning RMC semantics. I ***strongly encourage*** the reader to attempt the exercises—or, failing that, to cheat by looking ahead to the ***detailed solutions*** given at the end of the paper—because the exercises (and their solutions) help to illuminate a number of subtle aspects of RMC semantics.

3.1 Typechecking Recursive and Sealed Modules

Intuitively, the basic goal of the RMC type system is to typecheck an RMC program such as the example in Figure 1 as if it were the RTG program in Figure 2. To understand how this is achieved, let us consider the major ways in which Figure 2 differs from Figure 1.

First of all, the success of RTG is based on its built-in functionality for forward declaration of abstract types. Concretely, the typechecking of an RTG expression is done under the assumption that the names of the abstract types the expression wants to define have already been created and are bound as undefined (↑) in the typing context. The RMC type system relies on a similar functionality. In the case of our motivating example, we will assume that when we are typechecking the module `AB`, there exist two type variables α and β, which will represent `AB.A.t` and `AB.B.u`, respectively, and which are bound in the initial context of `AB` as undefined.[2] The names of these type variables are arbitrary—they are purely *semantic* representations of `AB`'s abstract types, *i.e.*, they play an important role in typechecking but are not visible to the RMC programmer syntactically.

Second, the forward declaration signature `S` in Figure 1 is opaque, whereas the forward declaration Σ in Figure 2 is transparent. To transform `S` into Σ, we need to reify the specifications of `S`'s opaque type components to be transparently equal to their definitions in the recursive module body. Under ordinary module type systems this would be difficult, since the recursive module body can (and, in the case of module `AB`, does) withhold the definitions of those components by defining them abstractly. However, under the RMC type system this is not an issue, since we have arranged for those components to be named ahead of time. For `AB`, the definitions of `A.t` and `B.u` are simply α and β (the undefined type variables bound in the context). Correspondingly, Σ is morally equivalent[3] to `S` filled in with those definitions, *i.e.*, `S where type A.t =` α `and type B.u =` β.

* * *

Exercise #1: Suppose that our example were modified so that the definition of module `B` were *not* sealed. What transparent signature would `S` be transformed to in that case, and consequently what effect would this have on the typechecking of the recursive module?

* * *

Third, sealed modules (such as `AB.A` and `AB.B`) in Figure 1 become `def` expressions in Figure 2. In order to typecheck a sealed module as if it were a `def` expression, we first compute a type substitution δ, mapping the names of the abstract types that the module will define to their definitions in the module body (underneath the sealing). For example, for `AB.A`, the computed δ would map α to `int`, and for `AB.B`, the computed δ would map β to `bool`. This substitution is then applied to the typing context before typechecking the module body. Thus, when typechecking the body of `AB.A`, all references to α in the signature of `X` are replaced by `int`; when typechecking the body of `AB.B`, all references to β in the signature of `X` are replaced by `bool`. In this way, each module's secret knowledge of certain type components is reflected in the context under which it is typechecked.

To summarize, and also to number the steps of typechecking for convenient reference later in the paper, here is how an RMC recursive module of the form `rec` $(\mathrm{X} : sig)\ mod$ is typechecked:

1. Assuming X has signature sig, compute the type components of mod, and use these to look up the definitions of the type components that are specified opaquely in sig.
2. Use the information from the previous step to construct a transparent forward declaration Σ, which is just sig with its opaque type components reified with their definitions from mod.
3. After rebinding X in the context with Σ, typecheck mod.
4. Check that mod's signature matches (is coercible to) Σ.

[1] In (Dreyer 2007b) this is called the `set` construct.

[2] Arranging for these variables to "magically" appear in the context—in the right number and with the right kinds—is not a problem. (See Section 4.5.)

[3] I say *morally* equivalent because Σ is a *semantic* signature, not a *syntactic* RMC signature, but this is largely a technical point. The distinction between syntactic and semantic signatures is explained in Section 4.

The typechecking of an opaquely sealed module, mod :> sig, proceeds as follows:

1. For each of the type components specified opaquely in sig, there should be an abstract type variable α bound as undefined in the context (with the appropriate kind).
2. Compute the type components of mod, and use these to determine a substitution δ that maps the undefined type variables from the first step to their definitions in mod.
3. After applying δ to the typing context, typecheck mod.
4. Check that mod's signature matches (is coercible to) sig.

* * *

Exercise #2: Suppose that the following is well-typed in RMC:

```
rec (X : sig structure A : sig end) struct
  structure A = mod
end
```

Here, sig is an RMC signature, and mod is an RMC module. Also, assume that neither `X` nor `B` appears free in sig. Determine whether or not the above recursive module would continue to be well-typed if its body (`structure A = mod`) were replaced with:

(a). `structure A :> sig = mod`

(b). `structure B :> sig = mod; structure A = B`

(c). `structure B = mod; structure A :> sig = B`

3.2 Forward Declarations, Not Signature Ascriptions

One design point on which existing recursive module proposals differ is the question of whether the forward declaration signature in a recursive module definition should also serve as the exported signature of the recursive module itself.

In Crary et al.'s foundational type system, recursive modules are modeled as fixed-points at the level of modules. According to this interpretation, rec $(\mathrm{X} : sig)$ mod has signature sig, as long as mod has sig under the assumption that X has sig. Although Crary et al.'s is the only proposal to treat recursive modules explicitly as fixed-points,[4] several other proposals, including Leroy's and the one in my thesis, follow suit in treating the forward declaration sig as the principal exported signature of the recursive module itself. In contrast, Russo's extension to Moscow ML treats sig merely as a forward declaration, not as a sealing signature. In other words, it allows the recursive module to export components that appear in mod but are not forward-declared in sig.

I believe that Russo's approach is a clear win, and RMC adopts it. If the author of a recursive module wishes to seal the body of the recursive module with the forward declaration signature, it is very easy to do so explicitly, *e.g.,* by writing rec $(\mathrm{X} : sig)$ $(mod$:> $sig)$. Moreover, if a recursive module is only "slightly" recursive—*e.g.,* if there is only one value component `f`, say, that needs to be referred to recursively through the recursive module variable X—then Russo's approach only requires one to specify the type of that one component `f` in the forward declaration signature. There is no need to forward-declare components of the module that will not be referred to recursively.

* * *

Exercise #3: In the example in Figure 1, is it necessary to use a recursively dependent signature to define `S`? Is there a simpler signature that could be used as a forward declaration, without effecting any changes to the exported signature of `AB`?

3.3 Computing the Type Components of a Module

A central step in both typing rules described above is the one in which we compute the type components of the underlying module mod. I will refer to this phase as the *static typechecking* of mod. In an earlier version of the RMC system (Dreyer 2006), which I discuss in Section 5.1, this static typechecking pass was formalized using a completely different set of rules from the regular module typing judgment, and I found that this made the language definition seem somewhat *ad hoc* and confusing. (It is a similar problem that plagues the typechecking algorithm in my thesis (Dreyer 2005).)

As a result, I have sought to develop a more *declarative* account of module typechecking. In particular, my initial idea was that the static typechecking steps could consist merely of nondeterministically *guessing* the type components of mod. For example, in typechecking the sealed definition of `AB.A` from Figure 1, one might simply *guess* that the type `t` was defined internally as `int`. That one had guessed *correctly* could be verified after the fact by making sure that the underlying module matched a signature with `t` transparently equal to `int`. In practice, one would of course need to supply a deterministic algorithm in order to perform the guessing, but the declarative definition of the type system would not need to specify the gory details of this algorithm.

Unfortunately, this idea does not seem to work, and it is instructive to see why. First, consider the following example:

```
rec (X : sig type t end)
  struct type t = X.t end
```

We run into trouble here if we try to nondeterministically guess the definition of `t` in the recursive module body. If we guess that `t` is `int`, and we typecheck the body under a context where `X` has the signature `sig type t = int end`—as demanded by step 3 of the recursive module typing rule described above—we find that the body has the same signature. However, if we guess that `t` is `bool`, the module still typechecks, but this time with signature `sig type t = bool end`. Hence, we lose the property that modules have principal signatures.

In short, the problem is that the recursive definition of the type `t` in this example is *non-contractive, i.e.,* there are infinitely many ways to solve it. Ideally, we would like to demand that the definitions of the type components be contractive. But this demand is a catch-22: we cannot even *state* it unless we have already computed the definitions of the type components, which is precisely what we were trying to avoid doing by nondeterministically guessing them.

A seemingly simple fix is to require that when we guess the type components of a module, the guess we make is the *unique* guess that enables typechecking to succeed. Such a restriction would banish the above example from consideration. However, this modified approach causes a different kind of trouble. Consider this second example:

```
rec (X : sig type 'a t end) struct
  type 'a t = 'a X.t
  val n : int t = 3
  val b : bool t = true
end
```

In this example, although the type constructor `t` is defined in the same non-contractive way as before, there is a unique solution for it so that the whole recursive module typechecks. Specifically, the type annotations on `n` and `b` constrain the definition of `t` to be the identity function, $\lambda\alpha.\alpha$. Figuring this out requires a form of higher-order unification, which in general is undecidable (Goldfarb 1981).

These types of examples have led me to abandon the idea of guessing the type components of a module nondeterministically. Fortunately, I have found a way to compute the type components of a module (deterministically) that does not complicate the lan-

[4] Most other proposals employ a Scheme-style backpatching semantics for module-level recursion, as does the present one. One exception is the proposal of Nakata and Garrigue (2006), which uses a call-by-name semantics for modules.

guage definition with a whole set of extra rules. The basic idea is that the inference rules that implement static typechecking are the same as those for normal typechecking, except with some of the premises—such as the ones concerning *value* components of modules—removed. As a result, it is only necessary to write down one set of inference rules for both the regular and static module typechecking judgments. I leave further discussion of this technique until Section 4.4.

In both of the problematic examples presented in this section, the RMC static typechecking judgment would successfully compute the type component `t` in the body of the recursive module to equal `X.t`. Whether such a cyclic type definition is considered acceptable is then a separate question, examined in the next section.

* * *

Exercise #4: Can you come up with a variant of the second problematic example above that achieves the same effect—*i.e.,* there is a unique way of guessing `X.t` correctly, and figuring it out involves higher-order unification—but where the module in your variant *only* has type components, no value components?

3.4 Cyclic Type Definitions

Recursive modules provide a natural means of writing down cyclic type definitions that span module boundaries. However, existing recursive module proposals differ on what kinds of cyclic type definitions they consider permissible.

One approach is to allow *transparent* type cycles, *i.e.,* type components that are defined transparently in terms of themselves, such as `type t = int * X.t`. This is the approach taken by Crary et al. (1999), but it requires them to extend their type theory with support for so-called *equi-recursive* type constructors of higher kind.[5] The meta-theory of higher-kinded equi-recursive type constructors is not well-understood (in particular, it is not known whether type equivalence in their presence is decidable).

A more restrictive approach is the one taken by Leroy (2003) and Nakata and Garrigue (2006), who permit cycles between transparent type definitions, but only if they are intercepted by the use of opaque sealing. For instance, suppose we were to modify the example in Figure 1 so that internally `A.t` were defined to equal `int * X.A.t`. The resulting cyclic definition would be permissible in OCaml and Traviata because the type cycle is broken by the use of opaque sealing in the definition of module `A`.

However, the ability to express such recursive type definitions in these languages is intricately tied to their failure to handle double vision. For example, if these languages were to solve the double vision problem, then in the definition of module `A`, the type `X.A.t` would be seen as transparently equal to `int * X.A.t`. This would constitute an unbroken transparent type cycle, which OCaml and Traviata treat as illegal.

In the interest of adopting a simple policy concerning type cycles that is compatible with solving double vision, my design for RMC follows RTG in requiring that all type cycles go through at least one component that is defined by a `datatype` declaration. That is, even if all uses of opaque sealing are stripped away, there must still be no transparent type cycles. This policy has the advantage that recursive modules do not introduce any new forms of (equi-)recursive type definitions that are not already expressible in the underlying core language of ML—they just provide the ability to decompose ML's existing forms of recursive type definitions into modular components.

That said, one consequence of following RTG is that the typechecking of certain constructs in RMC is somewhat conservative. Specifically, in order to ensure that no transparent type cycles arise in the presence of data abstraction, (1) the internal definitions of abstract types in a sealed module are not allowed to depend on any type variables bound as undefined (↑) in the context, and (2) in functor applications, the type components of the argument module may not depend on any type variables bound as undefined (↑) in the context. (These restrictions are derived directly from similar restrictions in RTG, and I refer the reader to (Dreyer 2007b) for detailed discussion.) Nevertheless, as demonstrated in the RTG paper, this approach is sufficient to typecheck common uses of sealing and functors in recursive modules—*e.g.,* Okasaki's *bootstrapped heap* example (Okasaki 1998). I am currently investigating ways to relax this restriction by generalizing RTG's treatment of type cycles.

Finally, it is important to mention how RMC defines and detects a transparent type cycle. The question arises once we have computed the type components of some module, at which point we typically need to use them in order to look up the definitions of opaque type components in some signature. RMC's policy is that there must be some way of ordering the type components we are looking up so that each component's definition only depends recursively on the previous ones in the ordering. For example, consider:

```
rec (X : sig type t; type 'a u end) struct
  type t = bool X.u
  type 'a u = 'a
end
```

This module is well-typed in RMC because the components of the module can be named in a certain order (`u`, then `t`) so that their definitions are acyclic—`t`'s recursive dependency on `X.u` is OK because `u` comes earlier in the ordering. In contrast, consider:

```
rec (X : sig type t; type 'a u end) struct
  type t = X.t X.u
  type 'a u = 'a
end
```

In this case, the recursive module typing rule rejects the program because the definition of `t` refers cyclically to itself (via `X.t`).

It is worth noting that RMC also rejects similar examples where no "true" transparent cycle exists, such as:

```
rec (X : sig type t; type 'a u end) struct
  type t = X.t X.u
  type 'a u = int
end
```

I do not consider this to be a serious limitation. In the above example, it does not seem like a serious hardship for the programmer to remove this cyclic dependency by replacing the module's definition of `t` with `type t = int`. Alternatively, the programmer could make the type definition `type 'a u = int` explicit in the forward declaration signature, in which case the static typechecking step would be able to determine that the definition of `t` in the body normalizes to `int`.

* * *

Exercise #5: The example in Figure 1 clearly does not have any type cycles. What would happen, however, if we changed it in any of the following ways? Would the RMC type system accept it or reject it?

(a). Change the internal definition of `A.t` to `int * X.B.u`.

(b). Change the internal definition of `B.u` to `bool * X.A.t`.

(c). Change the example as described in **(a)**,
and also remove the sealing in the definition of module `A`.

(d). Change the example as described in both **(b)** and **(c)**.

[5] Crary et al. coined the term *equi-recursive* to describe recursive types whose equational theory follows the style of Amadio and Cardelli (1993).

3.5 Recursively Dependent Signatures

RMC extends the signature language of ML with recursively dependent signatures (rds's), which have the form $\texttt{rec}\,(\mathrm{X})\ sig$. Compared with typechecking a recursive module, checking the well-formedness of an rds is fairly straightforward. The basic goal is to check that the rds does not contain any cyclic transparent type specifications, whose presence would demand support for equi-recursive types.[6] While RMC's treatment of rds's is not markedly different from their treatment in most other proposals, it is worth explaining informally nonetheless.

The well-formedness checking of $\texttt{rec}\,(\mathrm{X})\ sig$ proceeds as follows. First, we need to come up with some temporary signature sig' to which we can bind X during the checking of sig. This temporary sig' will act essentially as a forward declaration of sig. As such, it need only be a "shallow" representation of sig—it should record the (path-)names and kinds of sig's type components, but may ignore sig's value components, because sig can only possibly refer to the *type* components of X. In Russo's account of rds's, the programmer is required to write down this shallow sig' explicitly, but it is perfectly easy to infer sig' via a syntactic pass over sig.

Second, after binding X to sig' in the context, we proceed to check the well-formedness of sig.

Third, we check that there is some linear ordering of the type components of sig such that no transparent component's specification depends on a later component in the ordering. This is formalized in a manner very similar to the detection of transparent type cycles in recursive modules (as described in the previous section).

For example, `rec (X) sig type t = X.t end` will be ill-formed because `t` is defined transparently in terms of itself. In contrast, the signature `S` from Figure 1 will typecheck successfully—even though it contains references to the recursive module variable `X` in the specifications of `A.u` and `B.t`—because the type components of the signature can be ordered in such a way that those references are seen as acyclic (*i.e.,* `A.t`, `B.u`, `A.u`, `B.t`).

4. The RMC Type System

4.1 Syntax

Figure 3 gives the syntax of RMC. While RMC is intended to be representative of a Standard ML-like module language, it does not directly support all features of SML. I focus instead on supporting the most semantically interesting features, and leave formalization of a full-fledged ML extension to future work.

Core Language In the spirit of keeping the core language as underdetermined as possible, the only interesting type- and term-level construct considered here is the *path* P, which is a module variable X followed by zero or more component projections. As a matter of notation, I will write $\mathrm{X}.\ell_1...\ell_n$ as shorthand for $\mathrm{X}.\epsilon.\ell_1...\ell_n$. In particular, X is shorthand for $\mathrm{X}.\epsilon$.

As in ML, type constructors con 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 $\mathbf{T}$, where $n > 0$. For uniformity, in some typing rules $\mathbf{T}^0 \rightarrow \mathbf{T}$ is treated as synonymous with $\mathbf{T}$, and $con()$ as synonymous with con (when con has kind $\mathbf{T}$). I use the overbar syntax to denote a sequence of zero or more objects separated by commas (*e.g.,* $\overline{con} = con_1, \ldots, con_n$).

Signatures In order to simplify and regularize the syntax of modules and signatures, I model type components and value components as *atomic* modules. Corresponding to ML's notion of an opaque type specification, $[\![\mathrm{K}]\!]$ denotes the signature of an atomic module containing a single type component of kind K. While there is no primitive signature corresponding to a transparent type specification, RMC does support ML's `where type` (or `with type`) construct (abbreviated here as `where`), and Figure 3 shows how to define the transparent type signature $[\![= con : \mathrm{K}]\!]$ as a derived form. (This is how the Definition of SML defines transparent type specifications as well (Milner et al. 1997).)

$$
\begin{array}{lrl}
\text{Type Variables} & \alpha, \beta & \\
\text{Module Var's} & \mathrm{X}, \mathrm{Y} & \\
\text{Labels} & \ell & \\
\text{Label Sequences} & \ell s ::= & \epsilon \mid \ell s.\ell \\
\text{Paths} & \mathrm{P} ::= & \mathrm{X}.\ell s \\
\text{Kinds} & \mathrm{K}, \mathrm{L} ::= & \mathbf{T} \mid \mathbf{T}^n \rightarrow \mathbf{T}\ \ (n > 0) \\
\text{Type Constr's} & con ::= & \mathrm{P} \mid \alpha \mid \lambda(\overline{\alpha}).con \mid con(\overline{con}) \mid \ldots \\
\text{Terms} & exp ::= & \mathrm{P} \mid \ldots \\
 & & \\
\text{Signatures} & sig ::= & [\![\mathrm{K}]\!] \mid [\![con]\!] \mid [\![\ell \approx con : \mathrm{K}]\!] \mid \\
 & & [\![\overline{\ell \triangleright \mathrm{X} : sig}]\!] \mid (\mathrm{X} : sig_1) \rightarrow sig_2 \mid \\
 & & \texttt{rec}\,(\mathrm{X})\ sig \mid sig\ \texttt{where}\ \ell s = con \\
 & & \\
\text{Modules} & mod ::= & [con] \mid [exp] \mid [\ell \approx con : \mathrm{K}] \mid \\
 & & [\overline{\ell \triangleright \mathrm{X} = mod}] \mid \mathrm{P} \mid \\
 & & \texttt{let}\ \mathrm{X} = mod_1\ \texttt{in}\ mod_2 \mid \\
 & & \lambda(\mathrm{X} : sig).mod \mid \mathrm{P}_1(\mathrm{P}_2) \mid \\
 & & \texttt{rec}\,(\mathrm{X} : sig)\ mod \mid \\
 & & mod\ \texttt{:>}\ sig \mid mod\ \texttt{:}\ sig
\end{array}
$$

$$
\begin{array}{rcl}
[\![= con : \mathrm{K}]\!] & \overset{\text{def}}{=} & [\![\mathrm{K}]\!]\ \texttt{where}\ \epsilon = con \\
mod_1(mod_2) & \overset{\text{def}}{=} & \texttt{let}\ \mathrm{X}_1 = mod_1\ \texttt{in} \\
 & & \quad \texttt{let}\ \mathrm{X}_2 = mod_2\ \texttt{in}\ \mathrm{X}_1(\mathrm{X}_2)
\end{array}
$$

Figure 3. RMC Syntax

The signatures of RMC structures have the form $[\![\overline{\ell \triangleright \mathrm{X} : sig}]\!]$. As in ML, these structure signatures are dependently-typed, with each internal name X_i bound in the subsequent sig_j's. The reason for distinguishing between *external* labels ℓ (which are immutable) and *internal* variables X (which are α-convertible) is explained by Harper and Lillibridge (1994). Although SML does not make such a syntactic distinction, I maintain it here in order to simplify the presentation of typechecking. I also assume for simplicity that all the labels $\overline{\ell}$ and variables $\overline{\mathrm{X}}$ are distinct.

The signature $[\![\ell \approx con : \mathrm{K}]\!]$ represents a *non-recursive* SML `datatype` specification. It describes a module providing a type component ℓ of kind K that is isomorphic to con. (Following the style of Harper and Stone (2000), this isomorphism is witnessed via two value components—a data constructor called `in` and a data destructor called `out`—that the module also provides.)

The modeling of *recursive* `datatype` specifications is achieved by using the `datatype` signature $[\![\ell \approx con : \mathrm{K}]\!]$ in conjunction with a recursively dependent signature (rds), written $\texttt{rec}\,(\mathrm{X})\ sig$. For example, if we were to add unit, sum, and product types to the language, we could model the SML `datatype` declaration

```
datatype 'a list = Nil | Cons of 'a * 'a list
```

as

$$\texttt{rec}\,(\mathrm{X})\ [\![\texttt{list} \approx \lambda(\alpha).1 + \alpha \times \mathrm{X}.\texttt{list}(\alpha) : \mathbf{T} \rightarrow \mathbf{T}]\!]$$

To be able to *use* such a `datatype`, of course, the term language needs a mechanism for data constructor application, as well as data destructor application (pattern matching). For space reasons, I omit

[6] This is in stark contrast to Crary et al.'s original proposal for rds's, which *requires* them to be fully transparent specifically so that they *can* be implemented internally using equi-recursive types. No subsequent proposal has followed their approach.

$$
\begin{array}{lrl}
\text{Type Constructors} & \mathrm{A}, \mathrm{B}, \tau ::= & \alpha \mid b \mid \lambda(\overline{\alpha}).\tau \mid \mathrm{A}(\overline{\tau}) \\
\text{Base Types} & b ::= & \forall[\overline{\alpha}].\, \tau_1 \Rightarrow \tau_2 \mid \ldots \\
\text{Signatures} & \Sigma ::= & [\![= \mathrm{A} : \mathrm{K}]\!] \mid [\![\tau]\!] \mid [\![\overline{\ell : \Sigma}]\!] \mid \\
 & & \forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).(\mathcal{L}_1 ; \Sigma_1) \\
 & & \to \exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).(\mathcal{L}_2 ; \Sigma_2) \\
\text{Type Locators} & \mathcal{L} ::= & \{\overline{\alpha : \mathrm{K} \mapsto \ell s}\} \\
\text{Type Substitutions} & \delta ::= & \{\overline{\alpha \mapsto \mathrm{A}}\} \\
\text{Type Contexts} & \Delta ::= & \emptyset \mid \Delta, \alpha \uparrow \mathrm{K} \mid \Delta, \alpha \downarrow \mathrm{K} \\
\text{Module Contexts} & \Gamma ::= & \emptyset \mid \Gamma, \mathrm{X} : \Sigma
\end{array}
$$

$$
\begin{array}{rcl}
\uparrow(\Delta) & \stackrel{\text{def}}{=} & \{\alpha \mid \alpha \uparrow \mathrm{K} \in \Delta\} \\
\Delta \,@\, \overline{\alpha} \downarrow & \stackrel{\text{def}}{=} & \Delta \backslash \{\alpha \uparrow \Delta(\alpha) \mid \alpha \in \overline{\alpha}\} \cup \{\alpha \downarrow \Delta(\alpha) \mid \alpha \in \overline{\alpha}\} \\
\ell.\mathcal{L} & \stackrel{\text{def}}{=} & \{\alpha : \mathrm{K} \mapsto \ell.\ell s \mid \alpha : \mathrm{K} \mapsto \ell s \in \mathcal{L}\} \\
\Sigma.\ell s & \stackrel{\text{def}}{=} & \begin{cases} \Sigma & \text{if } \ell s = \epsilon \\ \Sigma' & \text{if } \ell s = \ell s'.\ell \\ & \text{and } \Sigma.\ell s' = [\![\ldots, \ell : \Sigma', \ldots]\!] \\ \text{undefined} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 4. Semantic Objects and Auxiliary Constructs

this feature, as the details would closely follow Harper and Stone (2000) and are orthogonal to the focus of the present work.

Lastly, functor signatures are denoted $(\mathrm{X} : sig_1) \to sig_2$. Here, sig_1 is the argument signature, and sig_2 is the result signature, which may refer to type components of the argument via X.

Modules $[con]$ and $[exp]$ are the atomic modules representing type and value components, respectively. The syntax of structures parallels that of their signatures, but structure projection (as in SML) is limited to paths. We include a module-level `let` construct with semantics similar to SML's `local` mechanism. Functors are modeled as λ-abstractions, and, for simplicity, functor application is limited to applications of paths to paths. Figure 3 defines syntactic sugar for general applications of the form $mod_1(mod_2)$.

The syntax for `datatype` modules parallels that of `datatype` signatures. Recursive `datatype`'s are encodable using a combination of `datatype` modules and recursive modules, written `rec` $(\mathrm{X} : sig)\, mod$. For instance, to implement the `list` datatype (above), we can write:

$$
\begin{array}{l}
\texttt{rec}\,(\mathrm{X} : [\![\texttt{list} : [\![\mathbf{T} \to \mathbf{T}]\!]]\!]) \\
\quad [\texttt{list} \approx \lambda(\alpha).1 + \alpha \times \mathrm{X}.\texttt{list}(\alpha) : \mathbf{T} \to \mathbf{T}]
\end{array}
$$

Lastly, RMC provides two sealing constructs—*opaque sealing*, written mod `:>` sig, and *transparent sealing*, written $mod : sig$—which model the corresponding constructs in SML. Transparent sealing has the effect of "narrowing" mod to the target signature sig, but allows the identity of mod's type components to leak through, even if they are specified opaquely in sig.

4.2 Semantic Objects

Following the Definition of SML, the static semantics of RMC employs a language of *semantic objects*, whose syntax appears in Figure 4. As it turns out, these semantic objects are really just types/signatures (in an "internal" type system) that have been decorated with some meta-data that is useful during typechecking. That internal type system is defined in the companion technical report (Dreyer 2007a), but the static semantics of RMC can be understood perfectly well without it.

Semantic types are similar to RMC types. The only difference is that semantic types include a base type $\forall[\overline{\alpha}].\, \tau_1 \Rightarrow \tau_2$, which represents the type of `datatype` constructors and destructors.

We assume and maintain the invariant that all types are kept in β-normal form.

Semantic signatures are very similar to those in Russo's thesis (Russo 1998), which is based closely on the style of the Definition. In short, semantic signatures are fully transparent signatures. Data abstraction is handled separately via universal and existential quantification over type variables—as evidenced in the semantic functor signature—instead of via opaque type specifications in signatures.

Type locators $\mathcal{L}$ are mappings from type variables to label sequences. The purpose of type locators is discussed below.

Type substitutions δ map type variables to type constructors. In order to maintain the invariant that types are kept in normal form, type substitutions are assumed to implicitly β-normalize when they are applied.

Regarding notation: Let $\mathrm{FV}(\delta)$ mean the free variables of the type constructors in the range of δ. Also, if $\mathcal{L} = \{\overline{\alpha : \mathrm{K} \mapsto \ell s}\}$ is a type locator, then $\delta\mathcal{L}$ means $\{\overline{\delta\alpha : \mathrm{K} \mapsto \ell s}\}$.

Type contexts Δ bind type variables as either *undefined* ($\uparrow$) or *defined* ($\downarrow$). Module contexts Γ bind module variables X to semantic signatures Σ. The notation $\uparrow(\Delta)$ denotes the set of undefined type variables in Δ, and the notation $\Delta \,@\, \overline{\alpha} \downarrow$ signifies the result of changing the bindings of $\overline{\alpha}$ in Δ from undefined to defined.

4.3 Interpretation of Signatures

Figure 5 shows how RMC type constructors and signatures are interpreted in terms of their semantic counterparts.

The interpretation of type constructors is straightforward. The only interesting point is that, when interpreting a type projected from a module X, we eliminate the dependency on the module variable X. (Semantic types only depend on type variables in Δ.)

RMC signatures are interpreted as *signature denotations* of the form $\exists(\overline{\alpha \downarrow \mathrm{K}}).(\mathcal{L}; \Sigma)$. Here, $\overline{\alpha \downarrow \mathrm{K}}$ represent the opaque type components of the signature, and Σ represents the signature itself (with transparent references to $\overline{\alpha}$). The *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 RMC signature. For example, the signature `S` from Figure 1 is interpreted as

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

where Σ is as defined in Figure 2. (Note that α and β are bound variables of the denotation.) This approach to signature interpretation is modeled closely after Russo (1998). The main novelty is the presence of the type locator $\mathcal{L}$. The reason for $\mathcal{L}$ is that it makes the definition of signature matching (see below) more deterministic by telling the elaborator explicitly where to look in order to fill in the opaque type components of a signature.

The interpretation rules for signatures are standard with the exception of Rule 13 for rds's, which follows closely the informal description given in Section 3.5. The first premise of the rule computes a *shallow* denotation of sig, $\exists(\overline{\alpha_0 \downarrow \mathrm{K}_0}).(\mathcal{L}_0; \Sigma_0)$, in which *all* 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: $\exists(\overline{\alpha \downarrow \mathrm{K}}).(\mathcal{L}; \Sigma)$. 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 in Σ.

To solve this system of equations, the third premise uses the `lookup` judgment defined in Figure 6, which in turn uses the type locator $\mathcal{L}_0$ to look up the definitions of the $\overline{\alpha_0}$ in Σ and return a type substitution δ that solves for them. If there is a transparent type cycle among the definitions, the `lookup` will fail. The detection of cycles is implemented as described in Section 3.4.

$\boxed{\textbf{Well-formed type constructors: } \Delta;\Gamma\vdash \mathit{con}\leadsto \mathrm{A}:\mathrm{K}}$

$$\frac{\Delta\vdash\alpha:\mathbf{T}}{\Delta;\Gamma\vdash\alpha\leadsto\alpha:\mathbf{T}}\ (1)\qquad \frac{\Delta;\Gamma\vdash \mathrm{P}:\llbracket{=}\,\mathrm{A}:\mathrm{K}\rrbracket}{\Delta;\Gamma\vdash\mathrm{P}\leadsto\mathrm{A}:\mathrm{K}}\ (2)\qquad \frac{\Delta,\overline{\alpha\downarrow\mathbf{T}};\Gamma\vdash \mathit{con}\leadsto\tau:\mathbf{T}\qquad \overline{\alpha}=\alpha_1,\ldots,\alpha_n}{\Delta;\Gamma\vdash\lambda(\overline{\alpha}).\mathit{con}\leadsto\lambda(\overline{\alpha}).\tau:\mathbf{T}^n\rightarrow\mathbf{T}}\ (3)$$

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash \mathit{con}'\leadsto\lambda(\overline{\alpha}).\tau':\mathbf{T}^n\rightarrow\mathbf{T}\\ \overline{\Delta;\Gamma\vdash \mathit{con}\leadsto\tau:\mathbf{T}}\qquad \overline{\tau}=\tau_1,\ldots,\tau_n\end{array}}{\Delta;\Gamma\vdash \mathit{con}'(\overline{\mathit{con}})\leadsto\{\overline{\alpha\mapsto\tau}\}\tau':\mathbf{T}}\ (4)$$

…Insert rules for your favorite base types here…

$\boxed{\textbf{Well-formed terms: } \Delta;\Gamma\vdash \mathit{exp}:\tau}$

$$\frac{\Delta;\Gamma\vdash\mathrm{P}:\llbracket\tau\rrbracket}{\Delta;\Gamma\vdash\mathrm{P}:\tau}\ (5)$$

…Insert rules for your favorite core language here…

$\boxed{\textbf{Well-formed signatures: } \Delta;\Gamma\vdash \mathit{sig}\leadsto\exists(\overline{\alpha\downarrow\mathrm{K}}).(\mathcal{L};\Sigma)}$

$$\frac{}{\Delta;\Gamma\vdash\llbracket\mathrm{K}\rrbracket\leadsto\exists(\alpha\downarrow\mathrm{K}).(\{\alpha:\mathrm{K}\mapsto\epsilon\};\llbracket{=}\,\alpha:\mathrm{K}\rrbracket)}\ (6)\qquad \frac{\Delta;\Gamma\vdash \mathit{con}\leadsto\tau:\mathbf{T}}{\Delta;\Gamma\vdash\llbracket \mathit{con}\rrbracket\leadsto\exists().(\emptyset;\llbracket\tau\rrbracket)}\ (7)$$

$$\frac{\mathrm{K}=\mathbf{T}^n\rightarrow\mathbf{T}\qquad \Delta;\Gamma\vdash \mathit{con}\leadsto\mathrm{A}:\mathrm{K}\qquad \overline{\beta}=\beta_1,\ldots,\beta_n}{\Delta;\Gamma\vdash\llbracket\ell\approx \mathit{con}:\mathrm{K}\rrbracket\leadsto\exists(\alpha\downarrow\mathrm{K}).(\{\alpha:\mathrm{K}\mapsto\ell\};\llbracket\ell:\llbracket{=}\,\alpha:\mathrm{K}\rrbracket,\mathtt{in}:\llbracket\forall[\overline{\beta}].\,\mathrm{A}(\overline{\beta})\Rightarrow\alpha(\overline{\beta})\rrbracket,\mathtt{out}:\llbracket\forall[\overline{\beta}].\,\alpha(\overline{\beta})\Rightarrow\mathrm{A}(\overline{\beta})\rrbracket\rrbracket)}\ (8)$$

$$\frac{}{\Delta;\Gamma\vdash\llbracket\rrbracket\leadsto\exists().(\emptyset;\llbracket\rrbracket)}\ (9)\qquad \frac{\begin{array}{c}\Delta;\Gamma\vdash \mathit{sig}_1\leadsto\exists(\overline{\alpha_1\downarrow\mathrm{K}_1}).(\mathcal{L}_1;\Sigma_1)\\ \Delta,\overline{\alpha_1\downarrow\mathrm{K}_1};\Gamma,\mathrm{X}_1:\Sigma_1\vdash\llbracket\overline{\ell\triangleright\mathrm{X}:\mathit{sig}}\rrbracket\leadsto\exists(\overline{\alpha\downarrow\mathrm{K}}).(\mathcal{L};\llbracket\overline{\ell:\Sigma}\rrbracket)\end{array}}{\Delta;\Gamma\vdash\llbracket\ell_1\triangleright\mathrm{X}_1:\mathit{sig}_1,\overline{\ell\triangleright\mathrm{X}:\mathit{sig}}\rrbracket\leadsto\exists(\overline{\alpha_1\downarrow\mathrm{K}_1},\overline{\alpha\downarrow\mathrm{K}}).(\ell_1.\mathcal{L}_1,\mathcal{L};\llbracket\ell_1:\Sigma_1,\overline{\ell:\Sigma}\rrbracket)}\ (10)$$

$$\frac{\Delta;\Gamma\vdash \mathit{sig}_1\leadsto\exists(\overline{\alpha_1\downarrow\mathrm{K}_1}).(\mathcal{L}_1;\Sigma_1)\qquad \Delta,\overline{\alpha_1\downarrow\mathrm{K}_1};\Gamma,\mathrm{X}:\Sigma_1\vdash \mathit{sig}_2\leadsto\exists(\overline{\alpha_2\downarrow\mathrm{K}_2}).(\mathcal{L}_2;\Sigma_2)}{\Delta;\Gamma\vdash(\mathrm{X}:\mathit{sig}_1)\rightarrow \mathit{sig}_2\leadsto\exists().(\emptyset;\forall(\overline{\alpha_1\downarrow\mathrm{K}_1}).(\mathcal{L}_1;\Sigma_1)\rightarrow\exists(\overline{\alpha_2\downarrow\mathrm{K}_2}).(\mathcal{L}_2;\Sigma_2))}\ (11)$$

$$\frac{\Delta;\Gamma\vdash \mathit{sig}\leadsto\exists(\overline{\alpha\downarrow\mathrm{K}}).(\mathcal{L};\Sigma)\qquad \beta:\mathrm{L}\mapsto\ell s\in\mathcal{L}\qquad \overline{\alpha\downarrow\mathrm{K}}=\overline{\alpha_1\downarrow\mathrm{K}_1},\beta\downarrow\mathrm{L},\overline{\alpha_2\downarrow\mathrm{K}_2}\qquad \Delta;\Gamma\vdash \mathit{con}\leadsto\mathrm{B}:\mathrm{L}}{\Delta;\Gamma\vdash \mathit{sig}\ \mathtt{where}\ \ell s=\mathit{con}\leadsto\exists(\overline{\alpha_1\downarrow\mathrm{K}_1},\overline{\alpha_2\downarrow\mathrm{K}_2}).(\mathcal{L}\backslash\{\beta:\mathrm{L}\mapsto\ell s\};\{\beta\mapsto\mathrm{B}\}\Sigma)}\ (12)$$

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash\mathsf{Shal}(\mathit{sig})\leadsto\exists(\overline{\alpha_0\downarrow\mathrm{K}_0}).(\mathcal{L}_0;\Sigma_0)\\ \Delta,\overline{\alpha_0\downarrow\mathrm{K}_0};\Gamma,\mathrm{X}:\Sigma_0\vdash \mathit{sig}\leadsto\exists(\overline{\alpha\downarrow\mathrm{K}}).(\mathcal{L};\Sigma)\qquad \vdash\mathsf{lookup}\ \mathcal{L}_0\ \mathsf{in}\ \Sigma\leadsto\delta\end{array}}{\Delta;\Gamma\vdash\mathtt{rec}\,(\mathrm{X})\ \mathit{sig}\leadsto\exists(\overline{\alpha\downarrow\mathrm{K}}).(\mathcal{L};\delta\Sigma)}\ (13)$$

$\boxed{\textbf{Shallow version of a signature: } \mathsf{Shal}(\mathit{sig})}$

$$\begin{array}{rcl}\mathsf{Shal}(\llbracket\mathrm{K}\rrbracket) & \overset{\mathrm{def}}{=} & \llbracket\mathrm{K}\rrbracket\\ \mathsf{Shal}(\llbracket \mathit{con}\rrbracket) & \overset{\mathrm{def}}{=} & \llbracket\rrbracket\\ \mathsf{Shal}(\llbracket\ell\approx \mathit{con}:\mathrm{K}\rrbracket) & \overset{\mathrm{def}}{=} & \llbracket\ell:\llbracket\mathrm{K}\rrbracket\rrbracket\\ \mathsf{Shal}(\llbracket\overline{\ell\triangleright\mathrm{X}:\mathit{sig}}\rrbracket) & \overset{\mathrm{def}}{=} & \llbracket\overline{\ell:\mathsf{Shal}(\mathit{sig})}\rrbracket\end{array}\qquad \begin{array}{rcl}\mathsf{Shal}((\mathrm{X}:\mathit{sig}_1)\rightarrow \mathit{sig}_2) & \overset{\mathrm{def}}{=} & \llbracket\rrbracket\\ \mathsf{Shal}(\mathtt{rec}\,(\mathrm{X})\ \mathit{sig}) & \overset{\mathrm{def}}{=} & \mathsf{Shal}(\mathit{sig})\\ \mathsf{Shal}(\mathit{sig}\ \mathtt{where}\ \ell s=\mathit{con}) & \overset{\mathrm{def}}{=} & \mathsf{Shal}(\mathit{sig})\end{array}$$

Figure 5. Well-formedness Rules for Type Constructors, Terms, and Signatures

4.4 Static Semantics of Modules

Figure 6 shows the typing rules for modules. The main module typing judgment has the form $\Delta;\Gamma\vdash \mathit{mod}:\Sigma$ with $\overline{\alpha}\downarrow$. The judgment assumes that $\overline{\alpha}$ represent the abstract types that mod is going to define, and thus they are bound as undefined ($\overline{\alpha\uparrow\mathrm{K}}$) in the input context Δ. The shaded premises in some of the rules mark the delta between the regular typing judgment and the static typing judgment, which we discuss below. In particular, static typechecking is defined by simply removing the shaded premises and replacing all references to the regular typing judgment with the static one.

To begin with, let us focus on ordinary module typing. Rules 14 through 16 for paths and atomic modules are straightforward.

Rule 17 for `datatype` modules $[\ell\approx \mathit{con}:\mathrm{K}]$ returns a signature that matches the interpretation of the corresponding `datatype` signature $\llbracket\ell\approx \mathit{con}:\mathrm{K}\rrbracket$.

The typing rules for structures (Rules 18 and 19) are self-explanatory. One point of note is that, after the first binding of a structure ($\ell_1\triangleright\mathrm{X}_1=\mathit{mod}_1$ in Rule 19) has been typechecked, the remainder of the bindings are typechecked in a context where the abstract types defined by mod_1 are bound as *defined* (namely, $\Delta\,@\,\overline{\alpha_1}\downarrow$). To see an instance where this is relevant, look at the solution to Exercise #5**(b)** given at the end of the paper.

The typechecking of module-level `let` (Rule 20) is essentially the same as the typechecking of a structure with two bindings. The only difference is that the result signature of the `let` only exports the second of the bindings.

Rule 21 for functors is fairly straightforward as well. It is worth noting that in the signature returned for the functor, there is no type locator $\mathcal{L}_2$ for the result (we just write $\emptyset$). The main reason is that, due to the so-called *avoidance problem*, a type locator does

Well-formed modules: $\Delta; \Gamma \vdash \mathit{mod} : \Sigma\ \mathsf{with}\ \overline{\alpha}\downarrow$

We omit "with $\overline{\alpha}\downarrow$" if $\overline{\alpha} = \emptyset$ (*i.e.,* if mod does not define any abstract types).

$$\frac{\mathrm{X} : \Sigma \in \Gamma}{\Delta; \Gamma \vdash \mathrm{X}.\ell s : \Sigma.\ell s} \tag{14}$$

$$\frac{\Delta; \Gamma \vdash \mathit{con} \rightsquigarrow \mathrm{A} : \mathrm{K}}{\Delta; \Gamma \vdash [\mathit{con}] : \llbracket = \mathrm{A} : \mathrm{K} \rrbracket} \tag{15}$$

$$\frac{\Delta \vdash \tau : \mathbf{T} \qquad \colorbox{lightgray}{$\Delta; \Gamma \vdash \mathit{exp} : \tau$}}{\Delta; \Gamma \vdash [\mathit{exp}] : \llbracket \tau \rrbracket} \tag{16}$$

$$\frac{\mathrm{K} = \mathbf{T}^n \to \mathbf{T} \qquad \Delta; \Gamma \vdash \mathit{con} \rightsquigarrow \mathrm{A} : \mathrm{K} \qquad \overline{\beta} = \beta_1, \ldots, \beta_n \qquad \alpha \uparrow \mathrm{K} \in \Delta}{\Delta; \Gamma \vdash [\ell \approx \mathit{con} : \mathrm{K}] : \llbracket \ell : \llbracket = \alpha : \mathrm{K} \rrbracket, \mathtt{in} : \llbracket \forall[\overline{\beta}].\, \mathrm{A}(\overline{\beta}) \Rightarrow \alpha(\overline{\beta}) \rrbracket, \mathtt{out} : \llbracket \forall[\overline{\beta}].\, \alpha(\overline{\beta}) \Rightarrow \mathrm{A}(\overline{\beta}) \rrbracket \rrbracket\ \mathsf{with}\ \alpha\downarrow} \tag{17}$$

$$\frac{}{\Delta; \Gamma \vdash [] : \llbracket\rrbracket} \tag{18}$$

$$\frac{\Delta; \Gamma \vdash \mathit{mod}_1 : \Sigma_1\ \mathsf{with}\ \overline{\alpha_1}\downarrow \qquad \Delta \mathbin{@} \overline{\alpha_1}\downarrow; \Gamma, \mathrm{X}_1 : \Sigma_1 \vdash [\overline{\ell \triangleright \mathrm{X} = \mathit{mod}}] : \llbracket \overline{\ell : \Sigma} \rrbracket\ \mathsf{with}\ \overline{\alpha_2}\downarrow}{\Delta; \Gamma \vdash [\ell_1 \triangleright \mathrm{X}_1 = \mathit{mod}_1, \overline{\ell \triangleright \mathrm{X} = \mathit{mod}}] : \llbracket \ell_1 : \Sigma_1, \overline{\ell : \Sigma} \rrbracket\ \mathsf{with}\ \overline{\alpha_1}, \overline{\alpha_2}\downarrow} \tag{19}$$

$$\frac{\Delta; \Gamma \vdash \mathit{mod}_1 : \Sigma_1\ \mathsf{with}\ \overline{\alpha_1}\downarrow \qquad \Delta \mathbin{@} \overline{\alpha_1}\downarrow; \Gamma, \mathrm{X} : \Sigma_1 \vdash \mathit{mod}_2 : \Sigma_2\ \mathsf{with}\ \overline{\alpha_2}\downarrow}{\Delta; \Gamma \vdash \mathtt{let}\ \mathrm{X} = \mathit{mod}_1\ \mathtt{in}\ \mathit{mod}_2 : \Sigma_2\ \mathsf{with}\ \overline{\alpha_1}, \overline{\alpha_2}\downarrow} \tag{20}$$

$$\frac{\Delta; \Gamma \vdash \mathit{sig} \rightsquigarrow \exists(\overline{\alpha_1 \downarrow \mathrm{K}_1}).(\mathcal{L}_1; \Sigma_1) \qquad \Delta, \overline{\alpha_1 \downarrow \mathrm{K}_1}, \overline{\alpha_2 \uparrow \mathrm{K}_2}; \Gamma, \mathrm{X} : \Sigma_1 \vdash \mathit{mod} : \Sigma_2\ \mathsf{with}\ \overline{\alpha_2}\downarrow}{\Delta; \Gamma \vdash \lambda(\mathrm{X} : \mathit{sig}).\mathit{mod} : \forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).(\mathcal{L}_1; \Sigma_1) \to \exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).(\emptyset; \Sigma_2)} \tag{21}$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \mathrm{P}_1 : \forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).(\mathcal{L}_1; \Sigma_1) \to \exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).(\mathcal{L}_2; \Sigma_2) \qquad \Delta; \Gamma \vdash \mathrm{P}_2 : \Sigma \\ \overline{\alpha \uparrow \mathrm{K}_2} \subseteq \Delta \qquad \vdash \mathsf{lookup}\ \mathcal{L}_1\ \mathsf{in}\ \Sigma \rightsquigarrow \delta \qquad \colorbox{lightgray}{$\mathrm{FV}(\delta) \cap \uparrow(\Delta) = \emptyset \qquad \vdash \Sigma \preceq \delta\Sigma_1$} \end{array}}{\Delta; \Gamma \vdash \mathrm{P}_1(\mathrm{P}_2) : \delta\{\overline{\alpha_2 \mapsto \alpha}\}\Sigma_2\ \mathsf{with}\ \overline{\alpha}\downarrow} \tag{22}$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \mathit{sig} \rightsquigarrow \exists(\overline{\alpha \downarrow \mathrm{K}}).(\mathcal{L}; \Sigma) \qquad \Delta, \overline{\alpha \downarrow \mathrm{K}}; \Gamma, \mathrm{X} : \Sigma \models_{\mathrm{stat}} \mathit{mod} : \Sigma_{\mathrm{stat}}\ \mathsf{with}\ \overline{\beta}\downarrow \\ \vdash \mathsf{lookup}\ \mathcal{L}\ \mathsf{in}\ \Sigma_{\mathrm{stat}} \rightsquigarrow \delta \qquad \Delta; \Gamma, \mathrm{X} : \delta\Sigma \vdash \mathit{mod} : \Sigma'\ \mathsf{with}\ \overline{\beta}\downarrow \qquad \colorbox{lightgray}{$\vdash \Sigma' \preceq \delta\Sigma$} \end{array}}{\Delta; \Gamma \vdash \mathtt{rec}\ (\mathrm{X} : \mathit{sig})\ \mathit{mod} : \Sigma'\ \mathsf{with}\ \overline{\beta}\downarrow} \tag{23}$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \mathit{sig} \rightsquigarrow \exists(\overline{\alpha_0 \downarrow \mathrm{K}_0}).(\mathcal{L}_0; \Sigma_0) \qquad \Delta = \Delta', \overline{\alpha \uparrow \mathrm{K}_0} \qquad (\mathcal{L}; \Sigma) = \{\overline{\alpha_0 \mapsto \alpha}\}(\mathcal{L}_0; \Sigma_0) \\ \colorbox{lightgray}{$\Delta, \overline{\beta \uparrow \mathrm{L}}; \Gamma \models_{\mathrm{stat}} \mathit{mod} : \Sigma_{\mathrm{stat}}\ \mathsf{with}\ \overline{\beta}\downarrow \qquad \vdash \mathsf{lookup}\ \mathcal{L}\ \mathsf{in}\ \Sigma_{\mathrm{stat}} \rightsquigarrow \delta \qquad \mathrm{FV}(\delta) \cap \uparrow(\Delta) = \emptyset$} \\ \colorbox{lightgray}{$\Delta', \overline{\beta \uparrow \mathrm{L}}; \delta\Gamma \vdash \mathit{mod} : \Sigma'\ \mathsf{with}\ \overline{\beta}\downarrow \qquad \vdash \Sigma' \preceq \delta\Sigma$} \end{array}}{\Delta; \Gamma \vdash \mathit{mod} \mathrel{\mathtt{:>}} \mathit{sig} : \Sigma\ \mathsf{with}\ \overline{\alpha}\downarrow} \tag{24}$$

$$\frac{\Delta; \Gamma \vdash \mathit{sig} \rightsquigarrow \exists(\overline{\alpha \downarrow \mathrm{K}}).(\mathcal{L}; \Sigma) \qquad \Delta; \Gamma \vdash \mathit{mod} : \Sigma'\ \mathsf{with}\ \overline{\beta}\downarrow \qquad \vdash \mathsf{lookup}\ \mathcal{L}\ \mathsf{in}\ \Sigma' \rightsquigarrow \delta \qquad \colorbox{lightgray}{$\vdash \Sigma' \preceq \delta\Sigma$}}{\Delta; \Gamma \vdash \mathit{mod} : \mathit{sig} : \delta\Sigma\ \mathsf{with}\ \overline{\beta}\downarrow} \tag{25}$$

Statically well-formed modules: $\Delta; \Gamma \models_{\mathrm{stat}} \mathit{mod} : \Sigma\ \mathsf{with}\ \overline{\alpha}\downarrow$

The rules defining this static judgment are precisely the same as the rules defining the regular module typing judgment (above), *except* with the shaded premises removed, and all occurrences of the regular module typing judgment replaced by this static judgment.

Signature matching: $\vdash \Sigma_1 \preceq \Sigma_2$

$$\frac{}{\vdash \Sigma \preceq \Sigma} \tag{26}$$

$$\frac{\Sigma' = \llbracket \overline{\ell : \Sigma} \rrbracket}{\vdash \Sigma' \preceq \llbracket\rrbracket} \tag{27}$$

$$\frac{\vdash \Sigma'.\ell_1 \preceq \Sigma_1 \qquad \vdash \Sigma' \preceq \llbracket \overline{\ell : \Sigma} \rrbracket}{\vdash \Sigma' \preceq \llbracket \ell_1 : \Sigma_1, \overline{\ell : \Sigma} \rrbracket} \tag{28}$$

$$\frac{\vdash \mathsf{lookup}\ \mathcal{L}_1'\ \mathsf{in}\ \Sigma_1 \rightsquigarrow \delta_1 \qquad \vdash \Sigma_1 \preceq \delta_1\Sigma_1' \qquad \vdash \mathsf{lookup}\ \mathcal{L}_2\ \mathsf{in}\ \delta_1\Sigma_2' \rightsquigarrow \delta_2 \qquad \vdash \delta_1\Sigma_2' \preceq \delta_2\Sigma_2}{\vdash \forall(\overline{\alpha_1' \downarrow \mathrm{K}_1'}).(\mathcal{L}_1'; \Sigma_1') \to \exists(\overline{\alpha_2' \downarrow \mathrm{K}_2'}).(\mathcal{L}_2'; \Sigma_2') \preceq \forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).(\mathcal{L}_1; \Sigma_1) \to \exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).(\mathcal{L}_2; \Sigma_2)} \tag{29}$$

Abstract type lookup: $\vdash \mathsf{lookup}\ \mathcal{L}\ \mathsf{in}\ \Sigma \rightsquigarrow \delta$

$$\frac{\begin{array}{c} \mathcal{L} = \{\alpha_1 : \mathrm{K}_1 \mapsto \ell s_1, \ldots, \alpha_n : \mathrm{K}_n \mapsto \ell s_n\} \qquad \delta_0 = \emptyset \\ \forall i \in 1..n : \quad \Sigma.\ell s_i = \llbracket = \mathrm{A}_i : \mathrm{K}_i \rrbracket \qquad \mathrm{FV}(\mathrm{A}_i) \cap \mathrm{dom}(\mathcal{L}) \subseteq \{\alpha_1, \ldots, \alpha_{i-1}\} \qquad \delta_i = \delta_{i-1} \cup \{\alpha_i \mapsto \delta_{i-1}\mathrm{A}_i\} \end{array}}{\vdash \mathsf{lookup}\ \mathcal{L}\ \mathsf{in}\ \Sigma \rightsquigarrow \delta_n} \tag{30}$$

Figure 6. Typing Rules for Modules

not necessarily exist (Harper and Lillibridge 1994). In particular, it may be that some of the abstract types in $\overline{\alpha_2}$ do not correspond to any type component specified in Σ_2, so there is no way to locate them. Fortunately, there is no need to locate them—a signature only needs a type locator if one is going to match *against* it, which is not the case for the result signature Σ_2. In general, we only need to match against signatures that correspond to RMC signatures that the programmer wrote down, and such signatures always have type locators (the signature interpretation judgment shows how to compute them).

The typing rule for functor applications (Rule 22) first uses $\mathcal{L}_1$ to look up the definitions of $\overline{\alpha_1}$ in the signature Σ of the argument P_2. This results in a substitution δ, which maps the abstract type components of P_1's argument signature to their appropriate instantiations. We then check whether Σ matches the reified argument signature $\delta\Sigma_1$. We also check that the type variables $\overline{\alpha}$ we are supposed to define have the same kinds $\overline{\mathrm{K}_2}$ as the abstract types in the result signature of P_1. Finally, we check that the types we are using to fill in the definitions of $\overline{\alpha_1}$—*i.e.*, $\mathrm{FV}(\delta)$—do not depend on any undefined variables. This last condition is necessitated by the avoidance of transparent type cycles, as explained in Section 3.4.

For the next two rules, which concern recursive and opaquely sealed modules, it is useful to refer back to the algorithmic descriptions of these rules given in Section 3.1. Beginning with Rule 23: The first three premises implement step 1 of the algorithmic description, resulting in a type substitution δ that maps the abstract type components of sig to their definitions in mod. Note that the computation of the type components of mod is achieved by a call to the static typechecking judgment. Step 2 of the algorithm is achieved by simply applying δ to Σ. Steps 3 and 4 correspond to the remaining two premises, respectively.

Rule 24 for opaque sealing matches the earlier algorithmic description as follows: The first three premises implement step 1, the next two premises implement step 2, and the last two premises implement steps 3 and 4, respectively. The side condition on $\mathrm{FV}(\delta)$ requires that the internal definitions of the abstract types $\overline{\alpha}$ not depend on any undefined types. The reason for this side condition is explained in Section 3.4.

Lastly, note that Rule 24 allows the module mod to internally define a set of "local" abstract types $\overline{\beta}$. These must be added to the context explicitly because they are not in scope outside of the module. In contrast, the typing rule for transparent sealing (Rule 25) assumes that the $\overline{\beta}$ that mod wants to define are already bound in the ambient context Δ. Indeed, it is important that $\overline{\beta}$ are bound in Δ since they may appear free in the resulting signature $\delta\Sigma$. The key difference between opaque and transparent sealing is that the former leaves the opaque type components of sig abstract, while the latter uses δ to reify the specifications of those components with their definitions in mod.

Static Typechecking The static typechecking judgment is written $\Delta; \Gamma \vdash_{\mathrm{stat}} mod : \Sigma$ `with` $\overline{\alpha}\downarrow$. Static typechecking is formalized using the same rules as regular module typechecking, except that we ignore the shaded premises. This technique underscores the semantic coherence of the two typing judgments.

The purpose of static typechecking is not to ensure that mod is well-typed—it is merely to compute mod's type components. In fact, the value components of mod are not necessarily well-typed, and the types for those value components that appear in the result signature Σ may be garbage. This is fine—all that matters are the *type* components of mod, which will be reflected correctly in Σ.

This point is driven home by the first rule with a shaded premise: Rule 16, the rule for atomic term modules. With the second premise removed, the static version of this rule may assign an arbitrary type to the term exp. This renders the static Rule 16 nondeterministic, but only in a way that doesn't matter because the nondeterminism concerns the type of a *value* component. In practice, for example, when implementing static typechecking, we can infer the type `int` for all core terms, and still have a complete typechecker.

The shaded premises in the other rules are conditions that are relevant to type-correctness in general, but are not important for computing the type components of a module. In particular, all references to the signature matching judgment $\vdash \Sigma_1 \preceq \Sigma_2$ are ignored during static typechecking because this judgment is useless in computing type components. The side conditions on $\mathrm{FV}(\delta)$ are similarly ignored.

The rule with the most shaded premises is Rule 24. The reason is that, in order to determine the type components of an opaquely sealed module mod `:>` sig, we need only look at the ascribed signature sig—the premises concerning mod are irrelevant. For a transparently sealed module, on the other hand, we must look at the module's implementation since its internal type definitions leak out.

Finally, one point of note: in the static version of Rule 23, it appears that we must statically typecheck mod twice, the second time under a context to which δ has been applied. In practice, the second typechecking pass can be avoided by observing that static typing is preserved under type substitution. Thus, under context $\Gamma, \mathrm{X} : \delta\Sigma$, we know that mod will (statically) have signature $\delta\Sigma_{\mathrm{stat}}$.

Signature Matching The signature matching judgment, written $\vdash \Sigma_1 \preceq \Sigma_2$, checks whether Σ_1 can be coerced to Σ_2. The definition of this judgment in Figure 6 is fairly standard. The rules for structure signatures permit both dropping and reordering of components. The rule for functor signatures uses contravariant matching on the arguments and covariant matching on the results.

4.5 Evidence Translation and Type Soundness

In order to claim that RMC has a sound type system, I must supply a dynamic semantics and a type soundness result. Following Harper and Stone (2000), I do not provide a dynamic semantics for RMC directly. Instead, I provide an evidence translation of well-typed RMC modules into an internal language that is based closely on the RTG language discussed in Section 2.3. This makes it possible to reuse the type soundness result for RTG, and it also offers an interpretation of RMC modules in terms of more primitive constructs. Details of the internal language, its static and dynamic semantics, and its type soundness are given in the companion technical report (Dreyer 2007a).

The evidence translation judgment for modules is simply the module typing judgment appended with $\rightsquigarrow \mathrm{M}$, indicating that the module M is the internal language translation of the given RMC module. Similarly, the evidence translation for signature matching returns an internal language functor F, which coerces the source signature to the target signature.

Figure 7 displays two of the most interesting evidence translation rules, namely those for recursive and sealed modules.

In the rule for recursive modules, the body of the recursive module translates to M, and the functor F represents the coercion from Σ' (the signature of M) to $\delta\Sigma$ (the reified forward declaration signature). The internal language type system does not permit the forward declaration of a recursive module to differ from its export signature. Thus, in order to encode the more general semantics of RMC modules (as described in Section 3.2), the internal forward declaration Σ_{rec} includes both Σ' and $\delta\Sigma$. The dynamic semantics of the internal-language recursive module construct implements recursive backpatching in the style of Scheme's `letrec`.

In the rule for sealed modules, the module underneath the sealing translates to M, and the functor F represents the coercion from Σ' (the signature of M) to the sealing signature sig. In the evidence translation of the sealing, the `new` construct is used to create the local abstract types $\overline{\beta}$, and the `def` construct is used to define the

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash \mathit{sig}\leadsto\exists(\overline{\alpha\downarrow\mathrm{K}}).(\mathcal{L};\Sigma)\qquad \Delta,\overline{\alpha\downarrow\mathrm{K}};\Gamma,\mathrm{X}:\Sigma\vdash_{\overline{\mathrm{stat}}}\mathit{mod}:\Sigma_{\mathrm{stat}}\ \mathsf{with}\ \overline{\beta}\downarrow\\ \vdash \mathsf{lookup}\ \mathcal{L}\ \mathsf{in}\ \Sigma_{\mathrm{stat}}\leadsto\delta\qquad \Delta;\Gamma,\mathrm{X}:\delta\Sigma\vdash\mathit{mod}:\Sigma'\ \mathsf{with}\ \overline{\beta}\downarrow\leadsto\mathrm{M}\qquad \vdash\Sigma'\preceq\delta\Sigma\leadsto\mathrm{F}\\ \Sigma_{\mathrm{rec}}=[\![\mathtt{extern}:\Sigma',\mathtt{intern}:\delta\Sigma]\!]\qquad \mathrm{M}_{\mathrm{rec}}=[\mathtt{extern}\triangleright\mathrm{Y}=\{\mathrm{X}\mapsto\mathrm{X}_{\mathrm{rec}}.\mathtt{intern}\}\mathrm{M},\ \mathtt{intern}\triangleright\mathrm{Z}=\mathrm{F}(\mathrm{Y})]\end{array}}{\Delta;\Gamma\vdash\mathtt{rec}\,(\mathrm{X}:\mathit{sig})\,\mathit{mod}:\Sigma'\ \mathsf{with}\ \overline{\beta}\downarrow\leadsto(\mathtt{rec}\,(\mathrm{X}_{\mathrm{rec}}:\Sigma_{\mathrm{rec}})\,\mathrm{M}_{\mathrm{rec}}).\mathtt{extern}}$$

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash\mathit{sig}\leadsto\exists(\overline{\alpha_0\downarrow\mathrm{K}_0}).(\mathcal{L}_0;\Sigma_0)\qquad \Delta=\Delta',\overline{\alpha\uparrow\mathrm{K}_0}\qquad (\mathcal{L};\Sigma)=\{\overline{\alpha_0\mapsto\alpha}\}(\mathcal{L}_0;\Sigma_0)\\ \Delta,\overline{\beta\uparrow\mathrm{L}};\Gamma\vdash_{\overline{\mathrm{stat}}}\mathit{mod}:\Sigma_{\mathrm{stat}}\ \mathsf{with}\ \overline{\beta}\downarrow\qquad \vdash\mathsf{lookup}\ \mathcal{L}\ \mathsf{in}\ \Sigma_{\mathrm{stat}}\leadsto\delta\qquad \mathrm{FV}(\delta)\cap\uparrow(\Delta)=\emptyset\\ \Delta',\overline{\beta\uparrow\mathrm{L}};\delta\Gamma\vdash\mathit{mod}:\Sigma'\ \mathsf{with}\ \overline{\beta}\downarrow\leadsto\mathrm{M}\qquad \vdash\Sigma'\preceq\delta\Sigma\leadsto\mathrm{F}\end{array}}{\Delta;\Gamma\vdash\mathit{mod}\ \mathtt{:>}\ \mathit{sig}:\Sigma\ \mathsf{with}\ \overline{\alpha}\downarrow\leadsto(\mathtt{new}\ \overline{\beta\uparrow\mathrm{L}}\ \mathtt{in}\ \mathtt{def}\ \overline{\alpha:=\delta\alpha}\ \mathtt{in}\ \mathtt{let}\ \mathrm{X}=\mathrm{M}\ \mathtt{in}\ \mathrm{F}(\mathrm{X}):\Sigma)}$$

Figure 7. Evidence Translation Rules for Recursive and Sealed Modules

abstract types $\overline{\alpha}$ corresponding to the opaque type components of sig. These are the same `new` and `def` constructs on display in the encoding from Figure 2.

4.6 Decidability

It is also important for practical purposes that the RMC type system be decidable. It is mostly straightforward to show this because the typing rules are essentially syntax-directed.

There are only two points of apparent nondeterminism. One involves the types of value components in the signature returned by the static typechecking judgment. As discussed above, this nondeterminism is irrelevant because these types are never inspected. The other point of potential nondeterminism is that in certain typing rules—specifically, Rules 21 and 24—we must guess a list of abstract types that a module is going to define. In fact, no guessing is required. It is easy to define a simple prepass over a module which will determine the *unique* number, order, and kinds of the abstract types that the module can possibly define. The definition of this prepass is omitted for space reasons.

5. Related and Future Work

Earlier sections of the paper contain detailed comparisons of RMC with related work. In this final section, I discuss some other related work, and suggest directions for future work.

5.1 Are Forward Declarations a Burden?

Most existing recursive module proposals demand that the programmer supply a forward declaration signature sig when defining a recursive module of the form `rec` $(\mathrm{X}:\mathit{sig})\ \mathit{mod}$. The only one that does not is that of Nakata and Garrigue (2006), who argue that it is burdensome for the programmer to have to write such signatures down. Instead, the typechecker for their Traviata language 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, checking for cyclic type definitions using a term-rewriting algorithm, and apparently (I believe) giving globally unique names to all bound variables. Given this information, the latter pass does relatively ordinary typechecking, although part of its simplicity is due to the fact that it does not address the double vision problem.

One reason for requiring forward declarations is to make recursive module code easier to read. But the main argument for forcing the programmer to write down a forward declaration is related to the implicit support that recursive modules provide for *polymorphic recursion*. For example, consider this recursive module:

```
rec (X : sig val f : 'a -> 'a end) struct
  fun f y = ...X.f(3)...X.f(true)...
end
```

Here, `f` may refer to itself recursively via `X.f`, and each such recursive reference may instantiate the polymorphic type variable `'a` differently. In general, type inference in the presence of polymorphic recursion is undecidable (Henglein 1993). Hence, even if a forward declaration signature is not required, the programmer may need to write down a type annotation for any function that can be referenced recursively through `X`. Indeed, in order for the first pass of Nakata and Garrigue's algorithm to collect type information about value bindings in a recursive module before they have been typechecked, terms are required to be explicitly annotated with their types. This would seem to negate the benefits of not requiring a forward declaration.[7]

On balance, in the interest of simplicity, I have opted to follow the norm and require the programmer to write down a forward declaration. However, Nakata and Garrigue's complaint about burdening the programmer with unnecessary annotations remains a reasonable one. In a previous version of the RMC type system (Dreyer 2006), I attempted to alleviate this burden using a different tactic. Instead of eliminating forward declarations, I tried to use them to my advantage. In particular, I provided a mechanism, written "`seal` mod", by which the programmer could opaquely seal a module using the signature for that module that appeared in the nearest enclosing forward declaration. For instance, in the case of the example from Figure 1, one would not need to supply explicit signature ascriptions for `A` and `B`—rather, one would simply `seal` them and the type system would look to the forward declaration `S` for the appropriate sealing signatures.

In the end, the `seal` mechanism proved to be more trouble than it was worth. The formalization of `seal` required a novel form of bidirectional typechecking for modules, which, while interesting from a formal point of view, made the typing rules very tricky to follow. I hope to find a simpler way of supporting the `seal` mechanism within the type system of the present paper.

5.2 Interaction of Recursive Modules and Type Inference

Formally, the closest relative of RMC is a type system that I developed recently together with Matthias Blume for a seemingly unrelated purpose—understanding the interaction of ML modules and Damas-Milner type inference (Dreyer and Blume 2007). I will refer to this type system as DB for short.

In the paper on DB, we demonstrate that subtle aspects of the interaction of modules and type inference cause all Standard ML typecheckers to be incomplete with respect to the Definition of SML. We show how to regain complete type inference by loos-

[7] Nakata and Garrigue briefly sketch a type inference algorithm they have implemented to avoid requiring explicitly-typed core terms in practice, but it cannot possibly succeed in all cases due to the undecidability of inference in the presence of polymorphic recursion (Henglein 1993).

ening the declarative definition of typing. Interestingly, the more liberal declarative semantics of DB makes critical use of the RTG language, even though DB does not support recursive modules.

While there are great formal similarities between RMC and DB, there are a few major differences. One is that RMC supports recursive modules. Another is RMC's novel formalization of the regular and static module typechecking judgments using the same set of typing rules. Typechecking in DB does not require any static typechecking judgment due to the lack of recursive modules.

Ironically, the completeness result for DB type inference does *not* hold up if one (naïvely) extends the language with RMC's recursive modules. Consider the following code:

```
signature S = sig type t; val v : t end
structure Foo =
rec (X : sig structure A : S end) struct
  val f = (print "Hello"; fn x => x)
  structure A :> S =
    struct type t = int; val v = f 3 end
end
```

Due to ML's (and DB's) *value restriction*, the type of f here cannot be polymorphically generalized, but f can be declaratively assigned any monomorphic type of the form $\tau \rightarrow \tau$. However, there are two inequivalent monotypes, int → int and X.A.t → X.A.t, which would both be valid types for f given its subsequent use inside the body of A. Thus, we lose the ability to assign a principal signature to the module Foo.

In essence, the problem here is that RMC's solution to the double vision problem conflates the types X.A.t and int inside the definition of A, but it does not conflate them outside of A, and this confuses the type inference engine. One way to handle this problem is to simply prohibit examples like the one above in which the right-hand side of a module-level val binding is side-effecting but does not have a unique type. (This is essentially the approach that is taken by the SML/NJ compiler.) I am currently investigating alternative ways of resolving this issue.

5.3 Modules and Units

Flatt and Felleisen (1998) 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 (2006) 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 an ML-style module system with recursion.

The key advantage offered by Flatt-style units is that they were designed from the beginning to support *separate compilation* of recursive components. In contrast, ML's separate compilation mechanism—the *functor*—while powerful in many respects, does not generalize naturally to support separate compilation of recursive modules. One of the benefits of basing the RMC type system on the RTG type system is that RTG provides built-in support for unit-style recursive linking. Thus, I hope that RMC will serve as a jumping-off point for future work on synthesizing the functionality of modules and units.

5.4 Applicative vs. Generative Functors

Following Standard ML, the semantics of functors in RMC is what is known as *generative*. This means that if a functor contains abstract type definitions in its body, then every application of the functor will result in the creation of fresh abstract types. In contrast, OCaml provides an *applicative* semantics of functors (Leroy 1995), in which every application of a functor to the same argument returns a module with the same abstract types.

The main reason that RMC supports generative functors is that they are simpler to account for in terms of universal and existential type quantification than applicative functors are. However, both applicative and generative semantics for functors are appropriate in different circumstances, and there have been several proposals for combining support for both in one language. (Dreyer et al. (2003) offer the most comprehensive existing proposal for doing this, as well as an overview of related work.) I am currently in the process of incorporating applicative functors into RMC.

References

Roberto Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 50–63, Atlanta, Georgia, 1999.

Derek Dreyer. Practical type theory for recursive modules. Technical Report TR-2006-07, University of Chicago, Department of Computer Science, August 2006.

Derek Dreyer. A type system for recursive modules. Technical Report TR-2007-10, University of Chicago, Department of Computer Science, July 2007a.

Derek Dreyer. Recursive type generativity. *Journal of Functional Programming*, 2007b. To appear. Original version published in *2005 ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 41–53, Tallinn, Estonia.

Derek Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2005.

Derek Dreyer and Matthias Blume. Principal type schemes for modular programs. In *European Symposium on Programming (ESOP)*, pages 441–457, Braga, Portugal, 2007.

Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, New Orleans, 2003.

Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 236–248, Montréal, Canada, 1998.

Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 123–137, Portland, Oregon, 1994.

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*, chapter 8. MIT Press, 2005.

Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*, pages 341–387. MIT Press, 2000.

Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 142–153, San Francisco, California, 1995.

Xavier Leroy. Manifest types, modules, and separate compilation. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 109–122, Portland, Oregon, 1994.

Xavier Leroy. A proposal for recursive modules in Objective Caml, 2003. Available at: `http://caml.inria.fr/about/papers.en.html`.

David MacQueen. Modules for Standard ML. In *ACM Symposium on LISP and Functional Programming*, pages 198–207, 1984.

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

John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10 (3):470–502, 1988.

Keiko Nakata and Jacques Garrigue. Recursive modules for programming. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 74–86, Portland, Oregon, 2006.

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

Scott Owens and Matthew Flatt. From structures and functors to modules and units. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 87–98, Portland, Oregon, 2006.

Claudio V. Russo. Recursive structures for Standard ML. In *International Conference on Functional Programming (ICFP)*, pages 50–61, Florence, Italy, 2001.

Claudio V. Russo. *Types for Modules*. PhD thesis, Edinburgh University, Edinburgh, Scotland, 1998. LFCS Thesis ECS–LFCS–98–389.

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

Solutions to Exercises from Section 3

Solution to Exercise #1: The key change is in the type components of the recursive module body. While `A.t` remains equal to α, the removal of `B`'s sealing means that `B.u` becomes transparently equal to `bool`. Consequently, the reified forward declaration signature that `S` is transformed to is $\{\beta \mapsto \texttt{bool}\}\Sigma$—*i.e.,* Σ from Figure 2 with `bool` substituted for β. The effect this has on the typechecking of the recursive module is simply that the identity of `X.B.u` and `bool` is visible throughout the whole recursive module definition, not just within `B`.

Solution to Exercise #2:

(a) is well-typed. Sealing `A` in this way has no effect on the information that `A` gets to see (via `X`) about its own type components. (One should certainly hope as much—if sealing caused a problem here, it would be an instance of the double vision problem!)

(b) is also well-typed. Intuitively, it ought to be, for it should not matter that mod is originally named `B` before it is named `A`. Stepping through: Since `A` is just a copy of `B`, the type components of `A` are transparently equal to the type components of `B`. Thus, when we reify the forward declaration signature, the type components of `X.A` will become visibly equal to whatever semantic type variables are being used to represent the abstract types defined by `B`. Then, when we go underneath the sealing of `B`, those variables will be substituted with their definitions inside `B`, and mod will see the same signature for `X.A` as it would have seen in the original version of the recursive module.

(c) is not necessarily well-typed. Intuitively, the reason is that mod is not underneath the sealing anymore. Although it happens that `A` is internally defined to be a copy of `B`, this information is only known *within* the sealed definition of `A`. At the point where we typecheck mod, it is not public knowledge that `A`'s (and consequently `X.A`'s) abstract type components are implemented internally by those of mod, so we run into the double vision problem. In this case, though, double vision is good. If mod were somehow able to know that `X.A`'s type components were equal to its own, the type system would not be respecting the data abstraction boundary erected by the programmer around the implementation of `A`.

Solution to Exercise #3: We could instead define the forward declaration signature `S` to be:

```
sig
  structure A : S_A
  structure B : S_B
end
```

This is sufficient to make `AB` typecheck because steps 1 and 2 of the typing rule for recursive modules (in Section 3.1) will reify the above forward declaration signature to the same Σ to which the original `S` was reified. Moreover, because RMC does not use the forward declaration signature as a sealing signature, the fact that the above signature is less transparent than the original `S` will not affect the exported signature of the module `AB`.

It is reasonable, then, to ask: do we need recursively dependent signatures at all? I would argue that we do. For example, suppose the programmer wishes to write down a source-level RMC signature representing the exported signature of `AB` (*e.g.,* if they want to parameterize another module over it). This cannot be done without the aid of recursively dependent signatures.

Solution to Exercise #4: Here is one example:

```
signature S = sig
  type ’a t; type n = int; type b = bool
end
rec (X : S) struct
  type ’a t = ’a X.t
  type n = int t
  type b = bool t
end
```

The unique solution for the type `t` is the same as before: $\lambda\alpha.\alpha$.

Solution to Exercise #5:

(a) is ill-typed because of RMC's rule that the internal definitions of abstract types must not depend on undefined type variables. At the point `A` is defined, `X.B.u` (*i.e.,* β) is bound in the context as undefined, so the definition of `t` is not allowed to depend on it.

(b) is well-typed. This might seem odd since the situation is similar to **(a)**. The difference is that the definition of `B` comes after the definition of `A`. Since typechecking processes module bindings in the order they appear syntactically, the typechecking of `B` is performed in a context in which `X.A.t` (*i.e.,* α) has already been defined (formally speaking, it is bound in the context as $\alpha \downarrow \mathbf{T}$). Since α is no longer undefined, it is fine for the internal definition of `B.u` to depend on it.

(c) is well-typed. By revealing the definition of `A.t` to be dependent on `X.B.u`, module `A` has placed the burden of ensuring absence of transparent type cycles on `B`. Since `B.u` is defined as `bool`, there is no problem.

(d) is ill-typed. Here, there is actually a transparent type cycle, which manifests itself as a type error during the typechecking of `B`. Specifically, the internal definition of the abstract type `B.u` depends on `X.A.t`, which is transparently equal to `int * X.B.u`, which equals `int *` β. Since β is what `B` is supposed to be defining, this constitutes a cyclic type definition, which is prohibited.